ATARI XL/XE · MOS 6502 · MAD ASSEMBLER

Asembler 6502 dla Atari XL/XE

Kompletny kurs programowania w MAD Assemblerze

Czyste mnemoniki — bez makr i pseudorozkazów

```
        org  $2000
start   lda  #$00       ; A = 0
        sta  $02C6      ; czarne tło ekranu
        lda  #$0F       ; A = 15
        sta  $02C5      ; białe litery
        rts             ; powrót do DOS-u
```

Rozkazy procesora · arytmetyka 8, 16 i 32 bity · mnożenie i dzielenie
pętle i instrukcje warunkowe · kopiowanie pamięci · system operacyjny Atari
grafika, dźwięk, przerwania, dyskietka · pamięć dodatkowa i RAM pod ROM-em

Wydanie I, październik 2026 · opracowanie: Claude (Anthropic)

Spis treści

I Podstawy 9

1 Zanim zaczniemy 10
- 1.1 Dla kogo jest ta książka . . . 10
- 1.2 Jak czytać listingi . . . 10
- 1.3 Potrzebne narzędzia . . . 11
- 1.4 Pierwszy program . . . 11
- 1.5 Asemblacja i uruchomienie . . . 12
- 1.6 Jak uczyć się z tej książki . . . 13

2 Bity, bajty i systemy liczbowe 14
- 2.1 Bit i bajt . . . 14
- 2.2 System szesnastkowy . . . 14
- 2.3 Słowo: liczby 16-bitowe i kolejność bajtów . . . 15
- 2.4 Liczby ze znakiem: kod uzupełnień do dwóch . . . 15
- 2.5 Przeniesienie i przepełnienie . . . 16
- 2.6 Kod BCD . . . 16
- 2.7 Operacje logiczne na bitach . . . 16
- 2.8 Przesunięcia bitów . . . 17

3 Procesor 6502 od środka 18
- 3.1 Model programowy . . . 18
- 3.2 Rejestr znaczników P . . . 19
- 3.3 Pamięć . . . 19
- 3.4 Jak procesor wykonuje program . . . 20
- 3.5 Przerwania w pigułce . . . 20

4 MAD Assembler — składnia bez makr 21
- 4.1 Budowa linii programu . . . 21
- 4.2 Liczby i znaki . . . 21
- 4.3 Etykiety . . . 22
- 4.4 Wyrażenia . . . 23
- 4.5 Dyrektywy . . . 23
- 4.6 Strona zerowa a pełny adres . . . 25
- 4.7 Akumulator jako argument . . . 25
- 4.8 Szablon programu . . . 26

5 Tryby adresowania 27
- 5.1 Implikowany . . . 27
- 5.2 Akumulator . . . 27
- 5.3 Natychmiastowy — # . . . 27

5.4 Absolutny . . . 28
5.5 Strona zerowa . . . 28
5.6 Indeksowany: absolutny,X i absolutny,Y . . . 28
5.7 Indeksowany: strona zerowa,X i strona zerowa,Y . . . 28
5.8 Pośredni indeksowany: `(zp),y` . . . 29
5.9 Indeksowany pośredni: `(zp,x)` . . . 29
5.10 Pośredni — tylko `jmp` . . . 30
5.11 Względny — rozgałęzienia . . . 30
5.12 Jak wybrać tryb — ściąga . . . 30

6 Leksykon rozkazów procesora 6502 **32**
6.1 Jak czytać opisy . . . 32
6.2 Ładowanie i zapisywanie rejestrów . . . 32
6.3 Przesyłanie między rejestrami . . . 35
6.4 Rozkazy stosu . . . 37
6.5 Arytmetyka . . . 39
6.6 Zwiększanie i zmniejszanie o jeden . . . 40
6.7 Operacje logiczne . . . 42
6.8 Przesunięcia i obroty . . . 44
6.9 Porównania . . . 47
6.10 Skoki warunkowe (rozgałęzienia) . . . 48
6.11 Skoki, podprogramy, przerwania . . . 51
6.12 Rozkazy sterujące znacznikami . . . 53
6.13 Rozkaz pusty . . . 55
6.14 Zestawienie: który rozkaz zmienia które znaczniki . . . 55
6.15 Czego w 6502 nie ma . . . 56

II Struktury sterujące **57**

7 Porównania i skoki warunkowe **58**
7.1 Porównanie z zerem — bez `cmp` . . . 58
7.2 Porównywanie liczb bez znaku . . . 58
7.3 Porównywanie liczb ze znakiem . . . 59
7.4 Badanie pojedynczych bitów . . . 59
7.5 Skoki dalekiego zasięgu . . . 60
7.6 Porównywanie liczb 16-bitowych . . . 60
7.7 Porównywanie liczb 32-bitowych . . . 61

8 Instrukcje warunkowe: IF ... THEN ... ELSE **62**
8.1 IF ... THEN . . . 62
8.2 IF ... THEN ... ELSE . . . 62
8.3 Warunki złożone: AND . . . 63
8.4 Warunki złożone: OR . . . 64
8.5 Zagnieżdżanie . . . 64
8.6 Przydatne wzorce . . . 64

9 Pętle: FOR, WHILE, REPEAT **66**
9.1 Pętla FOR licząca w dół — najprostsza i najszybsza . . . 66
9.2 Pętla FOR licząca w górę . . . 66
9.3 Pętle z licznikiem 16-bitowym . . . 67
9.4 Pętla WHILE . . . 69
9.5 Pętla REPEAT ... UNTIL . . . 69
9.6 Pętla nieskończona, BREAK i CONTINUE . . . 69
9.7 Pętle opóźniające . . . 70
9.8 Zestawienie . . . 70

10 Wybór wielowariantowy: CASE ... OF **71**
10.1 Technika 1: łańcuch porównań . . . 71
10.2 Technika 2: tablica skoków i `jmp (adres)` . . . 72
10.3 Technika 3: sztuczka z RTS . . . 72
10.4 Wywołanie podprogramu przez wskaźnik . . . 73
10.5 CASE jako tablica wartości . . . 73
10.6 Wyszukiwanie w tablicy kluczy . . . 73
10.7 Którą technikę wybrać . . . 74

11 Podprogramy i stos **75**
11.1 Jak działa JSR i RTS . . . 75
11.2 Opis podprogramu . . . 75
11.3 Przekazywanie parametrów . . . 76
11.4 Zwracanie wyników . . . 77
11.5 Zachowywanie rejestrów . . . 77
11.6 Podprogramy zagnieżdżone i głębokość stosu . . . 78
11.7 Rekurencja . . . 78
11.8 Biblioteka własnych procedur . . . 78

III Arytmetyka **80**

12 Dodawanie i odejmowanie: 8, 16 i 32 bity **81**
12.1 Liczby 8-bitowe . . . 81
12.2 Liczby 16-bitowe . . . 82
12.3 Liczby 32-bitowe . . . 83
12.4 Zmiana znaku (negacja) . . . 85
12.5 Liczby ze znakiem . . . 86
12.6 Najczęstsze błędy . . . 86

13 Przesunięcia, obroty i operacje na bitach **88**
13.1 Mnożenie przez 2, 4, 8... . . . 88
13.2 Dzielenie przez 2, 4, 8... . . . 88
13.3 Przesunięcie o zmienną liczbę bitów . . . 89
13.4 Obrót bajtu bez udziału przeniesienia . . . 89
13.5 Mnożenie przez stałą . . . 89
13.6 Ustawianie, kasowanie i badanie bitów . . . 90

13.7 Półbajty . . . 91
13.8 Zliczanie i przeglądanie bitów . . . 91
13.9 Bajt jako zestaw flag . . . 92

14 Mnożenie **93**
14.1 Najprostszy sposób: wielokrotne dodawanie . . . 93
14.2 Algorytm „przesuń i dodaj” . . . 93
14.3 Mnożenie 8 × 8 = 16 bitów — wersja czytelna . . . 94
14.4 Mnożenie 8 × 8 = 16 bitów — wersja szybka . . . 94
14.5 Mnożenie 16 × 16 = 32 bity . . . 95
14.6 Mnożenie 32 × 32 = 64 bity . . . 96
14.7 Mnożenie liczb ze znakiem . . . 96
14.8 Szybkie mnożenie tablicowe . . . 97
14.9 Którą procedurę wybrać . . . 97

15 Dzielenie **99**
15.1 Najprostszy sposób: wielokrotne odejmowanie . . . 99
15.2 Algorytm „przesuń i odejmij” . . . 99
15.3 Dzielenie 8 : 8 bitów . . . 100
15.4 Dzielenie 16 : 8 bitów . . . 100
15.5 Dzielenie 16 : 16 bitów . . . 101
15.6 Dzielenie 32 : 32 bity . . . 101
15.7 Dzielenie liczb ze znakiem . . . 102
15.8 Dzielenie przez stałą . . . 103
15.9 Reszta z dzielenia (modulo) . . . 103

16 Liczby dziesiętne: kod BCD i konwersje **105**
16.1 Znaki cyfr . . . 105
16.2 Liczba 8-bitowa na trzy cyfry . . . 105
16.3 Liczba 16-bitowa na pięć cyfr . . . 105
16.4 Liczba 32-bitowa na dziesięć cyfr . . . 106
16.5 Usuwanie zer wiodących . . . 107
16.6 Kod BCD i tryb dziesiętny procesora . . . 107
16.7 Zamiana liczby dwójkowej na BCD . . . 108
16.8 Wypisywanie szesnastkowe . . . 109
16.9 Zamiana tekstu dziesiętnego na liczbę . . . 109
16.10 Pakiet zmiennoprzecinkowy systemu Atari . . . 110

IV Dane **112**

17 Tablice, wskaźniki i napisy **113**
17.1 Zmienne . . . 113
17.2 Tablice bajtów . . . 113
17.3 Tablice słów . . . 114
17.4 Rekordy, czyli tablice struktur . . . 114
17.5 Tablice dwuwymiarowe . . . 114

17.6 Wskaźniki . . . 115
17.7 Napisy . . . 115
17.8 Tablice jako zamienniki obliczeń . . . 117

18 Kopiowanie, wypełnianie i porównywanie bloków pamięci 119
18.1 Bloki do 256 bajtów . . . 119
18.2 Bloki dowolnej długości . . . 120
18.3 Wypełnianie bloku dowolnej długości . . . 122
18.4 Porównywanie bloków . . . 122
18.5 Szybciej: kod samomodyfikujący się . . . 123
18.6 Przykład: przewijanie ekranu tekstowego . . . 123

V Atari XL/XE 126

19 Mapa pamięci i rejestry-cienie 127
19.1 Mapa pamięci . . . 127
19.2 Strona zerowa: co wolno zająć . . . 128
19.3 Rejestry sprzętowe i ich cienie . . . 128
19.4 Plik nagłówkowy z nazwami . . . 129

20 Plik wykonywalny: segmenty, RUN i INIT 132
20.1 Budowa pliku . . . 132
20.2 Wektor RUNAD — uruchomienie po załadowaniu . . . 132
20.3 Wektor INITAD — wykonanie w trakcie ładowania . . . 132
20.4 Zapis skrócony w MADS . . . 134
20.5 Pliki bez nagłówków . . . 134
20.6 Dobre obyczaje . . . 134

21 System operacyjny: wejście i wyjście przez CIO 136
21.1 Bloki IOCB . . . 136
21.2 Wypisywanie tekstu . . . 137
21.3 Wczytywanie tekstu z klawiatury . . . 139
21.4 Pliki dyskowe . . . 139
21.5 CIO czy bezpośredni dostęp do sprzętu? . . . 144

22 Ekran: pamięć obrazu, kolory, display list 145
22.1 Ekran tekstowy od środka . . . 145
22.2 Kolory . . . 147
22.3 Tryby graficzne . . . 148
22.4 Display list . . . 152
22.5 Własny zestaw znaków . . . 155

23 Klawiatura, joystick i klawisze konsoli 157
23.1 Klawiatura . . . 157
23.2 Klawisze konsoli: START, SELECT, OPTION . . . 159
23.3 Joystick . . . 159

23.4 Program: ruchomy znak sterowany joystickiem . . . 160

24 Przerwania: VBI, DLI, IRQ **163**
24.1 Rodzaje przerwań w Atari . . . 163
24.2 Ramka obrazu i czas . . . 163
24.3 Przerwanie VBI . . . 163
24.4 Przerwanie DLI . . . 166
24.5 Przerwania IRQ . . . 168
24.6 Sekcje krytyczne . . . 169

25 Dźwięk i liczby losowe: układ POKEY **170**
25.1 Rejestry dźwięku . . . 170
25.2 Program: gama . . . 171
25.3 Efekty dźwiękowe . . . 172
25.4 Liczby losowe . . . 173
25.5 Dźwięk klawiatury i głośnik wewnętrzny . . . 174

26 Duszki: grafika graczy i pocisków (PMG) **175**
26.1 Jak działa duszek . . . 175
26.2 Pamięć duszków . . . 175
26.3 Rejestry . . . 175
26.4 Kolizje . . . 176
26.5 Program: duszek sterowany joystickiem . . . 177
26.6 Co dalej z duszkami . . . 179

27 Stacja dysków od podszewki: SIO i sektory **180**
27.1 Dyskietka i sektory . . . 180
27.2 Warstwy oprogramowania . . . 180
27.3 Blok DCB . . . 180
27.4 Odczyt i zapis sektora . . . 182
27.5 Program: podgląd sektorów . . . 183
27.6 Rozpoznawanie formatu dyskietki . . . 185
27.7 Sektor startowy: program uruchamiany bez DOS-u . . . 186

28 Pamięć dodatkowa: PORTB, RAM pod ROM-em, banki **189**
28.1 Rejestr PORTB . . . 189
28.2 Wyłączenie BASIC-a . . . 189
28.3 Pamięć RAM pod ROM-em systemu . . . 190
28.4 Banki pamięci Atari 130XE . . . 193

29 Warsztat: debugowanie, typowe błędy, optymalizacja **200**
29.1 Debugowanie w emulatorze Altirra . . . 200
29.2 Lista kontrolna: najczęstsze błędy . . . 201
29.3 Optymalizacja szybkości . . . 202
29.4 Optymalizacja rozmiaru . . . 203
29.5 Liczenie cykli . . . 203
29.6 Nieudokumentowane rozkazy . . . 203

29.7 Styl i organizacja programu . . . 204
29.8 Co dalej . . . 204

VI Procedury w MAD Assemblerze 205

30 Procedury `.proc`: pisanie i używanie 206
30.1 Najprostsza procedura . . . 206
30.2 Etykiety lokalne . . . 206
30.3 Zmienne lokalne procedury . . . 207
30.4 Procedury zagnieżdżone . . . 208
30.5 Pomijanie nieużywanych procedur . . . 208
30.6 Kod przeznaczony dla innego adresu . . . 208
30.7 Długość procedury . . . 209
30.8 Program przykładowy . . . 209
30.9 Wywołanie z parametrami — skrót, którego nie używamy . . . 211
30.10 Zwykła etykieta, `.local` czy `.proc`? . . . 211
30.11 Dobre obyczaje . . . 212

VII Dodatki 213

A Tabele kodów rozkazów 214
A.1 Rozkazy według mnemoników . . . 214
A.2 Rozkazy według kodów . . . 215
A.3 Długości i czasy w pigułce . . . 215

B Pseudorozkazy MADS i ich rozwinięcia 216
B.1 Przesyłanie . . . 216
B.2 Arytmetyka . . . 216
B.3 Porównania i stos . . . 216
B.4 Skoki . . . 217
B.5 Inne skróty . . . 217

C Rejestry sprzętowe Atari XL/XE 218
C.1 GTIA (`$D000–$D01F`) . . . 218
C.2 POKEY (`$D200–$D20F`) . . . 218
C.3 PIA (`$D300–$D303`) . . . 219
C.4 ANTIC (`$D400–$D40F`) . . . 219
C.5 Wartości DMACTL / SDMCTL . . . 219

D Kody znaków 220
D.1 ATASCII a kody ekranowe . . . 220
D.2 Kody sterujące edytora ekranowego . . . 220

E Słowniczek 221

Cześć I

Podstawy

ROZDZIAŁ 1

Zanim zaczniemy

1.1 Dla kogo jest ta książka

Ten kurs jest przeznaczony dla osoby, która chce **naprawdę zrozumieć** programowanie procesora 6502 w komputerach Atari XL/XE i pisać programy w MAD Assemblerze (w skrócie: MADS). Nie zakładam, że znasz już asembler. Zakładam natomiast, że wiesz, czym jest program, zmienna, pętla i instrukcja warunkowa — na przykład z Turbo Basica, Pascala, C albo Pythona. Będziemy często pokazywać konstrukcję z języka wysokiego poziomu, a zaraz pod nią jej dokładny odpowiednik w rozkazach procesora.

Książka ma jedną, świadomie przyjętą zasadę:

> **Zapamiętaj**
>
> **W całym kursie używamy wyłącznie prawdziwych rozkazów procesora 6502.** MADS oferuje wiele wygodnych skrótów — pseudorozkazy (`mva`, `mwa`, `inw`, `jeq`, `add`...), makra, łączenie rozkazów dwukropkiem, dyrektywy `#if` i `#while`. Tutaj **nie używamy żadnego z nich**. Każda linia programu, która generuje kod maszynowy, to jeden z 56 oryginalnych mnemoników: `lda`, `sta`, `adc`, `bne`... Dzięki temu zawsze widzisz dokładnie to, co wykona procesor — bajt po bajcie i cykl po cyklu.

Korzystamy oczywiście z **dyrektyw** asemblera, bo bez nich nie da się napisać programu: `org` (gdzie w pamięci ma trafić kod), `equ` lub `=` (nadanie nazwy liczbie), `dta` (wstawienie danych), `icl` (dołączenie pliku). Dyrektywy nie są rozkazami procesora i niczego przed tobą nie ukrywają — tylko organizują tekst programu. Dodatek B pokazuje, na jakie prawdziwe rozkazy MADS rozwija każdy ze swoich pseudorozkazów; przyda się to przy czytaniu cudzych źródeł.

1.2 Jak czytać listingi

Wszystkie listingi mają ten sam układ trzech kolumn:

```
etykieta lda #$00              ; komentarz: co robi ten rozkaz i po co
         sta $02C6             ; każdy rozkaz ma swój komentarz
```

- **Kolumna 1 — etykieta.** Zaczyna się w pierwszym znaku linii. Etykieta to nazwa nadana miejscu w pamięci (adresowi). Może jej nie być.
- **Kolumna 2 — rozkaz** (mnemonik i jego argument). Musi być poprzedzony co najmniej jedną spacją lub tabulatorem. To ważne: MADS traktuje słowo zaczynające się w pierwszej kolumnie jako etykietę, a nie rozkaz.
- **Kolumna 3 — komentarz** po średniku. Asembler ignoruje wszystko od średnika do końca linii.

Komentarze w tej książce są celowo bardzo szczegółowe: **każda linia z rozkazem ma własny komentarz**. Spotkasz dwa ich rodzaje. Jedne mówią, *po co* dana linia istnieje („wyczyść ekran", „następny znak") — to one niosą sens programu. Drugie opisują dosłownie, *co robi* dany rozkaz („A = 0", „zapisz A do COLOR2", „skocz do petla, gdy Z = 0") — pomagają śledzić program krok po kroku, dopóki rozkazy nie staną się dla ciebie oczywiste. W opisach dosłownych stosujemy stałe sformułowania:

Zapis w komentarzu	Znaczenie
`A = licznik`	załaduj do rejestru zawartość zmiennej
`A = bajt spod adresu tab+X`	załaduj bajt spod adresu powiększonego o rejestr indeksowy
`A = bajt spod adresu (wsk) + Y`	załaduj bajt wskazywany przez wskaźnik, z przesunięciem Y
`A = młodszy bajt adresu tekst`	załaduj młodszą połowę 16-bitowego adresu etykiety
`zapisz A do licznik`	zapisz rejestr do zmiennej

Zapis w komentarzu	Znaczenie
`A = A + x + C`	dodawanie zawsze uwzględnia znacznik przeniesienia C
`A = A - x - pożyczka (gdy C = 0)`	odejmowanie uwzględnia pożyczkę, sygnalizowaną przez C = 0
`skocz do petla, gdy Z = 0 (różne / nie zero)`	skok warunkowy i badany przez niego znacznik

Listingi z nagłówkiem **„Plik: nazwa.asm”** to kompletne programy lub gotowe procedury. Wszystkie znajdziesz w archiwum dołączonym do kursu. Listingi bez nagłówka to fragmenty ilustrujące omawiane zagadnienie.

Zapamiętaj

Co zostało sprawdzone. Każdy listing z tej książki został zasemblowany programem MADS w wersji 2.1.7. Procedury arytmetyczne, konwersje liczb, operacje na napisach i blokach pamięci zostały uruchomione w emulatorze procesora 6502 i sprawdzone na tysiącach losowych danych. Kompletne programy dla Atari (ekran, klawiatura, pliki, sektory, dźwięk, przerwania, banki pamięci) zostały uruchomione w **uproszczonym modelu komputera**: emulator procesora uzupełniony o atrapy procedur systemowych i rejestrów sprzętowych. Taki test potwierdza poprawność logiki programu i sposobu wywoływania systemu, ale nie zastępuje prawdziwego sprzętu — programy **nie były uruchamiane na prawdziwym Atari ani w pełnym emulatorze** (takim jak Altirra). Przy pierwszym uruchomieniu traktuj je więc tak, jak każdy nowy kod: z debuggerem pod ręką.

1.3 Potrzebne narzędzia

MAD Assembler (MADS) — asembler skrośny autorstwa Tomasza Bieli (Tebe). Działa na komputerze PC (Windows, Linux, macOS), a produkuje pliki dla Atari. Źródła i gotowe programy wykonywalne znajdują się w repozytorium `github.com/tebe6502/Mad-Assembler`.

Edytor tekstu — dowolny, który zapisuje czysty tekst: Visual Studio Code, Notepad++, Sublime Text. Do VS Code i Notepad++ istnieją dodatki kolorujące składnię MADS.

Emulator Altirra — najdokładniejszy emulator Atari, z wbudowanym, bardzo dobrym debuggerem. Uruchamia pliki `.xex` bezpośrednio, a jeśli obok pliku wykonywalnego leżą pliki `.lab` i `.lst` wygenerowane przez MADS, debugger pokazuje twoje etykiety i linie źródła.

Prawdziwe Atari — plik `.xex` uruchomisz też na prawdziwym sprzęcie: z urządzenia typu SIDE3, SIO2SD, SDrive albo po wgraniu na dyskietkę i załadowaniu z DOS-u (w DOS 2.5 opcja `L`, w SpartaDOS po prostu wpisanie nazwy).

1.4 Pierwszy program

Zacznijmy od programu, który robi coś widocznego: zmienia kolor tła i ramki ekranu, a potem czeka na naciśnięcie dowolnego klawisza.

Plik: pierwszy.asm

```
; ------------------------------------------------------------------------
; pierwszy.asm - pierwszy program: zmiana kolorów ekranu
; Asemblacja:  mads pierwszy.asm -o:pierwszy.xex -l -t
; ------------------------------------------------------------------------

; --- nazwy adresów (stałe) - żeby nie pisać w kodzie "gołych" liczb ---------
COLOR2  = $02C6                 ; cień rejestru koloru tła w trybie tekstowym
COLOR4  = $02C8                 ; cień rejestru koloru ramki ekranu
CH      = $02FC                 ; kod ostatnio naciśniętego klawisza ($FF = żaden)

        org $2000               ; kod programu umieść od adresu $2000

start   lda #$94                ; A = $94: kolor niebieski (9), jasność 4
        sta COLOR2              ; zapisz A jako kolor tła
        lda #$46                ; A = $46: kolor czerwony (4), jasność 6
        sta COLOR4              ; zapisz A jako kolor ramki
```

```
        lda #$FF                ; A = $FF, czyli "żaden klawisz nie był naciśnięty"
        sta CH                  ; skasuj ewentualny wcześniej naciśnięty klawisz
czekaj  lda CH                  ; odczytaj kod klawisza do A
        cmp #$FF                ; czy to nadal $FF?
        beq czekaj              ; tak - nikt nic nie nacisnął, sprawdź ponownie

        lda #$FF                ; klawisz naciśnięty: skasuj go,
        sta CH                  ; żeby DOS nie odebrał go jako swojego polecenia
        rts                     ; koniec programu - powrót do DOS-u

; --- adres uruchomienia -------------------------------------------------
        org $02E0               ; $02E0-$02E1 to systemowy wektor RUNAD
        dta a(start)            ; wpisz tam adres etykiety "start" (2 bajty)
```

Przeanalizujmy ten program powoli, bo zawiera prawie wszystkie elementy, z których składa się każdy program w asemblerze.

Stałe. Linie `COLOR2 = $02C6` niczego nie umieszczają w pamięci. Mówią asemblerowi: „gdy zobaczysz nazwę `COLOR2`, podstaw liczbę `$02C6`". Znak dolara oznacza liczbę szesnastkową (o systemach liczbowych jest następny rozdział).

Dyrektywa `org`. Program musi gdzieś w pamięci leżeć. `org $2000` informuje asembler, że następny bajt kodu ma trafić pod adres `$2000`. Dlaczego akurat tam? Bo to bezpieczne miejsce powyżej obszaru zajmowanego przez typowy DOS.

Rozkazy. `lda #$94` to *LoaD Accumulator* — załaduj do akumulatora (rejestru A) liczbę `$94`. Znak `#` oznacza „tę właśnie liczbę", a nie „zawartość komórki o tym adresie". `sta COLOR2` to *STore Accumulator* — zapisz zawartość akumulatora do komórki pamięci o adresie `COLOR2`. W 6502 nie ma rozkazu „wpisz liczbę do komórki pamięci". Zawsze robimy to w dwóch krokach: liczba do rejestru, rejestr do pamięci.

Pętla oczekiwania. `lda CH` (tym razem bez `#`) odczytuje *zawartość komórki* `CH`. `cmp #$FF` porównuje akumulator z liczbą `$FF`, a `beq czekaj` (*Branch if EQual*) skacze do etykiety `czekaj`, jeśli porównywane wartości były równe. Te trzy linie to odpowiednik pętli `REPEAT UNTIL CH<>255`.

Koniec programu. `rts` (*ReTurn from Subroutine*) kończy program. DOS uruchamia nasz kod jak podprogram, więc `rts` oddaje sterowanie z powrotem do DOS-u.

Adres uruchomienia. Ostatnie dwie linie wpisują adres etykiety `start` do komórek `$02E0` i `$02E1`. System operacyjny po załadowaniu całego pliku skacze pod adres zapisany w tym miejscu. Szczegóły formatu pliku poznasz w rozdziale 20.

> **Wskazówka**
>
> MADS ma dyrektywę `run start`, która robi dokładnie to samo, co dwie ostatnie linie (`org $02E0` i `dta a(start)`). W tej książce używamy pełnego zapisu, żeby było widać, że nie ma w tym żadnej magii — to po prostu dwa bajty wpisane pod znany systemowi adres.

1.5 Asemblacja i uruchomienie

W wierszu poleceń, w katalogu z plikiem źródłowym, wpisz:

```
mads pierwszy.asm -o:pierwszy.xex -l -t
```

- `-o:pierwszy.xex` — nazwa pliku wynikowego (bez tej opcji powstanie `pierwszy.obx`),
- `-l` — utwórz listing `pierwszy.lst`,
- `-t` — utwórz tablicę etykiet `pierwszy.lab`.

Jeżeli w programie nie ma błędów, MADS wypisze podsumowanie w rodzaju:

```
Writing listing file...
Writing object file...
30 lines of source assembled in 2 pass
40 bytes written to the object file
```

Plik `pierwszy.xex` przeciągnij na okno Altirry (lub wybierz *File* → *Boot Image*). Ekran zmieni kolory, a po naciśnięciu klawisza program się zakończy.

1.5.1 Czytanie listingu

Otwórz plik `pierwszy.lst`. Jego fragment wygląda tak:

```
13 FFFF> 2000-201B> A9 94   start   lda #$94
14 2002 8D C6 02                    sta COLOR2
15 2005 A9 46                       lda #$46
16 2007 8D C8 02                    sta COLOR4
```

Od lewej: numer linii źródła, **adres**, pod którym leży rozkaz, **bajty kodu maszynowego** i tekst źródłowy. Widać tu rzecz fundamentalną: rozkaz `lda #$94` to dwa bajty — `A9` (kod rozkazu „LDA natychmiastowy") i `94` (argument). Rozkaz `sta COLOR2` to trzy bajty: `8D` oraz adres `$02C6` zapisany **od młodszego bajtu** (`C6 02`). Asembler nie robi nic więcej: zamienia mnemoniki na liczby. Jeden mnemonik — jeden rozkaz procesora — od jednego do trzech bajtów.

Zapis `FFFF> 2000-201B>` w pierwszej linii to informacja o nagłówku pliku Atari i o zakresie adresów bloku; wrócimy do tego w rozdziale 20.

> **Wskazówka**
>
> Nawyk zaglądania do pliku `.lst` jest bardzo cenny. Gdy program nie działa, listing pokazuje czarno na białym, jakie bajty powstały i pod jakimi adresami — na przykład czy asembler użył krótkiego adresowania strony zerowej (2 bajty), czy pełnego adresu (3 bajty).

1.6 Jak uczyć się z tej książki

1. **Przepisuj i uruchamiaj.** Każdy program wpisz, zasembluj i uruchom. Potem coś w nim zmień i sprawdź, czy efekt jest zgodny z przewidywaniem.
2. **Śledź w debuggerze.** W Altirze naciśnij F8, aby zatrzymać program, i wykonuj go rozkaz po rozkazie (F11), obserwując rejestry i znaczniki. Nic tak nie uczy asemblera, jak patrzenie na zmieniające się rejestry.
3. **Czytaj rozdziały po kolei** w częściach I–IV. Część V (Atari) można czytać wybiórczo, według potrzeb.
4. **Wracaj do leksykonu rozkazów** (rozdział 6) za każdym razem, gdy nie masz pewności, jak rozkaz wpływa na znaczniki. To najczęstsze źródło błędów.

ROZDZIAŁ 2

Bity, bajty i systemy liczbowe

Procesor 6502 jest **ośmiobitowy**: jego rejestry mieszczą 8 bitów, a wszystkie operacje arytmetyczne wykonuje na ośmiu bitach naraz. Żeby pisać w asemblerze, trzeba swobodnie poruszać się w systemie dwójkowym i szesnastkowym. Ten rozdział jest krótki, ale bez niego dalsze byłyby niezrozumiałe.

2.1 Bit i bajt

Bit to jedna cyfra dwójkowa: 0 albo 1. **Bajt** to osiem bitów. Bity w bajcie numerujemy od prawej, od zera:

```
numer bitu:    7     6     5     4     3     2     1     0
               ┌─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┐
wartość:       │  0  │  1  │  0  │  0  │  0  │  1  │  0  │  1  │
               └─────┴─────┴─────┴─────┴─────┴─────┴─────┴─────┘
waga bitu:    128    64    32    16     8     4     2     1

wartość bajtu = 64 + 4 + 1 = 69
```

Bit 7 nazywamy **najstarszym** (najbardziej znaczącym), bit 0 — **najmłodszym**. Bajt może przyjąć 256 różnych wartości: od 0 do 255.

Cztery bity to **półbajt** (ang. *nibble*). Bajt składa się ze starszego półbajtu (bity 7–4) i młodszego (bity 3–0).

2.2 System szesnastkowy

Zapis dwójkowy jest długi, a dziesiętny nie pokazuje układu bitów. Dlatego w asemblerze króluje system **szesnastkowy** (heksadecymalny). Ma 16 cyfr: 0–9 oraz A–F (A=10, B=11, C=12, D=13, E=14, F=15). Jego wielka zaleta: **jedna cyfra szesnastkowa to dokładnie jeden półbajt**, więc bajt to zawsze dwie cyfry.

Dziesiętnie	Dwójkowo	Szesnastkowo	Dziesiętnie	Dwójkowo	Szesnastkowo
0	`0000`	`0`	8	`1000`	`8`
1	`0001`	`1`	9	`1001`	`9`
2	`0010`	`2`	10	`1010`	`A`
3	`0011`	`3`	11	`1011`	`B`
4	`0100`	`4`	12	`1100`	`C`
5	`0101`	`5`	13	`1101`	`D`
6	`0110`	`6`	14	`1110`	`E`
7	`0111`	`7`	15	`1111`	`F`

Przykład: bajt `%01000101` dzielimy na półbajty `0100` i `0101`, czyli `4` i `5` — szesnastkowo `$45`. W drugą stronę: `$C6` to `1100` i `0110`, czyli `%11000110`.

W MADS (i w całej tej książce) systemy rozróżniamy przedrostkiem:

Zapis	System	Przykład	Wartość dziesiętna
bez przedrostka	dziesiętny	`197`	197
`$`	szesnastkowy	`$C5`	197
`%`	dwójkowy	`%11000101`	197
`'A'`	kod znaku ATASCII	`'A'`	65

Kiedy którego używać? **Szesnastkowego** — do adresów i większości wartości. **Dwójkowego** — gdy ważne są pojedyncze bity (maski, rejestry sprzętowe). **Dziesiętnego** — dla liczb „ludzkich": liczników, rozmiarów, współrzędnych.

2.3 Słowo: liczby 16-bitowe i kolejność bajtów

Jeden bajt to za mało, by zapisać adres w pamięci (Atari ma 65 536 komórek) czy większą liczbę. Dwa bajty tworzą **słowo** 16-bitowe o wartościach 0–65535 (`$0000`–`$FFFF`). Słowo składa się z **młodszego bajtu** (ang. *low*, w skrócie LO) i **starszego bajtu** (*high*, HI):

```
liczba $1234  =  starszy bajt $12 , młodszy bajt $34
wartość       =  $12 * 256 + $34  =  18 * 256 + 52  =  4660
```

Zapamiętaj

Procesor 6502 zapisuje liczby wielobajtowe w kolejności **little endian**: **najpierw młodszy bajt, potem starszy**. Adres `$1234` leży w pamięci jako dwa kolejne bajty: `$34`, `$12`. Dotyczy to adresów wewnątrz rozkazów, wektorów systemowych i — z własnego wyboru, dla spójności — wszystkich naszych zmiennych 16- i 32-bitowych.

Liczba 32-bitowa `$12345678` zajmuje więc cztery kolejne komórki: `$78`, `$56`, `$34`, `$12`. Taki układ jest wygodny: dodawanie zaczynamy od najmłodszego bajtu, który ma najniższy adres.

Strona pamięci. Starszy bajt adresu nazywamy numerem **strony**, a młodszy — pozycją na stronie. Pamięć dzieli się na 256 stron po 256 bajtów. Adres `$0600` to początek strony 6, adres `$00CB` leży na **stronie zerowej**. Pojęcie strony będzie wracało wielokrotnie, bo procesor 6502 traktuje stronę zerową i stronę pierwszą w sposób specjalny.

2.4 Liczby ze znakiem: kod uzupełnień do dwóch

Bajt może też przedstawiać liczbę ujemną. Stosuje się **kod uzupełnień do dwóch** (U2). Zasada: najstarszy bit ma wagę **ujemną**. W bajcie bit 7 waży −128 zamiast +128:

```
%11111111 = -128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = -1
%10000000 = -128                                 = -128
%01111111 =        64 + 32 + 16 + 8 + 4 + 2 + 1 = +127
%11111110 = -128 + 126                           = -2
```

Zakres bajtu ze znakiem to −128...+127, słowa ze znakiem: −32768...+32767. Bit 7 (lub 15 w słowie) pełni rolę **bitu znaku**: 1 oznacza liczbę ujemną.

Jak zmienić znak liczby? **Zaneguj wszystkie bity i dodaj 1.** Przykład dla 5:

```
 5  = %00000101
neg = %11111010     (każdy bit odwrócony)
+1  = %11111011  =  $FB  =  -5
```

Genialność kodu U2 polega na tym, że **procesor nie musi wiedzieć, czy liczba jest ze znakiem**. Dodawanie i odejmowanie działają identycznie dla liczb bez znaku i ze znakiem — układ bitów wyniku jest ten sam. Różnica pojawia się dopiero przy **interpretacji wyniku** (czy wystąpiło przepełnienie) i przy **porównaniach**. Bajt `$FB` to 251 albo −5, zależnie od tego, jak *ty* zdecydujesz go traktować.

Bajt	Bez znaku	Ze znakiem (U2)
`$00`	0	0
`$01`	1	+1
`$7F`	127	+127
`$80`	128	−128
`$81`	129	−127
`$FE`	254	−2
`$FF`	255	−1

2.5 Przeniesienie i przepełnienie

Dodajmy dwa bajty bez znaku: 200 + 100 = 300. Wynik nie mieści się w bajcie:

```
  %11001000    (200)
+ %01100100    (100)
-------------
1 %00101100    (44, a dziewiąty bit "wypadł" na zewnątrz)
```

W bajcie zostaje 44 (bo 300 − 256 = 44), a dziewiąty bit trafia do specjalnego jednobitowego znacznika procesora — **przeniesienia** (ang. *Carry*, znacznik C). Przeniesienie to klucz do arytmetyki wielobajtowej: przy dodawaniu starszych bajtów procesor dolicza je automatycznie. Dokładnie tak, jak w dodawaniu pisemnym „jeden w pamięci".

Dla liczb ze znakiem problemem jest co innego. 100 + 100 = 200, ale 200 nie mieści się w zakresie −128...+127:

```
  %01100100    (+100)
+ %01100100    (+100)
-------------
  %11001000    (to jest -56 w kodzie U2 - wynik błędny!)
```

Dodaliśmy dwie liczby dodatnie i wyszła ujemna. Taką sytuację sygnalizuje drugi znacznik — **nadmiar**, czyli przepełnienie (ang. *oVerflow*, znacznik V). Zapamiętaj rozróżnienie:

- **C (przeniesienie)** — wynik nie zmieścił się jako liczba **bez znaku**,
- **V (nadmiar)** — wynik nie zmieścił się jako liczba **ze znakiem**.

2.6 Kod BCD

Jest jeszcze jeden sposób zapisu liczb, który procesor 6502 obsługuje sprzętowo: **BCD** (*Binary Coded Decimal*). Każdy półbajt przechowuje jedną cyfrę dziesiętną 0–9. Bajt `$47` w kodzie BCD oznacza liczbę czterdzieści siedem, a nie 71. Bajt mieści więc wartości 00–99.

BCD jest wygodny tam, gdzie liczbę trzeba często **wyświetlać** — na przykład jako licznik punktów w grze — bo zamiana na cyfry jest banalna: starszy półbajt to pierwsza cyfra, młodszy to druga. Arytmetykę BCD omawia rozdział 16.

2.7 Operacje logiczne na bitach

Trzy podstawowe działania logiczne wykonuje się na każdej parze odpowiadających sobie bitów niezależnie:

A	B	A AND B	A OR B	A XOR B
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

W praktyce każda z tych operacji ma swoje typowe zastosowanie:

- **AND — kasowanie bitów (maskowanie).** Tam, gdzie w masce jest 0, wynik jest zerem; gdzie 1 — bit zostaje bez zmian. `x AND %00001111` zostawia tylko młodszy półbajt.
- **OR — ustawianie bitów.** Gdzie w masce jest 1, wynik jest jedynką. `x OR %10000000` ustawia bit 7.
- **XOR — odwracanie bitów.** Gdzie w masce jest 1, bit zmienia się na przeciwny. `x XOR %11111111` neguje cały bajt.

```
wartość      %10110110        %10110110        %10110110
maska    AND %00001111    OR  %00001111    XOR %00001111
         ---------------  ---------------  ---------------
```

```
wynik       %00000110        %10111111        %10111001
           (skasowane       (ustawione       (odwrócone
            bity 7-4)        bity 3-0)        bity 3-0)
```

2.8 Przesunięcia bitów

Przesunięcie wszystkich bitów o jedną pozycję **w lewo** mnoży liczbę przez 2, a **w prawo** — dzieli ją przez 2 (bez reszty):

```
%00010110  = 22
w lewo:   %00101100  = 44      (22 * 2)
w prawo:  %00001011  = 11      (22 / 2)
```

To dokładny odpowiednik dopisania lub skreślenia zera w systemie dziesiętnym. Ponieważ 6502 nie ma rozkazów mnożenia i dzielenia, przesunięcia są ich podstawowym budulcem — poświęcimy im rozdziały 13–15.

Ćwiczenia

1. Zamień na system szesnastkowy: `%11110000`, `%10100101`, 255, 1000.
2. Zamień na dziesiętny: `$20`, `$80`, `$0600`, `$D40A`.
3. Jaką liczbę ze znakiem przedstawia bajt `$F0`? A słowo `$FFF0`?
4. Jakie dwa bajty i w jakiej kolejności leżą w pamięci, jeśli zapisano tam adres `$E456`?
5. Oblicz: `$5A AND $0F`, `$5A OR $0F`, `$5A XOR $FF`.

Odpowiedzi: (1) `$F0`, `$A5`, `$FF`, `$03E8`. (2) 32, 128, 1536, 54282. (3) −16; −16. (4) `$56`, potem `$E4`. (5) `$0A`, `$5F`, `$A5`.

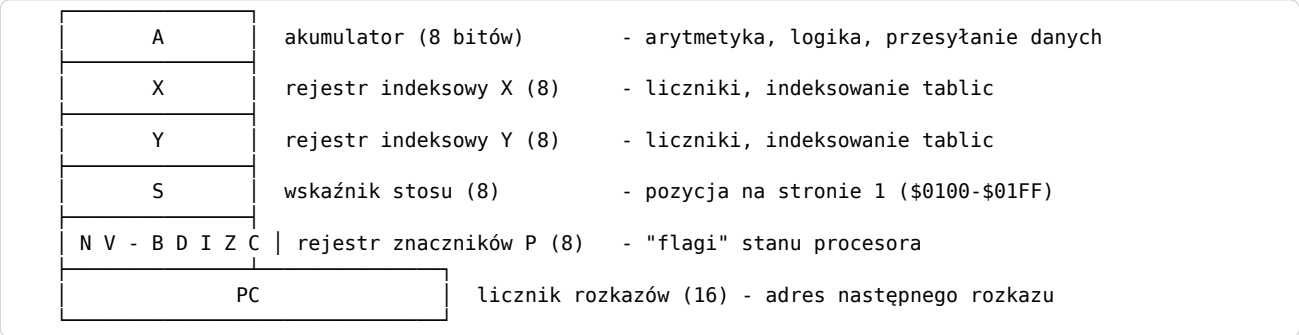
ROZDZIAŁ 3

Procesor 6502 od środka

3.1 Model programowy

W Atari XL/XE pracuje procesor **6502C „Sally”** — wariant klasycznego MOS 6502 z dodatkowym wejściem pozwalającym układowi ANTIC zatrzymywać procesor. Z punktu widzenia programisty to zwykły 6502. Taktowany jest zegarem ok. 1,77 MHz (w europejskim systemie PAL), co oznacza niespełna 1,8 miliona **cykli** na sekundę. Każdy rozkaz zajmuje od 2 do 7 cykli.

Cały stan procesora, który widzi programista, to sześć rejestrów:



To naprawdę wszystko. Trzy rejestry robocze po 8 bitów — dlatego programowanie 6502 polega w dużej mierze na przekładaniu danych między rejestrami a pamięcią.

3.1.1 Akumulator A

Główny rejestr roboczy. **Tylko na akumulatorze** można wykonywać dodawanie (`adc`), odejmowanie (`sbc`) i operacje logiczne (`and`, `ora`, `eor`). Wynik zawsze zostaje w akumulatorze.

3.1.2 Rejestry indeksowe X i Y

Służą przede wszystkim jako **liczniki pętli** i **indeksy** — ich zawartość procesor dodaje do adresu podanego w rozkazie (`lda tablica,x`). Można je zwiększać i zmniejszać o 1 (`inx`, `dex`, `iny`, `dey`), ładować, zapisywać i porównywać. **Nie można** na nich dodawać ani wykonywać operacji logicznych. X i Y nie są w pełni symetryczne: niektóre tryby adresowania działają tylko z X, inne tylko z Y (rozdział 5).

3.1.3 Licznik rozkazów PC

Jedyny rejestr 16-bitowy. Zawiera adres następnego rozkazu do wykonania. Nie ma rozkazów odczytu czy zapisu PC wprost — zmieniają go skoki (`jmp`, `jsr`, `rts`, rozgałęzienia), a po każdym zwykłym rozkazie sam przesuwa się dalej.

3.1.4 Wskaźnik stosu S

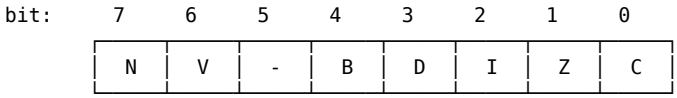
Stos to obszar pamięci służący do chwilowego odkładania danych, działający na zasadzie „ostatni wszedł — pierwszy wyjdzie” (jak stos talerzy). W 6502 stos zajmuje **zawsze stronę pierwszą**, czyli adresy `$0100–$01FF`. Rejestr S zawiera młodszy bajt adresu pierwszego **wolnego** miejsca.

- Odłożenie bajtu: zapis pod `$0100+S`, potem S zmniejsza się o 1. Stos **rośnie w dół**.
- Zdjęcie bajtu: S zwiększa się o 1, potem odczyt spod `$0100+S`.

Stos ma tylko 256 bajtów i procesor nie kontroluje jego przepełnienia — po przekroczeniu granicy S „zawija się” i zaczyna nadpisywać najstarsze dane. Stosu używają rozkazy `pha`, `pla`, `php`, `plp`, a także `jsr`/`rts` (adres powrotu z podprogramu) i mechanizm przerwań.

3.2 Rejestr znaczników P

Rejestr P to osiem bitów, z których każdy ma własne znaczenie. Nazywamy je **znacznikami** (flagami). To one są „pamięcią” procesora o wyniku ostatniej operacji i na nich opiera się **każda** decyzja w programie.



Znacznik	Nazwa	Znaczenie
C	Carry — przeniesienie	Dziewiąty bit wyniku dodawania; odwrotność „pożyczki” przy odejmowaniu; bit wysunięty przy przesunięciach; wynik porównania „większy lub równy”.
Z	Zero	Ustawiony (1), gdy wynik ostatniej operacji wyniósł zero.
I	Interrupt disable	Ustawiony blokuje przerwania maskowalne IRQ.
D	Decimal — tryb dziesiętny	Ustawiony przełącza `adc` i `sbc` na arytmetykę BCD.
B	Break	Nie jest prawdziwym bitem rejestru. Widać go tylko w kopii P odłożonej na stos: 1 po rozkazie `brk` lub `php`, 0 po przerwaniu sprzętowym.
–	(nieużywany)	W kopii na stosie zawsze 1.
V	oVerflow — nadmiar	Przepełnienie w arytmetyce liczb ze znakiem.
N	Negative — znak	Kopia bitu 7 wyniku. Dla liczb ze znakiem: 1 = wynik ujemny.

Zapamiętaj

Znaczniki zmieniają się „przy okazji”. Większość rozkazów, które coś obliczają lub ładują do rejestru (`lda`, `ldx`, `ldy`, `adc`, `and`, `inc`, `dex`, `tax`, `pla`...), ustawia znaczniki N i Z według wyniku. Nie trzeba więc pisać `cmp #0` po `lda` — znacznik Z już wie, czy załadowano zero.
Rozkazy **zapisu** do pamięci (`sta`, `stx`, `sty`) oraz skoki **nie zmieniają żadnych znaczników**. Dzięki temu można coś obliczyć, zapisać wynik, a dopiero potem podjąć decyzję na podstawie znaczników.

Dokładny wpływ każdego rozkazu na znaczniki opisuje rozdział 6. Pomylenie się w tym, który rozkaz zmienia który znacznik, to błąd numer jeden początkujących — w razie wątpliwości zawsze sprawdzaj.

3.3 Pamięć

Procesor ma 16-bitową szynę adresową, więc widzi 65 536 komórek (64 KB), o adresach `$0000`–`$FFFF`. Dla procesora wszystkie komórki są takie same: to, czy pod danym adresem jest RAM, ROM czy rejestr układu graficznego, zależy od konstrukcji komputera. Dwie strony są jednak wyróżnione przez sam procesor:

Strona zerowa (`$0000`–`$00FF`). Rozkazy odwołujące się do niej są o bajt krótsze i o cykl szybsze, bo zamiast dwóch bajtów adresu podaje się jeden. Co ważniejsze, **tylko na stronie zerowej można przechowywać wskaźniki** do adresowania pośredniego (rozdział 5). Strona zerowa jest więc najcenniejszym kawałkiem pamięci i zawsze jest jej za mało.

Strona pierwsza (`$0100`–`$01FF`). Stos.

Wektory (`$FFFA`–`$FFFF`). Sześć ostatnich bajtów pamięci to trzy adresy, pod które procesor skacze w wyjątkowych sytuacjach:

Adres	Wektor	Kiedy używany
`$FFFA–$FFFB`	NMI	przerwanie niemaskowalne (w Atari: VBI, DLI, klawisz RESET)
`$FFFC–$FFFD`	RESET	włączenie zasilania
`$FFFE–$FFFF`	IRQ/BRK	przerwanie maskowalne (w Atari: klawiatura, POKEY) lub rozkaz `brk`

W Atari te adresy leżą w pamięci ROM systemu operacyjnego. System udostępnia jednak własne wektory w pamięci RAM, które możemy podmieniać (rozdział 24).

3.4 Jak procesor wykonuje program

Praca procesora to nieskończone powtarzanie trzech kroków:

1. **Pobierz** bajt spod adresu PC — to **kod rozkazu** (ang. *opcode*). Zwiększ PC.
2. **Zdekoduj** go: kod jednoznacznie określa, jaka to operacja, jaki tryb adresowania i ile bajtów argumentu trzeba jeszcze pobrać (0, 1 lub 2). Pobierz je, zwiększając PC.
3. **Wykonaj** rozkaz. Wróć do kroku 1.

Procesor nie odróżnia kodu od danych. Jeśli PC trafi w obszar danych, procesor posłusznie zacznie je wykonywać jako rozkazy — zwykle kończy się to zawieszeniem. Dlatego dane trzymamy **poza ścieżką wykonywania**: za rozkazem `rts` lub `jmp`, nigdy „w środku" ciągu rozkazów.

Prześledźmy fragment pierwszego programu:

```
adres   bajty      rozkaz        co się dzieje
$2000   A9 94      lda #$94      pobierz $A9 -> "LDA natychmiastowy", pobierz $94;
                                 A = $94; PC = $2002
$2002   8D C6 02   sta $02C6     pobierz $8D -> "STA absolutny", pobierz $C6 i $02;
                                 zapisz A pod adres $02C6; PC = $2005
```

3.5 Przerwania w pigułce

Przerwanie to sygnał z zewnątrz, który każe procesorowi na chwilę porzucić bieżący program i wykonać specjalną procedurę obsługi. Procesor kończy wtedy aktualny rozkaz, odkłada na stos PC i rejestr P, ustawia znacznik I i skacze pod adres z odpowiedniego wektora. Procedura obsługi kończy się rozkazem `rti`, który zdejmuje ze stosu P i PC — przerwany program biegnie dalej, jakby nic się nie stało.

Są dwa rodzaje przerwań: **IRQ** (maskowalne — ignorowane, gdy znacznik I = 1) i **NMI** (niemaskowalne — procesor nie może ich zablokować). W Atari dzieją się one bez przerwy: co 1/50 sekundy system wykonuje przerwanie pionowego wygaszania ekranu (VBI). Ma to praktyczną konsekwencję: **twój program może zostać przerwany między dowolnymi dwoma rozkazami**. Zwykle nic z tego nie wynika, bo procedury systemowe zachowują rejestry. Trzeba o tym pamiętać dopiero przy operacjach wymagających precyzyjnego czasu lub przy zmianie konfiguracji pamięci (rozdziały 24 i 28).

Ćwiczenia

1. Ile różnych wartości może przyjąć rejestr S i dlaczego stos ma akurat 256 bajtów?
2. Po rozkazie `lda #$00` — jaki jest stan znaczników Z i N? A po `lda #$80`?
3. Czy rozkaz `sta $0600` zmienia znacznik Z?

Odpowiedzi: (1) 256; S ma 8 bitów, a starszy bajt adresu stosu jest na stałe równy `$01`. (2) Z=1, N=0; potem Z=0, N=1. (3) Nie — rozkazy zapisu nie zmieniają znaczników.

ROZDZIAŁ 4

MAD Assembler — składnia bez makr

MADS jest asemblerem bardzo rozbudowanym. My potrzebujemy tylko jego rdzenia: zapisu rozkazów, etykiet, wyrażeń i kilkunastu dyrektyw. Ten rozdział opisuje dokładnie ten podzbiór, którym posługuje się cała książka.

4.1 Budowa linii programu

```
etykieta lda #$00              ; komentarz
```

Linia składa się z maksymalnie trzech pól, z których każde jest opcjonalne:

1. **Etykieta** — zaczyna się w pierwszej kolumnie.
2. **Rozkaz lub dyrektywa** z argumentem — poprzedzone odstępem (spacja lub tabulator).
3. **Komentarz** — od średnika do końca linii.

> **Pułapka**
>
> Rozkaz wpisany od pierwszej kolumny MADS uzna za etykietę i zgłosi dziwnie wyglądający błąd. Zawsze zostawiaj przed rozkazem odstęp. (Opcja wiersza poleceń `-f` zmienia to zachowanie, ale lepiej trzymać się zasady.)

Wielkość liter domyślnie **nie ma znaczenia**: `LDA`, `lda` i `Lda` to ten sam rozkaz, a `Start` i `START` to ta sama etykieta. (Opcja `-c` włącza rozróżnianie wielkości liter w etykietach.) W tej książce rozkazy piszemy małymi literami, a nazwy rejestrów sprzętowych i systemowych — wielkimi.

4.1.1 Komentarze

```
; komentarz od średnika do końca linii
        lda #1                 ; komentarz za rozkazem
        lda #2 // tak też można - styl języka C
/*      komentarz blokowy
        może zajmować wiele linii */
* gwiazdka w pierwszej kolumnie także zaczyna komentarz
```

4.2 Liczby i znaki

```
        lda #65                ; liczba dziesiętna
        lda #$41               ; liczba szesnastkowa
        lda #%01000001         ; liczba dwójkowa
        lda #'A'               ; kod ATASCII znaku A (65, czyli $41)
        lda #"A"               ; kod EKRANOWY znaku A (33, czyli $21) - uwaga na różnicę!
```

Wszystkie pięć linii — poza ostatnią — ładuje do akumulatora tę samą wartość. Rozróżnienie apostrofu i cudzysłowu jest specyficzne dla MADS i bardzo przydatne: Atari stosuje dwa różne kodowania znaków. **ATASCII** to kody, którymi posługuje się system operacyjny (klawiatura, wypisywanie przez CIO, pliki). **Kody ekranowe** (wewnętrzne) to wartości, które trzeba wpisać wprost do pamięci obrazu, by znak pojawił się na ekranie. Szczegóły w rozdziale 22.

4.3 Etykiety

Etykieta to nazwa, której asembler przypisuje liczbę — najczęściej adres. Nazwa może zawierać litery, cyfry i znak podkreślenia, a zaczynać się musi od litery lub podkreślenia.

Etykieta miejsca w kodzie otrzymuje adres rozkazu lub danych, przy których stoi:

```
        org $2000               ; adres asemblacji: $2000
start   lda #0                  ; start = $2000 (adres tego rozkazu)
petla   sta $0600               ; petla = $2002 (bo "lda #0" zajął 2 bajty)
        jmp petla               ; skocz pod adres $2002
```

Etykieta może też stać sama w linii — oznacza wtedy adres tego, co nastąpi po niej:

```
dluga_nazwa_procedury
        lda #0                  ; etykieta powyżej wskazuje na ten rozkaz
        rts                     ; powrót z podprogramu
```

Stała powstaje przez przypisanie wartości znakiem `=` lub dyrektywą `equ` (oba zapisy są równoważne):

```
COLBK   = $D01A                 ; rejestr koloru tła układu GTIA
MAXY    equ 24                  ; liczba wierszy ekranu
LICZNIK = $80                   ; adres naszej zmiennej na stronie zerowej
```

Zwróć uwagę: asembler **nie rozróżnia „stałej liczbowej" i „adresu zmiennej"** — jedno i drugie to po prostu liczba. To, czy `MAXY` jest wartością, czy adresem, wynika wyłącznie z użycia: `lda #MAXY` ładuje liczbę 24, a `lda MAXY` ładuje zawartość komórki o adresie 24. Pomylenie tych dwóch zapisów to najczęstszy błąd w asemblerze 6502.

> **Pułapka**
>
> **`lda #10` a `lda 10` to dwa zupełnie różne rozkazy.** Pierwszy ładuje liczbę 10. Drugi ładuje to, co akurat znajduje się w komórce pamięci o adresie 10. Asembler nie zgłosi błędu, bo oba zapisy są poprawne. Gdy program zachowuje się „losowo", najpierw sprawdź, czy gdzieś nie zgubiłeś znaku `#`.

4.3.1 Etykiety anonimowe

Krótkie pętle i przeskoki nie zawsze zasługują na własną nazwę. MADS pozwala użyć etykiety anonimowej `@`. Odwołanie `@-` oznacza „najbliższa etykieta `@` wstecz", a `@+` — „najbliższa w przód":

```
        ldx #9                  ; X = 9 - licznik
@       lda zrodlo,x            ; pobierz bajt z tablicy źródłowej
        sta cel,x               ; zapisz do tablicy docelowej
        dex                     ; X = X - 1
        bpl @-                  ; skocz do najbliższego @ powyżej, dopóki X >= 0
```

W książce używamy ich oszczędnie; w dłuższych przykładach nazwane etykiety lepiej tłumaczą, co się dzieje.

4.3.2 Obszary lokalne

W dużym programie szybko brakuje pomysłów na unikalne nazwy typu `petla1`, `petla2`. Dyrektywy `.local` i `.endl` tworzą obszar, w którym etykiety są lokalne — te same nazwy mogą wystąpić w innym obszarze bez konfliktu:

```
        .local czysc_ekran      ; początek obszaru lokalnego o nazwie czysc_ekran
        lda #0                  ; kod wewnątrz obszaru "czysc_ekran"
        tax                     ; X = A
petla   sta $0600,x             ; ta "petla" istnieje tylko w tym obszarze
        inx                     ; X = X + 1
        bne petla               ; skocz do petla, gdy Z = 0 (różne / nie zero)
        rts                     ; powrót z podprogramu
        .endl                   ; koniec obszaru lokalnego

        .local czysc_bufor      ; początek drugiego obszaru lokalnego
        ldx #63                 ; X = 63
petla   sta $0700,x             ; inna "petla" - bez konfliktu z poprzednią
```

```
        dex                     ; X = X - 1
        bpl petla               ; skocz do petla, gdy N = 0 (bit 7 skasowany)
        rts                     ; powrót z podprogramu
        .endl                   ; koniec obszaru lokalnego
```

Nazwa obszaru jest jednocześnie etykietą jego początku, więc można napisać `jsr czysc_ekran`. Do etykiety wewnętrznej można sięgnąć z zewnątrz, pisząc `czysc_ekran.petla`. (MADS ma też dyrektywę `.proc`, która działa podobnie i potrafi dodatkowo pomijać nieużywane procedury. Poświęcony jest jej cały rozdział 30.)

4.4 Wyrażenia

Wszędzie, gdzie asembler oczekuje liczby, można wpisać wyrażenie. Asembler oblicza je **w trakcie asemblacji** — do programu trafia gotowy wynik, a procesor nic o wyrażeniu nie wie.

Operator	Znaczenie	Przykład	Wynik
`+ - * /`	cztery działania	`40*24`	960
`%`	reszta z dzielenia	`10%3`	1
`&`	iloczyn bitowy AND	`$1234&$FF`	`$34`
`|`	suma bitowa OR	`$80|$01`	`$81`
`^`	różnica symetryczna XOR	`$FF^$0F`	`$F0`
`<<, >>`	przesunięcie w lewo, w prawo	`1<<4`	16
`<` (jednoargumentowy)	młodszy bajt	`<$1234`	`$34`
`>` (jednoargumentowy)	starszy bajt	`>$1234`	`$12`
`*` (samodzielnie)	bieżący adres asemblacji	`*-start`	długość kodu

Najważniejsze dla nas są operatory **< i >**. Procesor jest ośmiobitowy, więc adres 16-bitowy trzeba mu zawsze podawać w dwóch kawałkach:

```
        lda #<tekst             ; A = młodszy bajt adresu etykiety "tekst"
        sta $80                 ; zapisz go do komórki $80
        lda #>tekst             ; A = starszy bajt adresu etykiety "tekst"
        sta $81                 ; zapisz go do komórki $81 - para $80/$81 wskazuje teraz na "tekst"
```

Pułapka

Nawiasy w wyrażeniach to nawiasy kwadratowe. Nawiasy okrągłe w argumencie rozkazu oznaczają adresowanie pośrednie! `lda (ptr),y` to tryb pośredni, natomiast grupowanie działań zapisujemy tak: `lda #[2+3]*4` albo `lda tablica+[N-1]*2`.

4.5 Dyrektywy

4.5.1 `org` — ustalenie adresu

`org adres` ustawia adres, od którego asembler umieszcza kolejne bajty. Program może zawierać wiele dyrektyw `org` — każda rozpoczyna nowy, osobny blok pamięci (w pliku wykonywalnym Atari każdy blok jest zapisany osobno, z własnym nagłówkiem).

```
        org $2000               ; kod od $2000
start   rts                     ; powrót z podprogramu
        org $3000               ; a dane od $3000
dane    dta 1,2,3               ; dane: bajty
```

4.5.2 `dta` — wstawianie danych

Dyrektywa `dta` umieszcza w programie bajty danych. Literą przed nawiasem wybieramy rodzaj:

```
        dta 1,2,3               ; trzy bajty: $01 $02 $03
        dta b(1,2,3)            ; to samo - "b" jak bajt
        dta a($1234)            ; słowo 16-bitowe: $34 $12 (młodszy bajt pierwszy)
        dta a(start,dane)       ; dwa adresy = cztery bajty
        dta l($1234)            ; tylko młodszy bajt: $34
        dta h($1234)            ; tylko starszy bajt: $12
        dta t($123456)          ; liczba 24-bitowa: $56 $34 $12
        dta f($12345678)        ; liczba 32-bitowa: $78 $56 $34 $12
        dta c'Atari'            ; tekst w kodach ATASCII: $41 $74 $61 $72 $69
        dta d'Atari'            ; tekst w kodach ekranowych: $21 $54 $41 $52 $49
        dta c'Koniec',$9B       ; tekst, a po nim bajt $9B (znak końca linii)
```

Dostępne są też zamienniki znane z innych asemblerów — warto je rozpoznawać:

```
        .byte 1,2,'A'           ; bajty
        .word $1234,start       ; słowa 16-bitowe
        .long $123456           ; liczba 24-bitowa
        .dword $12345678        ; liczba 32-bitowa
        .by 'Atari'             ; tekst ATASCII
        .sb 'Atari'             ; tekst w kodach ekranowych
        .he 00 FF A9            ; bajty podane szesnastkowo, bez znaku $
```

Typ `l` i `h` świetnie nadaje się do budowania **tablic adresów rozdzielonych na dwie części** — osobno młodsze, osobno starsze bajty. Zobaczymy to w rozdziale 10:

```
adr_lo  dta l(proc0,proc1,proc2)  ; młodsze bajty trzech adresów
adr_hi  dta h(proc0,proc1,proc2)  ; starsze bajty tych samych adresów
```

4.5.3 `.ds` — rezerwacja miejsca

`.ds n` rezerwuje n bajtów bez nadawania im wartości. Asembler po prostu przesuwa bieżący adres:

```
bufor   .ds 128                 ; 128 bajtów na bufor sektora
licznik .ds 2                   ; zmienna 16-bitowa
```

> **Uwaga**
>
> Zarezerwowany obszar **nie jest zerowany** — będzie zawierał to, co akurat było w pamięci. Jeżeli zmienna ma mieć wartość początkową, wpisz ją rozkazami na początku programu albo użyj `dta`. Ponadto `.ds` w środku kodu dzieli plik wynikowy na dwa bloki (przed i po rezerwacji), co jest poprawne, ale warto o tym wiedzieć. Najczyściej jest umieszczać wszystkie `.ds` razem, na końcu programu.

4.5.4 `icl` i `ins` — dołączanie plików

```
        icl 'atari.asm'         ; wstaw w tym miejscu treść innego pliku źródłowego
        ins 'obrazek.dat'       ; wstaw w tym miejscu zawartość pliku binarnego (bajt w bajt)
```

`icl` pozwala podzielić program na części i trzymać w osobnym pliku na przykład listę adresów systemowych. `ins` wkleja surowe dane: grafikę, muzykę, zestaw znaków.

4.5.5 `opt` — opcje asemblacji

Najczęściej przydają się dwie:

```
        opt h+                  ; h+ generuj nagłówki pliku Atari (domyślnie włączone)
        opt h-                  ; h- czysty plik binarny bez nagłówków (np. sektor startowy)
        opt l+                  ; włącz zapis listingu od tego miejsca
```

4.5.6 Asemblacja warunkowa

Pozwala włączać lub wyłączać fragmenty programu zależnie od stałej. To nie jest instrukcja warunkowa procesora — decyzja zapada w czasie asemblacji:

```
PAL     = 1                     ; 1 = wersja dla systemu PAL, 0 = NTSC

        .if PAL                 ; jeżeli stała PAL jest różna od zera...
RAMKI   = 50                    ; w PAL jest 50 ramek obrazu na sekundę
        .else                   ; w przeciwnym razie
RAMKI   = 60                    ; w NTSC 60 ramek na sekundę
        .endif                  ; koniec bloku warunkowego
```

4.5.7 Komunikaty i kontrola

```
        .print "Dlugosc kodu: ",*-start  ; wypisz informację podczas asemblacji
        .if *>$3FFF             ; jeżeli kod przekroczył adres $3FFF...
        .error "Kod wszedl w obszar bankow pamieci!"  ; ...przerwij asemblację
        .endif                  ; koniec bloku warunkowego
```

Dyrektywa `.error` przerywa asemblację z własnym komunikatem. To znakomity sposób pilnowania założeń, których złamanie dałoby trudny do znalezienia błąd — na przykład tego, że kod nie może zachodzić na obszar `$4000`–`$7FFF` (rozdział 28).

4.5.8 Powtarzanie

```
        .rept 4                 ; powtórz zawartość bloku 4 razy
        lsr @                   ; (w kodzie znajdą się cztery kolejne rozkazy LSR)
        .endr                   ; koniec powtarzania
```

MADS ma też skrótowy zapis `:4 lsr @`. Obie formy generują zwykłe rozkazy, jeden po drugim — niczego nie ukrywają. W książce dla przejrzystości wypisujemy powtarzane rozkazy wprost.

4.6 Strona zerowa a pełny adres

Wiele rozkazów ma dwie wersje: krótszą dla adresów ze strony zerowej (`$00`–`$FF`) i dłuższą dla pozostałych. MADS wybiera wersję automatycznie na podstawie wartości adresu:

```
        lda $80                 ; adres < $100: wersja "strona zerowa", 2 bajty: A5 80
        lda $0280               ; adres >= $100: wersja "absolutna", 3 bajty: AD 80 02
```

Czasem trzeba wymusić konkretną wersję — na przykład w kodzie samomodyfikującym się, gdzie argument ma zawsze zajmować 2 bajty:

```
        lda a:$80               ; wymuś tryb absolutny: AD 80 00 (3 bajty)
        lda z:$80               ; wymuś tryb strony zerowej: A5 80 (2 bajty)
```

4.7 Akumulator jako argument

Rozkazy przesunięć (`asl`, `lsr`, `rol`, `ror`) mogą działać na komórce pamięci albo na akumulatorze. Wersję akumulatorową w MADS zapisujemy ze znakiem `@` (zapis odziedziczony po asemblerach QA i XASM) albo bez argumentu:

```
        asl @                   ; przesuń akumulator w lewo - zapis używany w tej książce
        asl                     ; to samo - sam mnemonik bez argumentu
        asl $80                 ; przesuń w lewo komórkę pamięci $80
```

> **Pułapka**
>
> Znany z wielu innych asemblerów zapis `asl a` **nie działa** w MADS — litera `a` zostałaby potraktowana jako nazwa etykiety.

4.8 Szablon programu

Większość programów w tej książce ma następujący układ. Warto go przyjąć jako własny standard:

Plik: szablon.asm

```
; ------------------------------------------------------------------------
; szablon.asm - wzorcowy układ programu dla Atari XL/XE w MAD Assemblerze
; ------------------------------------------------------------------------

; --- 1. Stałe: adresy systemowe i sprzętowe --------------------------------
RUNAD   = $02E0                 ; wektor uruchomienia programu po załadowaniu

; --- 2. Zmienne na stronie zerowej (tylko nazwy adresów, bez kodu) ----------
wsk     = $80                   ; wskaźnik 16-bitowy: zajmuje komórki $80 i $81
licz    = $82                   ; licznik 8-bitowy: zajmuje komórkę $82

; --- 3. Kod programu -------------------------------------------------------
        org $2000               ; początek programu w pamięci

start   lda #0                  ; tu zaczyna się wykonywanie
        sta licz                ; nadaj zmiennym wartości początkowe
        jsr praca               ; wywołaj podprogram
        rts                     ; powrót do DOS-u

; --- 4. Podprogramy --------------------------------------------------------
praca   inc licz                ; (tu właściwa praca programu)
        rts                     ; powrót z podprogramu

; --- 5. Dane o ustalonej zawartości ----------------------------------------
tekst   dta c'Tekst',$9B        ; napis zakończony znakiem końca linii

; --- 6. Zmienne i bufory bez wartości początkowej --------------------------
bufor   .ds 128                 ; 128 bajtów na dane

; --- 7. Adres uruchomienia -------------------------------------------------
        org RUNAD               ; wektor RUNAD ($02E0)...
        dta a(start)            ; ...otrzymuje adres początku programu
```

ROZDZIAŁ 5

Tryby adresowania

Rozkaz procesora składa się z dwóch informacji: **co** zrobić (załadować, dodać, porównać) i **z czym** — skąd wziąć argument. Sposób wskazania argumentu nazywamy **trybem adresowania**. Ten sam mnemonik `lda` to w rzeczywistości osiem różnych rozkazów maszynowych, po jednym dla każdego trybu, w którym potrafi działać.

Procesor 6502 ma 13 trybów adresowania. Ich dobre zrozumienie to połowa nauki asemblera.

Tryb	Zapis	Bajtów	Skąd pochodzi argument
implikowany	`clc`	1	wynika z samego rozkazu
akumulator	`asl @`	1	rejestr A
natychmiastowy	`lda #$44`	2	liczba zapisana w rozkazie
strona zerowa	`lda $44`	2	komórka `$0044`
strona zerowa,X	`lda $44,x`	2	komórka `($44+X) & $FF`
strona zerowa,Y	`ldx $44,y`	2	komórka `($44+Y) & $FF`
absolutny	`lda $4400`	3	komórka `$4400`
absolutny,X	`lda $4400,x`	3	komórka `$4400+X`
absolutny,Y	`lda $4400,y`	3	komórka `$4400+Y`
pośredni	`jmp ($4400)`	3	adres skoku odczytany z `$4400/$4401`
pośredni (zp,X)	`lda ($44,x)`	2	komórka wskazana przez wskaźnik w `$44+X`
pośredni (zp),Y	`lda ($44),y`	2	komórka wskazana przez wskaźnik w `$44`, plus Y
względny	`bne etykieta`	2	PC + przesunięcie ze znakiem

5.1 Implikowany

Rozkaz nie potrzebuje argumentu — wiadomo, na czym działa. Zajmuje 1 bajt.

```
        clc                     ; skasuj znacznik C
        inx                     ; X = X + 1
        tay                     ; skopiuj A do Y
        rts                     ; wróć z podprogramu
```

5.2 Akumulator

Odmiana trybu implikowanego dla czterech rozkazów przesunięć, gdy działają na rejestrze A.

```
        asl @                   ; przesuń bity akumulatora o 1 w lewo
        ror @                   ; obróć bity akumulatora w prawo przez znacznik C
```

5.3 Natychmiastowy —

Argumentem jest **liczba zapisana bezpośrednio w rozkazie**, zaraz za kodem operacji. Znak `#` czytaj jako „liczba”. Zawsze 2 bajty.

```
        lda #$20                ; A = $20 (liczba trzydzieści dwa)
        ldx #0                  ; X = 0
        cmp #'A'                ; porównaj A z kodem litery "A"
        and #%00001111          ; zostaw w A tylko cztery młodsze bity
        lda #<bufor             ; A = młodszy bajt adresu etykiety "bufor"
```

Tryb natychmiastowy służy do stałych. Nie da się w nim niczego *zapisać* — `sta #5` nie ma sensu i nie istnieje.

5.4 Absolutny

Argumentem jest **zawartość komórki pamięci** o podanym adresie 16-bitowym. Rozkaz zajmuje 3 bajty: kod i dwa bajty adresu (najpierw młodszy).

```
        lda $D20A               ; A = zawartość komórki $D20A (generator liczb losowych)
        sta $02C8               ; zapisz A do komórki $02C8 (kolor ramki)
        inc licznik             ; zwiększ o 1 zawartość komórki "licznik"
        jmp start               ; skocz pod adres "start"
```

To odpowiednik zwykłej zmiennej z języków wysokiego poziomu: `lda zmienna` to odczyt zmiennej, `sta zmienna` to przypisanie.

5.5 Strona zerowa

Działa jak tryb absolutny, ale adres ma tylko jeden bajt — starszy bajt jest domyślnie zerem. Rozkaz zajmuje 2 bajty i wykonuje się o 1 cykl szybciej.

```
        lda $80                 ; A = zawartość komórki $0080
        sta $CB                 ; zapisz A do komórki $00CB
        inc $14                 ; zwiększ komórkę $0014
```

MADS wybiera ten tryb samoczynnie, gdy adres jest mniejszy od `$100`. Zmienne używane najczęściej warto więc trzymać na stronie zerowej — program jest krótszy i szybszy. W Atari bez uruchomionego BASIC-a do dyspozycji programu jest górna połowa strony zerowej: `$80`–`$FF` (dokładniej o tym w rozdziale 19).

5.6 Indeksowany: absolutny,X i absolutny,Y

Do adresu podanego w rozkazie procesor **dodaje zawartość rejestru X** (**lub Y**) i dopiero pod tak obliczonym adresem szuka argumentu. To podstawowy sposób dostępu do **tablic**.

```
    lda tablica,x          gdy tablica = $3000, a X = 5:

    adres efektywny = $3000 + 5 = $3005   ->   A = zawartość komórki $3005
```

```
        ldx #3                  ; X = 3 - numer elementu (liczymy od zera)
        lda tablica,x           ; A = czwarty element tablicy (tablica+3)
        sta ekran,x             ; zapisz A do czwartej komórki obszaru "ekran"
        rts                     ; powrót z podprogramu
tablica dta 10,20,30,40,50      ; pięć elementów: indeksy 0..4
```

Rejestr indeksowy ma 8 bitów, więc w ten sposób sięgniemy najwyżej 255 bajtów od adresu bazowego. **Tablica obsługiwana jednym indeksem może mieć do 256 elementów.** Większe obszary wymagają adresowania pośredniego.

Dostępność: `lda`, `adc`, `cmp` i pozostałe rozkazy „akumulatorowe” działają zarówno z `,x`, jak i z `,y`. Rozkazy modyfikujące pamięć (`inc`, `dec`, `asl`, `lsr`, `rol`, `ror`) mają tylko wersję `,x`. Rejestr X można ładować z indeksem Y (`ldx tab,y`), a Y z indeksem X (`ldy tab,x`).

> **Uwaga**
>
> **Przekroczenie granicy strony.** Jeżeli po dodaniu indeksu zmienia się starszy bajt adresu (np. `$30F0` + `$20` = `$3110`), rozkaz odczytu trwa o 1 cykl dłużej. Dla poprawności programu nie ma to znaczenia; liczy się tylko w kodzie, w którym odmierzamy czas co do cyklu.

5.7 Indeksowany: strona zerowa,X i strona zerowa,Y

To samo co wyżej, ale dla adresów jednobajtowych. Jedna istotna różnica: wynik dodawania **nie wychodzi poza stronę zerową** — zawija się.

```
        ldx #$10                ; X = $10
        lda $80,x               ; A = zawartość komórki $90 ($80+$10)
        ldx #$90                ; X = $90
        lda $80,x               ; A = zawartość komórki $10, a NIE $0110! ($80+$90=$110 -> $10)
```

Tryb `strona zerowa,Y` istnieje tylko dla dwóch rozkazów: `ldx` i `stx`. Jeśli napiszesz `lda $80,y`, MADS użyje po cichu trybu `absolutny,Y` (3 bajty) — program zadziała poprawnie, tylko będzie o bajt dłuższy.

5.8 Pośredni indeksowany: (zp),y

To **najważniejszy tryb** dla każdego poważniejszego programu, bo pozwala sięgnąć pod adres, który nie jest znany w chwili asemblacji i który może się zmieniać w czasie pracy programu. Tak realizuje się **wskaźniki**.

Działanie krok po kroku dla `lda ($80),y`:

1. Procesor odczytuje z komórek `$80` i `$81` dwa bajty — młodszy i starszy bajt **adresu bazowego**.
2. Do tego adresu dodaje zawartość rejestru Y.
3. Spod otrzymanego adresu pobiera argument.

```
   lda ($80),y        gdy Y = 3

   strona zerowa:   $80: $00  ┐
                    $81: $9C  ┘──►  adres bazowy $9C00
                                         + Y (3)
                                    ──────────────
                                    adres efektywny $9C03  ──►  A = zawartość $9C03
```

```
wsk     = $80                   ; nasz wskaźnik zajmuje komórki $80 (LO) i $81 (HI)

        lda #<bufor             ; młodszy bajt adresu bufora
        sta wsk                 ; -> do młodszego bajtu wskaźnika
        lda #>bufor             ; starszy bajt adresu bufora
        sta wsk+1               ; -> do starszego bajtu wskaźnika
        ldy #0                  ; Y = 0: przesunięcie względem wskaźnika
        lda (wsk),y             ; A = bajt spod adresu, na który wskazuje "wsk"
        iny                     ; Y = 1
        lda (wsk),y             ; A = następny bajt
```

Trzy rzeczy do zapamiętania:

- Wskaźnik **musi** leżeć na stronie zerowej. Nie ma trybu `(adres_absolutny),y`.
- Rejestrem indeksowym jest tu **zawsze Y**. Jeśli nie potrzebujesz przesunięcia, i tak musisz ustawić `ldy #0`.
- Jednym wskaźnikiem obejmujesz 256 bajtów (Y od 0 do 255). Aby iść dalej, zwiększ starszy bajt wskaźnika (`inc wsk+1`) — to przesuwa cały wskaźnik o 256 bajtów, czyli o jedną stronę.

Na tym trybie opiera się kopiowanie dużych bloków pamięci (rozdział 18), obsługa łańcuchów znaków i pamięci ekranu.

5.9 Indeksowany pośredni: (zp,x)

Tu kolejność jest odwrotna: **najpierw** do adresu na stronie zerowej dodawany jest X, a **potem** spod otrzymanego miejsca odczytywany jest wskaźnik. Żadnego dodawania po odczytaniu wskaźnika już nie ma.

```
   lda ($80,x)        gdy X = 4

   $80 + 4 = $84:   $84: $34  ┐
                    $85: $12  ┘──►  adres efektywny $1234  ──►  A = zawartość $1234
```

Ten tryb pozwala wybierać jeden z kilku wskaźników ułożonych w tablicy na stronie zerowej (X musi wtedy przyjmować wartości parzyste: 0, 2, 4...). W praktyce używa się go rzadko. Najczęstsze zastosowanie to... zastępowanie trybu `(zp),y`, gdy rejestr Y jest zajęty:

```
        ldx #0                  ; X = 0
        lda (wsk,x)             ; z X = 0 działa jak "lda (wsk)" - odczyt spod wskaźnika
```

Dodawanie X odbywa się w obrębie strony zerowej (z zawijaniem, jak w trybie `zp,x`).

5.10 Pośredni — tylko `jmp`

Jedyny rozkaz z tym trybem to `jmp`. Skacze pod adres **odczytany z pamięci**:

```
        jmp ($000A)             ; skocz pod adres zapisany w komórkach $0A/$0B (wektor DOSVEC)
        jmp (wektor)            ; skocz tam, gdzie wskazuje zmienna "wektor"
```

To mechanizm **wektorów**: zmieniając dwa bajty w pamięci, zmieniamy cel skoku bez modyfikowania kodu.

Pułapka

Błąd procesora 6502 w `jmp (adres)`. Jeżeli młodszy bajt adresu wektora to `$FF` (np. `jmp ($30FF)`), procesor pobierze młodszy bajt celu z `$30FF`, ale starszy z `$3000` zamiast z `$3100`. Nigdy nie umieszczaj wektora na ostatnim bajcie strony. Gdy wektor jest zwykłą zmienną zadeklarowaną przez `dta a(0)`, wystarczy sprawdzić w listingu, czy nie wypadł pod adresem `$xxFF`.

5.11 Względny — rozgałęzienia

Osiem rozkazów skoków warunkowych (`beq`, `bne`, `bcc`, `bcs`, `bmi`, `bpl`, `bvc`, `bvs`) używa wyłącznie trybu względnego. Argumentem nie jest adres docelowy, lecz **jednobajtowe przesunięcie ze znakiem** dodawane do licznika rozkazów. Ty piszesz etykietę — asembler sam oblicza przesunięcie.

```
petla   dex                     ; X = X - 1
        bne petla               ; jeśli X <> 0, cofnij się do etykiety "petla"
```

Ponieważ przesunięcie ma 8 bitów ze znakiem, skok warunkowy może sięgnąć **najwyżej 127 bajtów w przód lub 128 bajtów wstecz** (licząc od rozkazu następnego po skoku). Gdy cel jest dalej, MADS zgłosi błąd *Branch out of range*. Rozwiązanie — „odwrócić warunek i przeskoczyć `jmp`" — poznasz w rozdziale 7.

Zaletą trybu względnego jest to, że taki kod jest **relokowalny**: przesunięcie nie zależy od miejsca, w którym kod leży w pamięci.

5.12 Jak wybrać tryb — ściąga

Chcę...	Tryb	Przykład
użyć stałej liczby	natychmiastowy	`lda #10`
odczytać / zapisać zmienną	absolutny lub strona zerowa	`lda licznik`
sięgnąć do tablicy do 256 bajtów	absolutny,X lub ,Y	`lda tab,x`
sięgnąć pod adres obliczony w programie	pośredni (zp),Y	`lda (wsk),y`
przejść po dużym obszarze pamięci	pośredni (zp),Y + `inc wsk+1`	rozdział 18
skoczyć pod adres zapisany w pamięci	pośredni	`jmp (wektor)`
skoczyć warunkowo	względny	`bne petla`

Ćwiczenia

1. Komórka `$80` zawiera `$00`, `$81` zawiera `$06`, rejestr Y = `$10`. Spod jakiego adresu pobierze dane `lda ($80),y`?
2. Rejestr X = `$FF`. Którą komórkę odczyta `lda $02,x`, a którą `lda $0200,x`?
3. Dlaczego nie istnieje rozkaz `sta #$10`?
4. Ile bajtów zajmą rozkazy: `lda #0`, `lda $80`, `lda $0600`, `lda ($80),y`, `inx`?

Odpowiedzi: (1) `$0610`. (2) `$01` (zawinięcie w stronie zerowej) oraz `$02FF`. (3) Bo stałej zapisanej w rozkazie nie można zmienić — zapis wymaga adresu. (4) 2, 2, 3, 2, 1.

ROZDZIAŁ 6

Leksykon rozkazów procesora 6502

Procesor 6502 ma **56 rozkazów** (mnemoników), które w połączeniu z trybami adresowania dają 151 kodów maszynowych. Ten rozdział opisuje każdy z nich: co robi, jak wpływa na znaczniki, w jakich trybach działa, ile zajmuje bajtów i ile trwa — oraz pokazuje go w działaniu.

Nie ucz się tego rozdziału na pamięć. Przeczytaj go raz uważnie, a potem wracaj do niego jak do słownika.

6.1 Jak czytać opisy

Przy każdym rozkazie znajdują się dwie tabele. Pierwsza pokazuje wpływ na **znaczniki**:

Symbol	Znaczenie
✓	znacznik zmienia się zależnie od wyniku rozkazu
–	znacznik pozostaje bez zmian
0, 1	znacznik jest zawsze kasowany lub zawsze ustawiany
7, 6	do znacznika trafia bit 7 lub bit 6 argumentu (tylko rozkaz `bit`)

Druga tabela wymienia **tryby adresowania**, zapis w MADS, **kod maszynowy** rozkazu (pierwszy bajt, szesnastkowo), **liczbę bajtów** całego rozkazu oraz **liczbę cykli** zegara, które zajmuje wykonanie. Przy cyklach mogą stać gwiazdki:

- * — dolicz 1 cykl, jeżeli indeksowanie przekracza granicę strony,
- ** — dolicz 1 cykl, jeżeli skok warunkowy jest wykonywany, i jeszcze 1, jeżeli cel leży na innej stronie niż rozkaz następny po skoku.

W opisach działania litera **M** oznacza argument (liczbę lub zawartość komórki pamięci), a strzałka ← oznacza przypisanie.

6.2 Ładowanie i zapisywanie rejestrów

6.2.1 LDA — załaduj akumulator (*LoaD Accumulator*)

Działanie: A ← M. Kopiuje argument do akumulatora. Ustawia Z, gdy załadowano zero, oraz N według bitu 7 załadowanej wartości.

N	V	B	D	I	Z	C
✓	–	–	–	–	✓	–

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
natychmiastowy	`lda #$44`	`$A9`	2	2
strona zerowa	`lda $44`	`$A5`	2	3
strona zerowa,X	`lda $44,x`	`$B5`	2	4
absolutny	`lda $4400`	`$AD`	3	4

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
absolutny,X	`lda $4400,x`	`$BD`	3	4*
absolutny,Y	`lda $4400,y`	`$B9`	3	4*
pośredni indeksowany (zp,X)	`lda ($44,x)`	`$A1`	2	6
pośredni indeksowany (zp),Y	`lda ($44),y`	`$B1`	2	5*

`lda` to najczęściej używany rozkaz 6502. Każda operacja na danych zaczyna się od pobrania ich do akumulatora.

```
        lda #$7F                ; A = $7F (tryb natychmiastowy)     Z=0, N=0
        lda #0                  ; A = 0                             Z=1, N=0
        lda #$80                ; A = $80                           Z=0, N=1 (bit 7 ustawiony)
        lda $14                 ; A = zawartość komórki $14 (zegar systemowy RTCLOK+2)
        lda tablica,x           ; A = element tablicy o numerze X
        lda (wsk),y             ; A = bajt spod adresu (wsk) + Y
        beq zero                ; skok warunkowy może korzystać ze znaczników od razu po LDA
```

6.2.2 LDX — załaduj rejestr X (*LoaD X*)

Działanie: X ← M.

N	V	B	D	I	Z	C
✓	–	–	–	–	✓	–

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
natychmiastowy	`ldx #$44`	`$A2`	2	2
strona zerowa	`ldx $44`	`$A6`	2	3
strona zerowa,Y	`ldx $44,y`	`$B6`	2	4
absolutny	`ldx $4400`	`$AE`	3	4
absolutny,Y	`ldx $4400,y`	`$BE`	3	4*

Zwróć uwagę, że `ldx` indeksuje się rejestrem **Y** (nie można indeksować rejestru samym sobą).

```
        ldx #0                  ; X = 0 - typowe zerowanie indeksu przed pętlą
        ldx licznik             ; X = zawartość zmiennej "licznik"
        ldx tablica,y           ; X = element tablicy o numerze Y
```

6.2.3 LDY — załaduj rejestr Y (*LoaD Y*)

Działanie: Y ← M.

N	V	B	D	I	Z	C
✓	–	–	–	–	✓	–

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
natychmiastowy	`ldy #$44`	`$A0`	2	2
strona zerowa	`ldy $44`	`$A4`	2	3
strona zerowa,X	`ldy $44,x`	`$B4`	2	4
absolutny	`ldy $4400`	`$AC`	3	4
absolutny,X	`ldy $4400,x`	`$BC`	3	4*

```
        ldy #0                  ; Y = 0 - potrzebne przed "lda (wsk),y", gdy nie chcemy przesunięcia
        ldy dlugosc             ; Y = zawartość zmiennej "dlugosc"
        ldy tablica,x           ; Y = element tablicy o numerze X
```

6.2.4 STA — zapisz akumulator (*STore Accumulator*)

Działanie: M ← A. Kopiuje akumulator do komórki pamięci. Akumulator się nie zmienia. **Nie zmienia żadnych znaczników.**

N	V	B	D	I	Z	C
–	–	–	–	–	–	–

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
strona zerowa	`sta $44`	`$85`	2	3
strona zerowa,X	`sta $44,x`	`$95`	2	4
absolutny	`sta $4400`	`$8D`	3	4
absolutny,X	`sta $4400,x`	`$9D`	3	5
absolutny,Y	`sta $4400,y`	`$99`	3	5
pośredni indeksowany (zp,X)	`sta ($44,x)`	`$81`	2	6
pośredni indeksowany (zp),Y	`sta ($44),y`	`$91`	2	6

Nie ma trybu natychmiastowego (nie można „zapisać do liczby”). Zapisy indeksowane mają stały czas wykonania — nie zależą od przekroczenia strony.

```
        lda #$0E                ; A = $0E (jasny kolor)
        sta $02C5               ; zapisz do COLOR1 - kolor liter
        sta $02C8               ; ta sama wartość do COLOR4 - A się nie zmienił
        sta bufor,x             ; zapisz do elementu X tablicy "bufor"
        sta (wsk),y             ; zapisz pod adres (wsk) + Y
```

6.2.5 STX — zapisz rejestr X (*STore X*)

Działanie: M ← X. Nie zmienia znaczników.

N	V	B	D	I	Z	C
–	–	–	–	–	–	–

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
strona zerowa	`stx $44`	`$86`	2	3
strona zerowa,Y	`stx $44,y`	`$96`	2	4
absolutny	`stx $4400`	`$8E`	3	4

```
        stx zapas_x             ; zapamiętaj X w zmiennej, żeby móc go potem odtworzyć
        stx $80,y               ; zapis indeksowany - tylko na stronie zerowej i tylko z Y
```

Brak trybu `stx adres,y` dla pełnych adresów bywa dokuczliwy; wtedy trzeba przejść przez akumulator: `txa` i `sta adres,y`.

6.2.6 STY — zapisz rejestr Y (*STore Y*)

Działanie: M ← Y. Nie zmienia znaczników.

N	V	B	D	I	Z	C
–	–	–	–	–	–	–

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
strona zerowa	`sty $44`	`$84`	2	3
strona zerowa,X	`sty $44,x`	`$94`	2	4
absolutny	`sty $4400`	`$8C`	3	4

```
        sty zapas_y             ; zapamiętaj Y
        sty $80,x               ; zapis indeksowany - tylko na stronie zerowej i tylko z X
```

6.3 Przesyłanie między rejestrami

Sześć rozkazów jednobajtowych, każdy trwa 2 cykle. Kopiują zawartość jednego rejestru do drugiego; rejestr źródłowy pozostaje bez zmian. Nie ma bezpośredniego przesłania między X i Y — trzeba iść przez akumulator (`txa`, `tay`).

6.3.1 TAX — prześlij A do X (*Transfer A to X*)

Działanie: X ← A.

N	V	B	D	I	Z	C
✓	–	–	–	–	✓	–

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
implikowany	`tax`	`$AA`	1	2

```
        lda numer               ; A = numer elementu
        asl @                   ; A = A * 2 (bo elementy tablicy są dwubajtowe)
        tax                     ; X = A - indeks trzeba mieć w rejestrze indeksowym
        lda tablica,x           ; pobierz młodszy bajt elementu
```

6.3.2 TAY — prześlij A do Y (*Transfer A to Y*)

Działanie: Y ← A.

N	V	B	D	I	Z	C
✓	–	–	–	–	✓	–

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
implikowany	`tay`	`$A8`	1	2

```
        lda dlugosc             ; A = długość napisu
        tay                     ; Y = A - teraz Y może służyć jako licznik lub indeks
```

6.3.3 TXA — prześlij X do A (*Transfer X to A*)

Działanie: A ← X.

N	V	B	D	I	Z	C
✓	–	–	–	–	✓	–

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
implikowany	`txa`	`$8A`	1	2

```
        txa                     ; A = X - np. żeby wykonać na liczniku arytmetykę
        clc                     ; (dodawać można tylko w akumulatorze)
        adc #10                 ; A = X + 10
```

6.3.4 TYA — prześlij Y do A (*Transfer Y to A*)

Działanie: A ← Y.

N	V	B	D	I	Z	C
✓	–	–	–	–	✓	–

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
implikowany	`tya`	`$98`	1	2

```
        tya                     ; A = Y
        pha                     ; odłóż Y na stos (rozkazu "odłóż Y" nie ma, trzeba przez A)
```

6.3.5 TSX — prześlij wskaźnik stosu do X (*Transfer S to X*)

Działanie: X ← S.

N	V	B	D	I	Z	C
✓	–	–	–	–	✓	–

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
implikowany	`tsx`	`$BA`	1	2

Jedyny sposób odczytania wskaźnika stosu. Po `tsx` adres `$0101,x` wskazuje ostatni odłożony bajt, co pozwala zaglądać w głąb stosu bez zdejmowania danych.

```
        tsx                     ; X = wskaźnik stosu
        stx zapas_s             ; zapamiętaj go, aby móc awaryjnie wrócić do tego stanu stosu
        lda $0101,x             ; A = bajt ze szczytu stosu (bez zdejmowania go)
```

6.3.6 TXS — prześlij X do wskaźnika stosu (*Transfer X to S*)

Działanie: S ← X. **Nie zmienia znaczników** (jako jedyny z rozkazów przesłań).

N	V	B	D	I	Z	C
–	–	–	–	–	–	–

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
implikowany	`txs`	`$9A`	1	2

```
        ldx zapas_s             ; X = zapamiętana wcześniej wartość wskaźnika stosu
        txs                     ; S = X - stos wraca do zapamiętanego stanu;
                                ; wszystko, co odłożono później, zostaje "zapomniane"
```

Uwaga

`txs` to narzędzie do zadań specjalnych: inicjalizacji stosu w programie startującym bez systemu operacyjnego albo awaryjnego wyjścia z głęboko zagnieżdżonych podprogramów. W zwykłym programie uruchamianym z DOS-u nie zmieniaj wskaźnika stosu bez wyraźnej potrzeby.

6.4 Rozkazy stosu

6.4.1 PHA — odłóż akumulator na stos (*PusH Accumulator*)

Działanie: komórka `$0100+S` ← A, następnie S ← S − 1.

N	V	B	D	I	Z	C
–	–	–	–	–	–	–

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
implikowany	`pha`	`$48`	1	3

6.4.2 PLA — zdejmij akumulator ze stosu (*PuLl Accumulator*)

Działanie: S ← S + 1, następnie A ← komórka `$0100+S`. Ustawia N i Z według zdjętej wartości.

N	V	B	D	I	Z	C
✓	–	–	–	–	✓	–

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
implikowany	`pla`	`$68`	1	4

Para `pha`/`pla` to najprostszy sposób chwilowego przechowania wartości:

```
        pha                     ; zapamiętaj A na stosie
        jsr inna_proc           ; wywołaj procedurę, która niszczy A
        pla                     ; odtwórz A
```

Rejestry X i Y nie mają własnych rozkazów stosu, więc odkłada się je przez akumulator. Oto pełne zachowanie wszystkich trzech rejestrów — zwróć uwagę na **odwrotną kolejność** przy zdejmowaniu:

```
        pha                     ; odłóż A
        txa                     ; A = X
        pha                     ; odłóż X
        tya                     ; A = Y
        pha                     ; odłóż Y
                                ; ... tu kod, który może dowolnie zmieniać A, X, Y ...
        pla                     ; zdejmij ostatnio odłożoną wartość (to był Y)
        tay                     ; Y = odtworzony
        pla                     ; zdejmij kolejną (to był X)
        tax                     ; X = odtworzony
        pla                     ; zdejmij pierwszą odłożoną (to był A)
```

Pułapka

Każdemu `pha` musi odpowiadać dokładnie jedno `pla` na każdej możliwej ścieżce wykonania. Jeżeli podprogram zostawi na stosie dodatkowy bajt, rozkaz `rts` potraktuje go jako część adresu powrotu i procesor skoczy w przypadkowe miejsce.

6.4.3 PHP — odłóż rejestr znaczników na stos (*PusH Processor status*)

Działanie: komórka `$0100+S` ← P (z ustawionymi bitami 4 i 5), następnie S ← S − 1.

N	V	B	D	I	Z	C
–	–	–	–	–	–	–

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
implikowany	`php`	`$08`	1	3

6.4.4 PLP — zdejmij rejestr znaczników ze stosu (*PuLl Processor status*)

Działanie: S ← S + 1, następnie P ← komórka `$0100+S`. Odtwarza **wszystkie** znaczniki naraz.

N	V	B	D	I	Z	C
✓	✓	–	✓	✓	✓	✓

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
implikowany	`plp`	`$28`	1	4

Typowe zastosowanie: zapamiętanie stanu przerwań (znacznik I), ich zablokowanie na chwilę i przywrócenie *poprzedniego* stanu — niezależnie od tego, jaki był:

```
        php                     ; zapamiętaj wszystkie znaczniki, w tym I (stan blokady przerwań)
        sei                     ; zablokuj przerwania IRQ
        lda #<nowy_wektor       ; ... operacja, której nie wolno przerwać w połowie:
        sta $0208               ; zmiana dwóch bajtów wektora przerwania
        lda #>nowy_wektor       ; A = starszy bajt adresu nowy_wektor
        sta $0209               ; zapisz A do komórki $0209
        plp                     ; przywróć znaczniki - przerwania wracają do stanu sprzed "php"
```

Drugie zastosowanie to przeniesienie wyniku porównania przez fragment kodu, który zmienia znaczniki:

```
        cmp #10                 ; porównanie ustawia znaczniki
        php                     ; zapamiętaj wynik porównania
        lda #0                  ; ten rozkaz zmienia Z i N...
        sta licznik             ; zapisz A do licznik
        plp                     ; ...ale tu wynik porównania wraca
        bcc mniejsze            ; i można na nim oprzeć decyzję
```

6.5 Arytmetyka

6.5.1 ADC — dodaj z przeniesieniem (*ADd with Carry*)

Działanie: A ← A + M + C. Dodaje do akumulatora argument **oraz znacznik przeniesienia** (0 lub 1).

N	V	B	D	I	Z	C
✓	✓	–	–	–	✓	✓

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
natychmiastowy	`adc #$44`	`$69`	2	2
strona zerowa	`adc $44`	`$65`	2	3
strona zerowa,X	`adc $44,x`	`$75`	2	4
absolutny	`adc $4400`	`$6D`	3	4
absolutny,X	`adc $4400,x`	`$7D`	3	4*
absolutny,Y	`adc $4400,y`	`$79`	3	4*
pośredni indeksowany (zp,X)	`adc ($44,x)`	`$61`	2	6
pośredni indeksowany (zp),Y	`adc ($44),y`	`$71`	2	5*

Znaczniki po dodawaniu:

- **C** = 1, gdy wynik przekroczył 255 (przeniesienie z bitu 7),
- **V** = 1, gdy wystąpiło przepełnienie liczb ze znakiem (dodano dwie liczby o tym samym znaku, a wynik ma znak przeciwny),
- **Z** = 1, gdy w akumulatorze zostało zero, **N** = bit 7 wyniku.

> **Zapamiętaj**
>
> Procesor 6502 **nie ma rozkazu dodawania bez przeniesienia**. `adc` zawsze dolicza znacznik C. Dlatego przed zwykłym dodawaniem trzeba ten znacznik skasować rozkazem `clc`. Zapomniane `clc` daje wynik „czasem większy o 1" — jeden z najbardziej podstępnych błędów.

```
        lda liczba1             ; A = pierwszy składnik
        clc                     ; C = 0 - żeby do sumy nie doliczyło się przypadkowe przeniesienie
        adc liczba2             ; A = liczba1 + liczba2 + 0
        sta suma                ; zapisz wynik; C mówi teraz, czy suma przekroczyła 255
```

To, że `adc` dolicza C, jest zaletą: pozwala łączyć dodawania w łańcuch i sumować liczby dowolnej długości (rozdział 12).

6.5.2 SBC — odejmij z pożyczką (*SuBtract with Carry*)

Działanie: A ← A − M − (1 − C). Odejmuje od akumulatora argument oraz **negację** znacznika C.

N	V	B	D	I	Z	C
✓	✓	–	–	–	✓	✓

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
natychmiastowy	`sbc #$44`	`$E9`	2	2
strona zerowa	`sbc $44`	`$E5`	2	3
strona zerowa,X	`sbc $44,x`	`$F5`	2	4

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
absolutny	`sbc $4400`	`$ED`	3	4
absolutny,X	`sbc $4400,x`	`$FD`	3	4*
absolutny,Y	`sbc $4400,y`	`$F9`	3	4*
pośredni indeksowany (zp,X)	`sbc ($44,x)`	`$E1`	2	6
pośredni indeksowany (zp),Y	`sbc ($44),y`	`$F1`	2	5*

W 6502 znacznik C przy odejmowaniu działa **odwrotnie** niż intuicyjna „pożyczka”:

- przed odejmowaniem **C musi być ustawiony** (`sec`), aby wynik był dokładny,
- po odejmowaniu **C = 1 oznacza brak pożyczki** (wynik nieujemny, A było ≥ M),
- po odejmowaniu **C = 0 oznacza pożyczkę** (wynik „przeszedł przez zero”, A było < M).

```
        lda liczba1             ; A = odjemna
        sec                     ; C = 1 - "nie ma pożyczki"; obowiązkowe przed zwykłym odejmowaniem
        sbc liczba2             ; A = liczba1 - liczba2
        sta roznica             ; zapisz wynik
        bcc ujemny              ; C = 0: liczba1 była mniejsza od liczba2 (wynik "zawinął się")
```

Zapamiętaj

Przed dodawaniem `clc`, przed odejmowaniem `sec`. Powtarzaj to, aż wejdzie w krew.

6.6 Zwiększanie i zmniejszanie o jeden

Sześć rozkazów zmieniających wartość o 1. Wszystkie ustawiają znaczniki N i Z, ale — uwaga — **żaden nie zmienia znacznika C**. Po `inc` nie da się więc sprawdzić przeniesienia; sprawdza się znacznik Z (wynik „przekręcił się” na zero).

6.6.1 INC — zwiększ komórkę pamięci (*INCrement memory*)

Działanie: M ← M + 1.

N	V	B	D	I	Z	C
✓	–	–	–	–	✓	–

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
strona zerowa	`inc $44`	`$E6`	2	5
strona zerowa,X	`inc $44,x`	`$F6`	2	6
absolutny	`inc $4400`	`$EE`	3	6
absolutny,X	`inc $4400,x`	`$FE`	3	7

```
        inc licznik             ; licznik = licznik + 1
        bne dalej               ; jeżeli wynik <> 0, nie było "przekręcenia" przez 255
        inc licznik+1           ; było: zwiększ także starszy bajt (licznik 16-bitowy)
dalej
```

Nie istnieje `inc` dla akumulatora. Zwiększenie A o 1 zapisuje się `clc` i `adc #1`.

6.6.2 INX — zwiększ rejestr X (*INcrement X*)

Działanie: X ← X + 1. Po `$FF` następuje `$00`.

N	V	B	D	I	Z	C
✓	–	–	–	–	✓	–

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
implikowany	`inx`	`$E8`	1	2

6.6.3 INY — zwiększ rejestr Y (*INcrement Y*)

Działanie: Y ← Y + 1.

N	V	B	D	I	Z	C
✓	–	–	–	–	✓	–

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
implikowany	`iny`	`$C8`	1	2

```
        ldy #0                  ; Y = 0 - indeks pierwszego znaku
nast    lda napis,y             ; pobierz znak numer Y
        beq koniec              ; bajt zerowy = koniec napisu
        sta ekran,y             ; postaw znak na ekranie
        iny                     ; przejdź do następnego znaku
        bne nast                ; powtarzaj (pętla skończy się najpóźniej po 256 znakach)
koniec
```

6.6.4 DEC — zmniejsz komórkę pamięci (*DECrement memory*)

Działanie: M ← M − 1.

N	V	B	D	I	Z	C
✓	–	–	–	–	✓	–

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
strona zerowa	`dec $44`	`$C6`	2	5
strona zerowa,X	`dec $44,x`	`$D6`	2	6
absolutny	`dec $4400`	`$CE`	3	6
absolutny,X	`dec $4400,x`	`$DE`	3	7

```
        dec zycia               ; jedno życie mniej
        beq koniec_gry          ; jeżeli licznik doszedł do zera - koniec gry
```

6.6.5 DEX — zmniejsz rejestr X (*DEcrement X*)

Działanie: X ← X − 1. Po `$00` następuje `$FF`.

N	V	B	D	I	Z	C
✓	–	–	–	–	✓	–

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
implikowany	`dex`	`$CA`	1	2

6.6.6 DEY — zmniejsz rejestr Y (*DEcrement Y*)

Działanie: Y ← Y − 1.

N	V	B	D	I	Z	C
✓	–	–	–	–	✓	–

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
implikowany	`dey`	`$88`	1	2

`dex` i `dey` to naturalne liczniki pętli, bo zejście do zera od razu ustawia znacznik Z — nie potrzeba osobnego porównania:

```
        ldx #8                  ; X = 8 - pętla wykona się 8 razy
petla   jsr cos_zrob            ; ciało pętli
        dex                     ; X = X - 1; gdy X osiągnie 0, ustawi się Z
        bne petla               ; dopóki X <> 0, powtarzaj
```

6.7 Operacje logiczne

6.7.1 AND — iloczyn logiczny (*logical AND*)

Działanie: A ← A AND M. Bit wyniku jest jedynką tylko wtedy, gdy jest jedynką **w obu** argumentach.

N	V	B	D	I	Z	C
✓	–	–	–	–	✓	–

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
natychmiastowy	`and #$44`	`$29`	2	2
strona zerowa	`and $44`	`$25`	2	3
strona zerowa,X	`and $44,x`	`$35`	2	4
absolutny	`and $4400`	`$2D`	3	4
absolutny,X	`and $4400,x`	`$3D`	3	4*
absolutny,Y	`and $4400,y`	`$39`	3	4*
pośredni indeksowany (zp,X)	`and ($44,x)`	`$21`	2	6
pośredni indeksowany (zp),Y	`and ($44),y`	`$31`	2	5*

Zastosowanie: **kasowanie wybranych bitów** i **wydzielanie** części bajtu.

```
        lda bajt                ; A = %abcdefgh
        and #%00001111          ; A = %0000efgh - zostaw tylko młodszy półbajt
        sta cyfra               ; (bity 7-4 zostały wyzerowane)
```

```
        lda $D20A               ; A = liczba losowa 0..255 z układu POKEY
        and #%00000111          ; A = liczba losowa 0..7 (zostały tylko trzy najmłodsze bity)

        lda znak                ; A = kod znaku ATASCII
        and #%01111111          ; skasuj bit 7 - usuń cechę "znak w negatywie"
```

6.7.2 ORA — suma logiczna (*logical OR with Accumulator*)

Działanie: A ← A OR M. Bit wyniku jest jedynką, gdy jest jedynką **w którymkolwiek** z argumentów.

N	V	B	D	I	Z	C
✓	–	–	–	–	✓	–

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
natychmiastowy	`ora #$44`	`$09`	2	2
strona zerowa	`ora $44`	`$05`	2	3
strona zerowa,X	`ora $44,x`	`$15`	2	4
absolutny	`ora $4400`	`$0D`	3	4
absolutny,X	`ora $4400,x`	`$1D`	3	4*
absolutny,Y	`ora $4400,y`	`$19`	3	4*
pośredni indeksowany (zp,X)	`ora ($44,x)`	`$01`	2	6
pośredni indeksowany (zp),Y	`ora ($44),y`	`$11`	2	5*

Zastosowanie: **ustawianie wybranych bitów** i **składanie** bajtu z części.

```
        lda znak                ; A = kod znaku
        ora #%10000000          ; ustaw bit 7 - znak zostanie wyświetlony w negatywie

        lda kolor               ; A = numer koloru 0..15
        asl @                   ; przesuń go o 4 bity w lewo,
        asl @                   ; czyli do starszego półbajtu
        asl @                   ; przesuń A o 1 bit w lewo (bit 7 -> C)
        asl @                   ; przesuń A o 1 bit w lewo (bit 7 -> C)
        ora jasnosc             ; dołącz jasność 0..15 w młodszym półbajcie
        sta $02C6               ; zapisz gotowy bajt koloru tła

        lda liczba              ; test "czy liczba 16-bitowa jest zerem":
        ora liczba+1            ; suma logiczna obu bajtów daje zero tylko wtedy, gdy oba są zerami
        beq jest_zero           ; skocz do jest_zero, gdy Z = 1 (równe / zero)
```

6.7.3 EOR — różnica symetryczna (*Exclusive OR*)

Działanie: A ← A XOR M. Bit wyniku jest jedynką, gdy bity argumentów **różnią się**.

N	V	B	D	I	Z	C
✓	–	–	–	–	✓	–

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
natychmiastowy	`eor #$44`	`$49`	2	2
strona zerowa	`eor $44`	`$45`	2	3
strona zerowa,X	`eor $44,x`	`$55`	2	4
absolutny	`eor $4400`	`$4D`	3	4
absolutny,X	`eor $4400,x`	`$5D`	3	4*

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
absolutny,Y	`eor $4400,y`	`$59`	3	4*
pośredni indeksowany (zp,X)	`eor ($44,x)`	`$41`	2	6
pośredni indeksowany (zp),Y	`eor ($44),y`	`$51`	2	5*

Zastosowanie: **odwracanie wybranych bitów**.

```
        lda bajt                ; A = wartość
        eor #$FF                ; odwróć wszystkie bity (negacja bitowa, "NOT")

        lda stan                ; A = zmienna-przełącznik (0 albo 1)
        eor #1                  ; odwróć bit 0: z 0 robi się 1, z 1 robi się 0
        sta stan                ; zapisz - przełącznik zmienił położenie

        lda znak                ; A = kod znaku
        eor #$80                ; odwróć bit 7: znak normalny <-> negatyw (efekt migania)
```

`eor` ma ciekawą własność: dwukrotne wykonanie z tą samą maską przywraca wartość pierwotną. Na tym opiera się proste szyfrowanie danych i rysowanie „znikających” kursorów.

6.7.4 BIT — testuj bity (*BIT test*)

Działanie: oblicza A AND M, ale **wyniku nigdzie nie zapisuje** — ustawia tylko znaczniki. Dodatkowo kopiuje bit 7 argumentu do znacznika N, a bit 6 do znacznika V.

N	V	B	D	I	Z	C
7	6	–	–	–	✓	–

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
strona zerowa	`bit $44`	`$24`	2	3
absolutny	`bit $4400`	`$2C`	3	4

- **Z** = 1, gdy A AND M = 0 (żaden z bitów wskazanych przez maskę w A nie jest ustawiony w M),
- **N** = bit 7 komórki M, **V** = bit 6 komórki M — niezależnie od zawartości akumulatora.

`bit` pozwala zbadać komórkę pamięci **bez niszczenia akumulatora**:

```
        bit $D20F               ; zbadaj rejestr SKSTAT; A się nie zmienia
        bmi brak                ; N = bit 7 komórki; skocz, jeśli jest ustawiony
        bvs inne                ; V = bit 6 komórki; skocz, jeśli jest ustawiony

        lda #%00000100          ; maska: interesuje nas bit 2
        bit $D01F               ; czy w rejestrze CONSOL bit 2 jest ustawiony?
        beq opcja               ; Z = 1: bit 2 jest zerem - klawisz OPTION jest wciśnięty
```

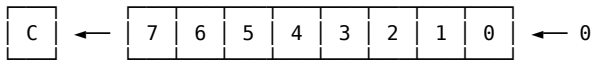
Rozkaz `bit` nie ma trybu natychmiastowego. Bywa też używany jako sztuczka do „połknięcia” następnego rozkazu: bajt `$2C` (kod `bit` absolutnego) sprawia, że dwa kolejne bajty zostaną potraktowane jako adres, a nie jako rozkaz.

6.8 Przesunięcia i obroty

Cztery rozkazy, które przesuwają wszystkie bity o jedną pozycję. Działają na akumulatorze (`asl @`) albo bezpośrednio na komórce pamięci (`asl adres`). Bit „wypadający” trafia zawsze do znacznika C.

6.8.1 ASL — przesunięcie arytmetyczne w lewo (*Arithmetic Shift Left*)

Działanie: C ← bit 7; wszystkie bity o jedno miejsce w lewo; bit 0 ← 0.



N	V	B	D	I	Z	C
✓	–	–	–	–	✓	✓

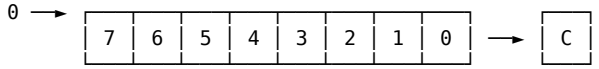
Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
akumulator	`asl @`	`$0A`	1	2
strona zerowa	`asl $44`	`$06`	2	5
strona zerowa,X	`asl $44,x`	`$16`	2	6
absolutny	`asl $4400`	`$0E`	3	6
absolutny,X	`asl $4400,x`	`$1E`	3	7

Efekt arytmetyczny: **mnożenie przez 2**. Znacznik C sygnalizuje przepełnienie.

```
        lda #%00010101          ; A = 21
        asl @                   ; A = %00101010 = 42, C = 0
        asl @                   ; A = %01010100 = 84, C = 0
        asl @                   ; A = %10101000 = 168, C = 0
        asl @                   ; A = %01010000 = 80, C = 1 (wynik 336 nie zmieścił się w bajcie)
```

6.8.2 LSR — przesunięcie logiczne w prawo (*Logical Shift Right*)

Działanie: C ← bit 0; wszystkie bity o jedno miejsce w prawo; bit 7 ← 0.



N	V	B	D	I	Z	C
0	–	–	–	–	✓	✓

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
akumulator	`lsr @`	`$4A`	1	2
strona zerowa	`lsr $44`	`$46`	2	5
strona zerowa,X	`lsr $44,x`	`$56`	2	6
absolutny	`lsr $4400`	`$4E`	3	6
absolutny,X	`lsr $4400,x`	`$5E`	3	7

Efekt arytmetyczny: **dzielenie liczby bez znaku przez 2**; w znaczniku C zostaje reszta. Znacznik N jest po `lsr` zawsze zerem.

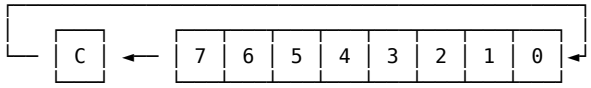
```
        lda #%00010101          ; A = 21
        lsr @                   ; A = %00001010 = 10, C = 1 (reszta z dzielenia 21 przez 2)

        lda bajt                ; A = %abcdefgh
        lsr @                   ; cztery przesunięcia w prawo
        lsr @                   ; przenoszą starszy półbajt
        lsr @                   ; na miejsce młodszego:
        lsr @                   ; A = %0000abcd
```

```
        lda liczba              ; test parzystości:
        lsr @                   ; bit 0 trafia do C
        bcs nieparzysta         ; C = 1 - liczba była nieparzysta
```

6.8.3 ROL — obrót w lewo przez przeniesienie (*ROtate Left*)

Działanie: wszystkie bity w lewo; do bitu 0 wchodzi **stara** wartość C; bit 7 przechodzi do C. Obrót obejmuje więc 9 bitów.



N	V	B	D	I	Z	C
✓	–	–	–	–	✓	✓

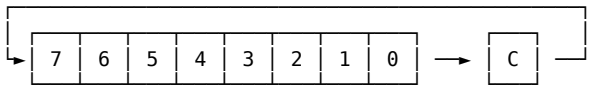
Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
akumulator	`rol @`	`$2A`	1	2
strona zerowa	`rol $44`	`$26`	2	5
strona zerowa,X	`rol $44,x`	`$36`	2	6
absolutny	`rol $4400`	`$2E`	3	6
absolutny,X	`rol $4400,x`	`$3E`	3	7

`rol` jest partnerem `asl` w przesuwaniu liczb wielobajtowych: bit wysunięty z młodszego bajtu wchodzi do starszego.

```
        asl liczba              ; przesuń młodszy bajt w lewo; jego bit 7 trafia do C
        rol liczba+1            ; przesuń starszy bajt w lewo, wsuwając C do bitu 0
                                ; razem: 16-bitowa liczba została pomnożona przez 2
```

6.8.4 ROR — obrót w prawo przez przeniesienie (*ROtate Right*)

Działanie: wszystkie bity w prawo; do bitu 7 wchodzi **stara** wartość C; bit 0 przechodzi do C.



N	V	B	D	I	Z	C
✓	–	–	–	–	✓	✓

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
akumulator	`ror @`	`$6A`	1	2
strona zerowa	`ror $44`	`$66`	2	5
strona zerowa,X	`ror $44,x`	`$76`	2	6
absolutny	`ror $4400`	`$6E`	3	6
absolutny,X	`ror $4400,x`	`$7E`	3	7

```
        lsr liczba+1            ; przesuń STARSZY bajt w prawo; jego bit 0 trafia do C
        ror liczba              ; przesuń młodszy bajt w prawo, wsuwając C do bitu 7
                                ; razem: 16-bitowa liczba została podzielona przez 2
```

Zauważ kolejność: przy przesuwaniu w lewo zaczynamy od młodszego bajtu, przy przesuwaniu w prawo — od starszego.

6.9 Porównania

Rozkazy porównania wykonują **odejmowanie na próbę**: obliczają rejestr − M, ustawiają znaczniki i **wyrzucają wynik**. Rejestr się nie zmienia. W odróżnieniu od `sbc` nie wymagają wcześniejszego `sec` i nie zmieniają znacznika V.

6.9.1 CMP — porównaj z akumulatorem (*CoMPare*)

Działanie: oblicza A − M i ustawia znaczniki N, Z, C.

N	V	B	D	I	Z	C
✓	–	–	–	–	✓	✓

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
natychmiastowy	`cmp #$44`	`$C9`	2	2
strona zerowa	`cmp $44`	`$C5`	2	3
strona zerowa,X	`cmp $44,x`	`$D5`	2	4
absolutny	`cmp $4400`	`$CD`	3	4
absolutny,X	`cmp $4400,x`	`$DD`	3	4*
absolutny,Y	`cmp $4400,y`	`$D9`	3	4*
pośredni indeksowany (zp,X)	`cmp ($44,x)`	`$C1`	2	6
pośredni indeksowany (zp),Y	`cmp ($44),y`	`$D1`	2	5*

Zależność (liczby bez znaku)	Z	C
A = M	1	1
A > M	0	1
A < M	0	0

Stąd reguła, na której opierają się wszystkie instrukcje warunkowe:

> **Zapamiętaj**
>
> Po `cmp`: **`beq`** skacze, gdy A = M. **`bne`** skacze, gdy A ≠ M. **`bcs`** skacze, gdy A ≥ M. **`bcc`** skacze, gdy A < M. (Dla liczb bez znaku.)

```
        lda wiek                ; A = wartość do zbadania
        cmp #18                 ; porównaj z 18 (oblicz A - 18 i ustaw znaczniki)
        bcc niepelnoletni       ; C = 0: A < 18
        beq rowno_18            ; Z = 1: A = 18
                                ; tutaj: A > 18
```

6.9.2 CPX — porównaj z rejestrem X (*ComPare X*)

Działanie: oblicza X − M i ustawia znaczniki N, Z, C.

N	V	B	D	I	Z	C
✓	–	–	–	–	✓	✓

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
natychmiastowy	`cpx #$44`	`$E0`	2	2
strona zerowa	`cpx $44`	`$E4`	2	3
absolutny	`cpx $4400`	`$EC`	3	4

6.9.3 CPY — porównaj z rejestrem Y (*ComPare Y*)

Działanie: oblicza Y − M i ustawia znaczniki N, Z, C.

N	V	B	D	I	Z	C
✓	–	–	–	–	✓	✓

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
natychmiastowy	`cpy #$44`	`$C0`	2	2
strona zerowa	`cpy $44`	`$C4`	2	3
absolutny	`cpy $4400`	`$CC`	3	4

`cpx` i `cpy` służą głównie do wykrywania końca pętli liczącej w górę:

```
        ldx #0                  ; X = 0 - pierwszy indeks
petla   lda zrodlo,x            ; przepisz bajt
        sta cel,x               ; zapisz A pod adres cel+X
        inx                     ; następny indeks
        cpx #40                 ; czy X doszedł do 40?
        bne petla               ; jeszcze nie - powtarzaj (pętla wykona się dla X = 0..39)
```

6.10 Skoki warunkowe (rozgałęzienia)

Osiem rozkazów, które badają **jeden znacznik** i — jeśli ma on wymaganą wartość — przenoszą wykonanie pod wskazaną etykietę. W przeciwnym razie program biegnie dalej, do następnego rozkazu. Wszystkie używają trybu względnego (zasięg −128…+127 bajtów), zajmują 2 bajty i nie zmieniają znaczników.

Rozkaz	Skacze, gdy	Kod	Typowe znaczenie po `cmp`
`beq`	Z = 1	`$F0`	równe
`bne`	Z = 0	`$D0`	różne
`bcs`	C = 1	`$B0`	większe lub równe (bez znaku)
`bcc`	C = 0	`$90`	mniejsze (bez znaku)
`bmi`	N = 1	`$30`	wynik ujemny (bit 7 = 1)
`bpl`	N = 0	`$10`	wynik nieujemny (bit 7 = 0)
`bvs`	V = 1	`$70`	wystąpił nadmiar
`bvc`	V = 0	`$50`	nie było nadmiaru

Czas wykonania: 2 cykle, gdy skok nie jest wykonywany; 3 cykle, gdy jest; 4 cykle, gdy cel leży na innej stronie pamięci.

6.10.1 BEQ — skocz, jeśli równe / zero (*Branch if EQual*)

N	V	B	D	I	Z	C
–	–	–	–	–	–	–

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
względny	`beq etykieta`	`$F0`	2	2**

```
        lda klawisz             ; A = kod klawisza
        cmp #27                 ; czy to kod klawisza ESC?
        beq wyjscie             ; tak (Z = 1) - skocz do "wyjscie"
```

6.10.2 BNE — skocz, jeśli różne / nie zero (*Branch if Not Equal*)

N	V	B	D	I	Z	C
–	–	–	–	–	–	–

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
względny	`bne etykieta`	`$D0`	2	2**

```
        dex                     ; X = X - 1
        bne petla               ; dopóki X <> 0 (Z = 0), wracaj na początek pętli
```

6.10.3 BCS — skocz, jeśli jest przeniesienie (*Branch if Carry Set*)

N	V	B	D	I	Z	C
–	–	–	–	–	–	–

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
względny	`bcs etykieta`	`$B0`	2	2**

```
        lda punkty              ; A = liczba punktów
        cmp #100                ; porównaj ze 100
        bcs nagroda             ; C = 1: punkty >= 100
```

6.10.4 BCC — skocz, jeśli nie ma przeniesienia (*Branch if Carry Clear*)

N	V	B	D	I	Z	C
–	–	–	–	–	–	–

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
względny	`bcc etykieta`	`$90`	2	2**

```
        clc                     ; C = 0
        adc #50                 ; A = A + 50
        bcc zmiescil_sie        ; C = 0: suma nie przekroczyła 255
```

6.10.5 BMI — skocz, jeśli minus (*Branch if MInus*)

N	V	B	D	I	Z	C
–	–	–	–	–	–	–

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
względny	`bmi etykieta`	`$30`	2	2**

```
        lda $0343               ; A = status operacji wejścia/wyjścia (ICSTA)
        bmi blad                ; bit 7 ustawiony (wartość >= 128) oznacza błąd
```

6.10.6 BPL — skocz, jeśli plus (*Branch if PLus*)

N	V	B	D	I	Z	C
–	–	–	–	–	–	–

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
względny	`bpl etykieta`	`$10`	2	2**

```
        ldx #7                  ; X = 7
petla   sta tablica,x           ; wypełnij element X
        dex                     ; X = X - 1
        bpl petla               ; dopóki X >= 0 (N = 0); pętla obejmie także X = 0
```

Pętla z `bpl` jest wygodna, bo — inaczej niż `bne` — wykonuje się również dla indeksu zero. Działa jednak poprawnie tylko dla wartości początkowych do 128 (`$80`): już pierwsze `dex` dla większej wartości zostawia ustawiony bit 7 i pętla skończyłaby się po jednym obiegu.

6.10.7 BVS — skocz, jeśli nadmiar (*Branch if oVerflow Set*)

N	V	B	D	I	Z	C
–	–	–	–	–	–	–

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
względny	`bvs etykieta`	`$70`	2	2**

6.10.8 BVC — skocz, jeśli brak nadmiaru (*Branch if oVerflow Clear*)

N	V	B	D	I	Z	C
–	–	–	–	–	–	–

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
względny	`bvc etykieta`	`$50`	2	2**

```
        lda temp1               ; A = liczba ze znakiem (-128..+127)
        clc                     ; C = 0 (przed dodawaniem)
        adc temp2               ; dodaj drugą liczbę ze znakiem
        bvs przepelnienie       ; V = 1: wynik nie mieści się w zakresie -128..+127
```

Znacznik V przydaje się też z rozkazem `bit` (bada bit 6 komórki). Ponieważ niewiele rozkazów go zmienia, para `clv` + `bvc` bywa stosowana jako „skok bezwarunkowy względny" — krótszy o bajt niż `jmp` i niezależny od położenia kodu.

6.11 Skoki, podprogramy, przerwania

6.11.1 JMP — skok bezwarunkowy (*JuMP*)

Działanie: PC ← adres. Program jest kontynuowany od podanego miejsca.

N	V	B	D	I	Z	C
–	–	–	–	–	–	–

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
absolutny	`jmp $4400`	`$4C`	3	3
pośredni	`jmp ($4400)`	`$6C`	3	5

```
petla   jsr rysuj               ; główna pętla programu
        jsr ruch                ; wywołaj podprogram ruch
        jmp petla               ; skocz na początek - pętla nieskończona

        jmp ($000A)             ; skok pośredni: pod adres ZAPISANY w komórkach $0A i $0B
                                ; (wektor DOSVEC - wyjście do DOS-u)
```

6.11.2 JSR — skok do podprogramu (*Jump to SubRoutine*)

Działanie: odkłada na stos adres powrotu (dokładnie: adres ostatniego bajtu rozkazu `jsr`, najpierw starszy bajt, potem młodszy), po czym PC ← adres.

N	V	B	D	I	Z	C
–	–	–	–	–	–	–

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
absolutny	`jsr $4400`	`$20`	3	6

6.11.3 RTS — powrót z podprogramu (*ReTurn from Subroutine*)

Działanie: zdejmuje ze stosu dwa bajty adresu, dodaje 1 i umieszcza wynik w PC. Program wraca do rozkazu następnego po `jsr`.

N	V	B	D	I	Z	C
–	–	–	–	–	–	–

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
implikowany	`rts`	`$60`	1	6

```
        jsr czekaj              ; wywołaj podprogram "czekaj"; na stos trafia adres powrotu
        lda #0                  ; <- tu wróci wykonanie po rozkazie RTS
        rts                     ; powrót z podprogramu

czekaj  ldx #0                  ; podprogram: krótka pętla opóźniająca
cz1     dex                     ; X = X - 1
        bne cz1                 ; 256 obiegów
        rts                     ; zdejmij adres ze stosu i wróć do wywołującego
```

Podprogramy mogą wywoływać kolejne podprogramy; każdy poziom zagnieżdżenia zajmuje 2 bajty stosu. Całość omawia rozdział 11.

6.11.4 RTI — powrót z przerwania (*ReTurn from Interrupt*)

Działanie: zdejmuje ze stosu rejestr znaczników P, a następnie adres powrotu do PC (bez dodawania 1).

N	V	B	D	I	Z	C
✓	✓	–	✓	✓	✓	✓

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
implikowany	`rti`	`$40`	1	6

Kończy procedurę obsługi przerwania. Ponieważ odtwarza rejestr P w całości, przerwany program nie zauważy żadnej zmiany znaczników.

```
dli     pha                     ; procedura przerwania: zachowaj A (przerwany program go używa!)
        lda #$34                ; nowy kolor
        sta $D40A               ; WSYNC: poczekaj do końca bieżącej linii obrazu
        sta $D01A               ; COLBK: zmień kolor tła
        pla                     ; odtwórz A
        rti                     ; wróć do przerwanego programu
```

6.11.5 BRK — przerwanie programowe (*BReaK*)

Działanie: odkłada na stos PC + 2 oraz rejestr P z ustawionym bitem B, ustawia znacznik I i skacze pod adres z wektora `$FFFE/$FFFF`.

N	V	B	D	I	Z	C
–	–	1	–	1	–	–

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
implikowany	`brk`	`$00`	1	7

Choć `brk` ma jeden bajt, adres powrotu wskazuje o jeden bajt dalej — bajt za `brk` jest pomijany. Kod rozkazu to `$00`, więc procesor, który „wpadnie" w wyzerowany obszar pamięci, wykona właśnie `brk`. W Atari system operacyjny przekazuje wtedy sterowanie przez wektor VBREAK (`$0206`). W praktyce `brk` wykorzystuje się jako pułapkę dla debuggera — Altirra zatrzymuje się na tym rozkazie.

6.12 Rozkazy sterujące znacznikami

Siedem rozkazów jednobajtowych po 2 cykle. Każdy ustawia lub kasuje jeden znacznik.

6.12.1 CLC — skasuj przeniesienie (*CLear Carry*)

Działanie: C ← 0. Obowiązkowy przed dodawaniem.

N	V	B	D	I	Z	C
–	–	–	–	–	–	0

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
implikowany	`clc`	`$18`	1	2

6.12.2 SEC — ustaw przeniesienie (*SEt Carry*)

Działanie: C ← 1. Obowiązkowy przed odejmowaniem.

N	V	B	D	I	Z	C
–	–	–	–	–	–	1

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
implikowany	`sec`	`$38`	1	2

```
        clc                 ; C = 0 przed dodawaniem
        adc #5              ; A = A + 5
        sec                 ; C = 1 przed odejmowaniem
        sbc #3              ; A = A - 3
```

Znacznik C służy też jako wygodna, jednobitowa **wartość zwracana z podprogramu**: `clc` + `rts` oznacza „udało się", `sec` + `rts` — „błąd". Wywołujący sprawdza wynik rozkazem `bcs`.

6.12.3 CLD — wyłącz tryb dziesiętny (*CLear Decimal*)

Działanie: D ← 0. Rozkazy `adc` i `sbc` liczą dwójkowo (stan normalny).

N	V	B	D	I	Z	C
–	–	–	0	–	–	–

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
implikowany	`cld`	`$D8`	1	2

6.12.4 SED — włącz tryb dziesiętny (*SEt Decimal*)

Działanie: D ← 1. Rozkazy `adc` i `sbc` traktują bajty jako liczby BCD.

N	V	B	D	I	Z	C
–	–	–	1	–	–	–

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
implikowany	`sed`	`$F8`	1	2

```
        sed                     ; włącz arytmetykę dziesiętną
        lda #$19                ; A = 19 w kodzie BCD
        clc                     ; C = 0 (przed dodawaniem)
        adc #$01                ; A = $20 (dwadzieścia), a nie $1A, jak byłoby dwójkowo
        cld                     ; KONIECZNIE wyłącz tryb dziesiętny!
```

Pułapka

Pozostawiony włączony tryb dziesiętny psuje każde następne dodawanie i odejmowanie w całym programie. Zawsze zamykaj fragment BCD rozkazem `cld`. Szczegóły w rozdziale 16.

6.12.5 CLI — odblokuj przerwania (*CLear Interrupt disable*)

Działanie: I ← 0. Procesor zaczyna reagować na przerwania IRQ.

N	V	B	D	I	Z	C
–	–	–	–	0	–	–

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
implikowany	`cli`	`$58`	1	2

6.12.6 SEI — zablokuj przerwania (*SEt Interrupt disable*)

Działanie: I ← 1. Procesor ignoruje przerwania IRQ (ale nie NMI).

N	V	B	D	I	Z	C
–	–	–	–	1	–	–

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
implikowany	`sei`	`$78`	1	2

```
        sei                     ; zablokuj IRQ - zaczyna się sekcja krytyczna
        lda #<moja_obsluga      ; zmień wektor przerwania klawiatury...
        sta $0208               ; zapisz A do komórki $0208
```

```
        lda #>moja_obsluga      ; ...na adres własnej procedury
        sta $0209               ; zapisz A do komórki $0209
        cli                     ; odblokuj IRQ
```

W Atari `sei` blokuje między innymi obsługę klawiatury. Przerwań NMI (VBI, DLI) nie da się zablokować rozkazem procesora — wyłącza się je w rejestrze NMIEN układu ANTIC (rozdział 24).

6.12.7 CLV — skasuj nadmiar (*CLear oVerflow*)

Działanie: V ← 0. Nie istnieje rozkaz ustawiający V.

N	V	B	D	I	Z	C
–	0	–	–	–	–	–

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
implikowany	`clv`	`$B8`	1	2

6.13 Rozkaz pusty

6.13.1 NOP — nic nie rób (*No OPeration*)

Działanie: żadne. Procesor zużywa 2 cykle i przechodzi do następnego rozkazu.

N	V	B	D	I	Z	C
–	–	–	–	–	–	–

Tryb adresowania	Zapis w MADS	Kod	Bajty	Cykle
implikowany	`nop`	`$EA`	1	2

Zastosowania: precyzyjne odmierzanie czasu (np. w procedurach zmieniających kolory w trakcie rysowania linii obrazu), rezerwowanie miejsca na późniejszą modyfikację kodu, tymczasowe „wyłączanie” rozkazu podczas szukania błędu.

```
        sta $D40A               ; WSYNC - zaczekaj na początek linii
        nop                     ; 2 cykle zwłoki...
        nop                     ; ...jeszcze 2 - aby zmiana koloru wypadła w dobrym miejscu linii
        sta $D01A               ; zmień kolor tła
```

6.14 Zestawienie: który rozkaz zmienia które znaczniki

Znaczniki	Rozkazy
N, Z	`lda ldx ldy`, `tax tay txa tya tsx`, `pla`, `inc inx iny`, `dec dex dey`, `and ora eor`
N, Z, C	`asl lsr rol ror`, `cmp cpx cpy`
N, V, Z, C	`adc sbc`
N, V, Z	`bit`
wszystkie	`plp`, `rti`
tylko jeden	`clc sec` (C), `cld sed` (D), `cli sei` (I), `clv` (V)
żadnego	`sta stx sty`, `txs`, `pha php`, wszystkie skoki (`jmp jsr rts`, `bxx`), `nop`

6.15 Czego w 6502 nie ma

Lista braków jest równie ważna jak lista rozkazów — od razu wiadomo, co trzeba zbudować samemu:

- **mnożenia i dzielenia** — rozdziały 14 i 15,
- **dodawania i odejmowania bez przeniesienia** — stąd `clc`/`sec`,
- **zwiększania i zmniejszania akumulatora** — `clc` + `adc #1`, `sec` + `sbc #1`,
- **bezpośredniego przesłania X ↔ Y** — przez akumulator,
- **odkładania X i Y na stos** — przez akumulator,
- **rejestrów 16-bitowych** — arytmetyka wielobajtowa bajt po bajcie (rozdział 12),
- **skoków warunkowych dalekiego zasięgu** — odwrócony warunek i `jmp` (rozdział 7),
- **skoku do podprogramu przez wskaźnik** `jsr (adres)` — rozdział 10,
- **adresowania pośredniego bez indeksu** `lda (wsk)` — używamy `(wsk),y` z $Y = 0$.

Wszystkie te braki uzupełnił późniejszy procesor 65C02, ale w Atari XL/XE go nie ma. Piszemy na klasycznym 6502.

Cześć II

Struktury sterujące

ROZDZIAŁ 7

Porównania i skoki warunkowe

W językach wysokiego poziomu piszemy `IF A>5 THEN...` i kompilator zajmuje się resztą. W asemblerze każda decyzja składa się z **dwóch osobnych kroków**:

1. rozkaz, który **ustawia znaczniki** (porównanie, operacja arytmetyczna, załadowanie wartości),
2. rozkaz **skoku warunkowego**, który bada jeden znacznik.

Cała sztuka polega na tym, by wiedzieć, który znacznik odpowiada na zadane pytanie.

7.1 Porównanie z zerem — bez `cmp`

Ponieważ `lda`, `ldx`, `ldy`, `inc`, `dec`, `and`, `adc` i wiele innych rozkazów same ustawiają znaczniki Z i N, porównanie z zerem nie wymaga żadnego dodatkowego rozkazu:

```
        lda licznik             ; załadowanie wartości ustawia Z i N
        beq jest_zero           ; Z = 1  ->  licznik = 0
        bmi ujemny              ; N = 1  ->  bit 7 ustawiony (wartość >= 128 lub ujemna w kodzie U2)
                                ; tutaj: wartość 1..127
```

Pułapka

Znaczniki pochodzą zawsze z **ostatniego** rozkazu, który je zmienił. W poniższym fragmencie `beq` bada wynik `ldx`, a nie `lda`:

```
        lda licznik             ; ustawia Z według zmiennej "licznik"...
        ldx #5                  ; ...ale ten rozkaz ustawia Z od nowa (Z = 0, bo 5 <> 0)
        beq jest_zero           ; ten skok NIGDY się nie wykona
```

7.2 Porównywanie liczb bez znaku

Rozkaz `cmp` oblicza A − M i ustawia znaczniki. Wynik opisują dwa z nich: **Z** (równość) i **C** (brak pożyczki, czyli A ≥ M).

Warunek	Kod po `lda a` / `cmp b`	Uwagi
a = b	`beq prawda`	
a ≠ b	`bne prawda`	
a < b	`bcc prawda`	
a ≥ b	`bcs prawda`	
a > b	`beq falsz` / `bcs prawda`	dwa skoki albo zamiana argumentów
a ≤ b	`bcc prawda` / `beq prawda`	dwa skoki albo zamiana argumentów

Procesor daje wprost tylko „mniejsze” i „większe lub równe”. Dwa pozostałe warunki uzyskujemy na jeden z dwóch sposobów.

Sposób 1 — dwa skoki:

```
        lda a                   ; A = a
        cmp b                   ; porównaj z b
        beq nie_wieksze         ; a = b, więc nie jest większe - omiń
        bcs wieksze             ; a >= b i a <> b, czyli a > b
nie_wieksze
```

Sposób 2 — zamiana argumentów miejscami (elegantszy: „a > b” to to samo co „b < a”):

```
        lda b                   ; A = b (argumenty zamienione!)
        cmp a                   ; porównaj b z a
        bcc wieksze             ; b < a, czyli a > b
```

Przy porównaniu ze **stałą** jest jeszcze prostsza droga: zmienić stałą o jeden. Dla liczb całkowitych „a > 9” to to samo co „a ≥ 10”:

```
        lda a                   ; A = a
        cmp #10                 ; porównaj z 10 zamiast z 9
        bcs wieksze_od_9        ; a >= 10, czyli a > 9
```

7.3 Porównywanie liczb ze znakiem

Znacznik C daje poprawną odpowiedź tylko dla liczb bez znaku. Przykład: −1 to bajt `$FF`, a +1 to `$01`; rozkaz `cmp` uzna, że `$FF` > `$01`. Dla liczb w kodzie U2 trzeba badać **znak różnicy**, uwzględniając możliwe przepełnienie. Poprawny wzorzec:

```
; Skok do "mniejsze", gdy a < b - obie liczby traktowane jako -128..+127
        lda a                   ; A = a
        sec                     ; przygotuj odejmowanie
        sbc b                   ; A = a - b; ustawia N (znak wyniku) i V (przepełnienie)
        bvc bez_nadm            ; V = 0: znak wyniku jest wiarygodny
        eor #$80                ; V = 1: znak wyniku jest odwrócony - popraw go
bez_nadm bmi mniejsze           ; N = 1: a < b
                                ; tutaj: a >= b
```

Dlaczego tak? Wynik odejmowania jest ujemny, gdy a < b — chyba że nastąpiło przepełnienie, które odwraca bit znaku. Rozkaz `eor #$80` odwraca go wtedy z powrotem (i przy okazji ustawia znacznik N według poprawionego bitu 7).

Używamy tu `sbc`, a nie `cmp`, bo **`cmp` nie ustawia znacznika V**.

> **Wskazówka**
>
> W grach i programach użytkowych na Atari liczby ze znakiem są rzadkie — współrzędne, liczniki, adresy i kody znaków to liczby bez znaku. Jeśli tylko możesz, projektuj dane tak, by znak nie był potrzebny (na przykład przesuwając zakres: zamiast −20…+20 przechowuj 0…40).

7.4 Badanie pojedynczych bitów

Często pytamy nie o całą wartość, lecz o jeden bit. Jest na to kilka sposobów:

```
; --- bit 7: wystarczy załadować wartość ---
        lda rejestr             ; N = bit 7
        bmi bit7_jest           ; skocz, jeśli bit 7 = 1

; --- dowolny bit: maska i AND (niszczy A) ---
        lda rejestr             ; A = rejestr
        and #%00001000          ; zostaw tylko bit 3
        bne bit3_jest           ; wynik <> 0, więc bit 3 = 1

; --- bity 7 i 6 bez niszczenia A: rozkaz BIT ---
        bit rejestr             ; N = bit 7, V = bit 6 komórki "rejestr"
        bmi bit7_jest           ; skocz do bit7_jest, gdy N = 1 (bit 7 ustawiony)
        bvs bit6_jest           ; skocz do bit6_jest, gdy V = 1

; --- bit 0: wysuń go do znacznika C ---
        lda rejestr             ; A = rejestr
        lsr @                   ; bit 0 -> C
        bcs bit0_jest           ; skocz, jeśli bit 0 = 1
```

7.5 Skoki dalekiego zasięgu

Skok warunkowy sięga najwyżej około 127 bajtów. Gdy cel jest dalej, MADS zgłasza błąd *Branch out of range*. Rozwiązanie jest zawsze takie samo: **odwróć warunek i przeskocz rozkaz `jmp`**.

```
; Chcemy: jeśli równe, skocz do odległej etykiety "daleko"
        cmp #5                  ; porównanie
        bne nie                 ; warunek ODWROTNY (różne): omiń skok daleki
        jmp daleko              ; równe: skocz - JMP sięga w dowolne miejsce pamięci
nie
```

Pary warunków przeciwnych: `beq` ↔ `bne`, `bcc` ↔ `bcs`, `bmi` ↔ `bpl`, `bvc` ↔ `bvs`.

(Pseudorozkazy MADS `jeq`, `jne`, `jcc`… generują dokładnie taką parę rozkazów. Pisząc ją jawnie, widzimy, że skok daleki kosztuje 5 bajtów zamiast 2.)

7.6 Porównywanie liczb 16-bitowych

7.6.1 Równość

Dwie liczby są równe, gdy równe są oba ich bajty:

```
        lda a                   ; porównaj młodsze bajty
        cmp b                   ; porównaj A z: b
        bne rozne               ; różnią się - liczby różne, nie trzeba sprawdzać dalej
        lda a+1                 ; porównaj starsze bajty
        cmp b+1                 ; porównaj A z: bajt spod adresu b+1
        bne rozne               ; skocz do rozne, gdy Z = 0 (różne / nie zero)
                                ; tutaj: a = b
```

Sprawdzenie, czy liczba 16-bitowa **jest zerem**, zapisuje się krócej:

```
        lda a                   ; młodszy bajt
        ora a+1                 ; suma logiczna ze starszym: zero tylko gdy oba bajty są zerami
        beq jest_zero           ; skocz do jest_zero, gdy Z = 1 (równe / zero)
```

7.6.2 Mniejsze / większe (bez znaku) — metoda „od starszego bajtu”

Postępujemy jak przy porównywaniu liczb wielocyfrowych: najpierw najbardziej znacząca część.

```
        lda a+1                 ; porównaj STARSZE bajty
        cmp b+1                 ; porównaj A z: bajt spod adresu b+1
        bcc mniejsze            ; starszy bajt a < starszy bajt b: na pewno a < b
        bne wieksze             ; starsze bajty różne i nie mniejsze: na pewno a > b
        lda a                   ; starsze bajty równe - rozstrzygają młodsze
        cmp b                   ; porównaj A z: b
        bcc mniejsze            ; a < b
        beq rowne               ; a = b
wieksze                         ; tutaj: a > b
```

7.6.3 Mniejsze / większe — metoda odejmowania

Jeżeli wystarczy odpowiedź „a < b” albo „a ≥ b”, jest krótszy sposób. Wykonujemy pełne 16-bitowe odejmowanie na próbę, nie zapisując wyniku:

```
        lda a                   ; młodsze bajty: CMP ustawia C tak samo jak odejmowanie
        cmp b                   ; porównaj A z: b
        lda a+1                 ; starsze bajty: SBC uwzględnia pożyczkę z młodszych
        sbc b+1                 ; A = A - bajt spod adresu b+1 - pożyczka (gdy C = 0)
        bcc mniejsze            ; C = 0: a < b
                                ; tutaj: a >= b  (uwaga: znacznik Z NIE oznacza tu równości!)
```

Pierwsze porównanie zostawia w C informację o pożyczce, a `sbc` ją wykorzystuje — dokładnie jak w odejmowaniu wielobajtowym. Końcowy stan C dotyczy całych liczb 16-bitowych.

7.6.4 Liczby 16-bitowe ze znakiem

Łączymy metodę odejmowania z poprawką na przepełnienie:

```
        lda a                   ; młodsze bajty
        cmp b                   ; porównaj A z: b
        lda a+1                 ; starsze bajty z pożyczką
        sbc b+1                 ; N i V opisują wynik 16-bitowy
        bvc bez_nad             ; brak przepełnienia: N jest poprawny
        eor #$80                ; przepełnienie: odwróć bit znaku
bez_nad bmi mniejsze            ; a < b (ze znakiem)
                                ; tutaj: a >= b
```

7.7 Porównywanie liczb 32-bitowych

Metoda odejmowania rozszerza się w oczywisty sposób — po prostu przedłużamy łańcuch:

```
        lda a                   ; bajt 0 (najmłodszy)
        cmp b                   ; porównaj A z: b
        lda a+1                 ; bajt 1
        sbc b+1                 ; A = A - bajt spod adresu b+1 - pożyczka (gdy C = 0)
        lda a+2                 ; bajt 2
        sbc b+2                 ; A = A - bajt spod adresu b+2 - pożyczka (gdy C = 0)
        lda a+3                 ; bajt 3 (najstarszy)
        sbc b+3                 ; A = A - bajt spod adresu b+3 - pożyczka (gdy C = 0)
        bcc mniejsze            ; C = 0: a < b (bez znaku)
                                ; tutaj: a >= b
```

Równość liczb 32-bitowych sprawdzamy czterema parami `lda/cmp/bne` albo w pętli:

```
        ldx #3                  ; X = 3: zaczynamy od najstarszego bajtu
por     lda a,x                 ; bajt numer X pierwszej liczby
        cmp b,x                 ; porównaj z bajtem numer X drugiej liczby
        bne rozne               ; pierwsza różnica kończy sprawdzanie
        dex                     ; następny (młodszy) bajt
        bpl por                 ; powtarzaj dla X = 3, 2, 1, 0
                                ; tutaj: wszystkie cztery bajty równe
```

Ponieważ ta pętla idzie od najstarszego bajtu, po wyjściu skokiem `bne rozne` znacznik C mówi, która liczba jest większa (C = 0: a < b). To pełne porównanie 32-bitowe w siedmiu rozkazach.

Ćwiczenia

1. Napisz fragment, który skacze do `ok`, gdy zmienna `x` zawiera wartość z przedziału 10…20 włącznie.
2. Dlaczego po `lda a` / `cmp b` / `lda a+1` / `sbc b+1` nie można użyć `beq` do sprawdzenia równości?
3. Zamień warunek „a ≤ b” na postać używającą tylko `bcs`.

Odpowiedzi: (1) `lda x` / `cmp #10` / `bcc nie` / `cmp #21` / `bcc ok` / `nie`. (2) Bo Z opisuje tylko wynik ostatniego odejmowania, czyli starszych bajtów. (3) `lda b` / `cmp a` / `bcs prawda` — czyli „b ≥ a”.

ROZDZIAŁ 8

Instrukcje warunkowe: IF … THEN … ELSE

Każdą instrukcję warunkową z języka wysokiego poziomu da się zapisać w asemblerze według stałego szablonu. Gdy opanujesz kilka wzorców z tego rozdziału, będziesz je pisać odruchowo.

8.1 IF … THEN

Zasada podstawowa: **skaczemy wtedy, gdy warunek NIE jest spełniony** — omijamy blok THEN.

```
   IF warunek THEN              sprawdź warunek
      instrukcje                skocz do "dalej", jeśli warunek FAŁSZYWY
   ENDIF                        instrukcje
                              dalej
```

Przykład: `IF zycia = 0 THEN koniec_gry := 1`

```
        lda zycia               ; A = zycia; znacznik Z = 1, gdy zycia = 0
        bne dalej               ; warunek odwrotny (zycia <> 0): omiń blok THEN
        lda #1                  ; --- blok THEN ---
        sta koniec_gry          ; koniec_gry := 1
dalej                           ; --- ENDIF ---
```

Przykład: `IF x >= 40 THEN x := 0`

```
        lda x                   ; A = x
        cmp #40                 ; porównaj z 40
        bcc dalej               ; warunek odwrotny (x < 40): omiń blok
        lda #0                  ; --- blok THEN ---
        sta x                   ; x := 0
dalej
```

Tabela odwracania warunków dla liczb bez znaku (po `lda a` / `cmp b`):

Warunek w IF	Skok omijający blok THEN
$a = b$	`bne dalej`
$a \neq b$	`beq dalej`
$a < b$	`bcs dalej`
$a \geq b$	`bcc dalej`
$a > b$	`bcc dalej` / `beq dalej`
$a \leq b$	`beq wykonaj` / `bcs dalej` / `wykonaj`

8.2 IF … THEN … ELSE

Blok THEN musi kończyć się skokiem `jmp`, który omija blok ELSE:

```
   IF warunek THEN              sprawdź warunek
      instrukcje_T              skocz do "inaczej", jeśli warunek FAŁSZYWY
   ELSE                         instrukcje_T
      instrukcje_E              jmp koniec
   ENDIF                      inaczej
                                 instrukcje_E
                              koniec
```

Przykład: `IF punkty >= 100 THEN kolor := $C8 ELSE kolor := $34`

```
        lda punkty              ; A = punkty
        cmp #100                ; porównaj ze 100
        bcc inaczej             ; punkty < 100: przejdź do bloku ELSE
        lda #$C8                ; --- blok THEN: kolor zielony ---
        sta kolor               ; zapisz A do kolor
        jmp koniec              ; omiń blok ELSE
inaczej lda #$34                ; --- blok ELSE: kolor czerwony ---
        sta kolor               ; zapisz A do kolor
koniec                          ; --- ENDIF ---
```

Pułapka

Zapomniany `jmp koniec` po bloku THEN to klasyczny błąd: program wykona blok THEN, a zaraz po nim — również blok ELSE.

8.2.1 Oszczędniejsze warianty

Gdy oba bloki tylko nadają wartość tej samej zmiennej, można zacząć od wartości domyślnej i ewentualnie ją zmienić. Znika `jmp`:

```
        ldx #$34                ; załóż wynik bloku ELSE: X = kolor czerwony
        lda punkty              ; A = punkty
        cmp #100                ; porównaj ze 100
        bcc zapisz              ; punkty < 100: zostaje wartość domyślna
        ldx #$C8                ; punkty >= 100: X = kolor zielony
zapisz  stx kolor               ; zapisz wybrany kolor
```

Jeżeli po bloku THEN stan któregoś znacznika jest pewny, zamiast trzybajtowego `jmp` można użyć dwubajtowego skoku warunkowego, który wykona się zawsze:

```
        lda #$C8                ; A = $C8: wartość niezerowa, więc Z = 0
        sta kolor               ; STA nie zmienia znaczników
        bne koniec              ; Z = 0 na pewno - ten skok wykona się zawsze
```

Taki skok koniecznie opatrz komentarzem „zawsze” — inaczej za miesiąc nie będziesz pewien, czy to nie błąd.

8.3 Warunki złożone: AND

`IF (a = 1) AND (b = 2) THEN ...` — blok wykonuje się tylko wtedy, gdy spełnione są **wszystkie** warunki. Wystarczy więc, że **którykolwiek** zawiedzie, by blok ominąć:

```
        lda a                   ; pierwszy warunek: a = 1 ?
        cmp #1                  ; porównaj A z: 1
        bne dalej               ; nie - całość fałszywa, omiń blok
        lda b                   ; drugi warunek: b = 2 ?
        cmp #2                  ; porównaj A z: 2
        bne dalej               ; nie - całość fałszywa, omiń blok
        inc licznik             ; --- blok THEN: oba warunki spełnione ---
dalej
```

Zauważ, że drugi warunek w ogóle nie jest sprawdzany, gdy pierwszy okazał się fałszywy. Warto ustawiać na początku warunek, który najczęściej zawodzi lub który najszybciej się sprawdza.

8.3.1 Sprawdzanie zakresu

Najczęstszy warunek złożony: „czy wartość leży w przedziale”. Przykład — czy znak jest cyfrą: `IF (znak >= '0') AND (znak <= '9')`

```
        lda znak                ; A = kod znaku
        cmp #'0'                ; porównaj z kodem cyfry 0
        bcc nie_cyfra           ; znak < '0': poza zakresem
        cmp #'9'+1              ; porównaj z kodem następnym po '9'
        bcs nie_cyfra           ; znak >= '9'+1, czyli znak > '9': poza zakresem
                                ; --- tutaj: znak jest cyfrą ---
```

Drugie porównanie używa wartości `'9'+1`, żeby zamienić niewygodny warunek „większe” na dostępny wprost „większe lub równe”.

8.4 Warunki złożone: OR

`IF (a = 1) OR (b = 2) THEN ...` — wystarczy, że **jeden** warunek jest prawdziwy. Pierwszy spełniony warunek skacze od razu do bloku THEN; dopiero ostatni warunek, jeśli jest fałszywy, blok omija:

```
        lda a                   ; pierwszy warunek: a = 1 ?
        cmp #1                  ; porównaj A z: 1
        beq wykonaj             ; tak - całość prawdziwa, od razu do bloku THEN
        lda b                   ; drugi (ostatni) warunek: b = 2 ?
        cmp #2                  ; porównaj A z: 2
        bne dalej               ; nie - żaden warunek nie jest spełniony, omiń blok
wykonaj inc licznik             ; --- blok THEN ---
dalej
```

Przykład praktyczny — reakcja na klawisz `T` niezależnie od wielkości litery:

```
        lda klawisz             ; A = kod ATASCII naciśniętego klawisza
        cmp #'T'                ; wielkie T ?
        beq tak                 ; tak
        cmp #'t'                ; małe t ?
        bne nie                 ; też nie - to inny klawisz
tak     jsr potwierdz           ; --- naciśnięto T lub t ---
nie
```

Gdy porównujemy tę samą wartość z kilkoma stałymi, nie trzeba ładować jej ponownie: `cmp` nie zmienia akumulatora.

8.5 Zagnieżdżanie

Instrukcje warunkowe wewnątrz innych zapisuje się tak samo — każda ma własne etykiety. Przykład: ograniczenie ruchu w poziomie do zakresu 0…39.

```
   IF kierunek = PRAWO THEN
      IF x < 39 THEN x := x + 1
   ELSE
      IF x > 0 THEN x := x - 1
   ENDIF
```

```
        lda kierunek            ; A = kierunek ruchu
        cmp #PRAWO              ; czy w prawo?
        bne w_lewo              ; nie - blok ELSE
        lda x                   ; --- THEN: ruch w prawo ---
        cmp #39                 ; czy x < 39 ?
        bcs koniec              ; x >= 39: jesteśmy przy prawej krawędzi, bez ruchu
        inc x                   ; x := x + 1
        jmp koniec              ; omiń blok ELSE
w_lewo  lda x                   ; --- ELSE: ruch w lewo ---
        beq koniec              ; x = 0: jesteśmy przy lewej krawędzi, bez ruchu
        dec x                   ; x := x - 1
koniec
```

8.6 Przydatne wzorce

8.6.1 Minimum i maksimum

```
; wynik := MAX(a, b)
        lda a                   ; załóż, że większe jest a
        cmp b                   ; porównaj z b
        bcs mam_max             ; a >= b: w A jest już maksimum
```

```
        lda b                   ; a < b: maksimum to b
mam_max sta wynik               ; zapisz A do wynik
```

8.6.2 Ograniczenie wartości z góry (nasycenie)

```
; x := x + 5, ale nie więcej niż 200
        lda x                   ; A = x
        clc                     ; C = 0 (przed dodawaniem)
        adc #5                  ; A = x + 5
        bcs obetnij             ; C = 1: suma przekroczyła 255 - na pewno za dużo
        cmp #201                ; czy wynik <= 200 ?
        bcc zapisz              ; tak - w porządku
obetnij lda #200                ; nie - przytnij do 200
zapisz  sta x                   ; zapisz A do x
```

8.6.3 Wartość bezwzględna liczby ze znakiem

```
        lda liczba              ; A = liczba w kodzie U2; N = bit znaku
        bpl dodatnia            ; nieujemna - nic nie trzeba robić
        eor #$FF                ; ujemna: odwróć wszystkie bity...
        clc                     ; C = 0 (przed dodawaniem)
        adc #1                  ; ...i dodaj 1 (zmiana znaku w kodzie U2)
dodatnia sta modul              ; zapisz |liczba|
```

8.6.4 Zmienna logiczna

Wartości „prawda/fałsz” wygodnie przechowywać jako `$00` i `$FF` albo `$00` i `$80`. Wtedy test nie wymaga `cmp`, a z rozkazem `bit` — nawet akumulatora:

```
        lda #$FF                ; ustaw flagę "dźwięk włączony"
        sta dzwiek              ; zapisz A do dzwiek
                                ; ... gdzie indziej w programie ...
        bit dzwiek              ; N = bit 7 flagi; akumulator zostaje nienaruszony
        bpl cicho               ; bit 7 = 0: flaga wyłączona
        jsr graj                ; flaga włączona
cicho
```

Ćwiczenia

1. Zapisz w asemblerze: `IF (x > 10) AND (y < 5) THEN z := 1 ELSE z := 0`.
2. Zapisz: `IF znak jest wielką literą (A..Z) THEN zamień na małą` (małe litery mają w ATASCII kody o 32 większe).
3. Przepisz przykład z kolorami tak, by nie używał ani `jmp`, ani rejestru X.

Wskazówka do 3: zacznij od `lda punkty` / `cmp #100`, potem `lda #$34` — pamiętając, że `lda` **nie zmienia znacznika C**, więc wynik porównania przetrwa załadowanie nowej wartości.

ROZDZIAŁ 9

Pętle: FOR, WHILE, REPEAT

Pętla w asemblerze to nic innego jak skok wstecz. Wszystkie rodzaje pętli znane z języków wysokiego poziomu buduje się z trzech klocków: **licznika**, **ciała** i **skoku warunkowego**.

9.1 Pętla FOR licząca w dół — najprostsza i najszybsza

Najbardziej naturalna dla 6502 jest pętla, w której licznik **maleje do zera**, bo rozkazy `dex`/`dey`/`dec` same ustawiają znacznik Z — nie trzeba żadnego porównania.

```
    FOR i := 10 DOWNTO 1 DO ciało
```

```
        ldx #10                 ; X = 10: liczba powtórzeń
petla   jsr cialo               ; --- ciało pętli (X kolejno: 10, 9, ... 1) ---
        dex                     ; X = X - 1; gdy X dojdzie do zera, ustawi się Z
        bne petla               ; X <> 0: powtórz
                                ; tutaj X = 0, pętla wykonała się dokładnie 10 razy
```

Ta pętla wykonuje się dla X od 10 do 1 — **nie obejmuje zera**. Gdy X służy jednocześnie za indeks tablicy, potrzebujemy zwykle zakresu od N−1 do 0. Są na to dwa sposoby.

Sposób 1: `bpl` — pętla trwa, dopóki licznik jest nieujemny:

```
        ldx #9                  ; X = 9: indeks ostatniego z 10 elementów
petla   lda zrodlo,x            ; --- ciało: skopiuj element X (kolejno: 9, 8, ... 0) ---
        sta cel,x               ; zapisz A pod adres cel+X
        dex                     ; X = X - 1; po zerze następuje $FF, czyli N = 1
        bpl petla               ; N = 0 (X >= 0): powtórz
```

Ograniczenie: licznik początkowy nie może przekroczyć 128, bo wartości z ustawionym bitem 7 wyglądają dla `bpl` jak ujemne.

Sposób 2: przesunięty adres — licznik idzie od N do 1, a adres tablicy zmniejszamy o 1:

```
        ldx #200                ; X = 200: liczba elementów (może być do 255)
petla   lda zrodlo-1,x          ; X = 200 sięga do elementu 199, X = 1 do elementu 0
        sta cel-1,x             ; zapisz A pod adres cel-1+X
        dex                     ; X = X - 1
        bne petla               ; powtarzaj do X = 0 (dla X = 0 ciało już się nie wykona)
```

9.1.1 Dokładnie 256 powtórzeń

Licznik ośmiobitowy pozwala na 256 obiegów, jeśli zaczniemy od zera:

```
        ldx #0                  ; X = 0
petla   lda #0                  ; --- ciało (X kolejno: 0, 255, 254, ... 1) ---
        sta bufor,x             ; wyzeruj komórkę bufora
        dex                     ; X = X - 1: z 0 robi się 255, potem 254...
        bne petla               ; powtarzaj, aż X wróci do zera - razem 256 obiegów
```

9.2 Pętla FOR licząca w górę

Gdy kolejność ma znaczenie (np. wypisywanie tekstu), liczymy w górę i porównujemy z wartością końcową:

```
    FOR i := 0 TO 39 DO ciało
```

```
        ldx #0                  ; X = 0: wartość początkowa
petla   lda tekst,x             ; --- ciało (X kolejno: 0, 1, ... 39) ---
        sta ekran,x             ; zapisz A pod adres ekran+X
        inx                     ; X = X + 1
        cpx #40                 ; porównaj z wartością o 1 większą od końcowej
        bne petla               ; X <> 40: powtórz
```

Kosztuje to jeden rozkaz więcej (`cpx`) w każdym obiegu. Przy dokładnie 256 obiegach w górę porównanie znów jest zbędne — `inx` z 255 daje 0 i ustawia Z.

9.2.1 Krok inny niż 1

```
    FOR i := 0 TO 30 STEP 3 DO ciało
```

```
        lda #0                  ; i := 0
        sta i                   ; zapisz A do i
petla   jsr cialo               ; --- ciało ---
        lda i                   ; i := i + 3
        clc                     ; C = 0 (przed dodawaniem)
        adc #3                  ; A = A + 3 + C
        sta i                   ; zapisz A do i
        cmp #31                 ; czy i > 30 ?  (czyli i >= 31)
        bcc petla               ; i < 31: powtórz
```

Przy kroku różnym od 1 kończ pętlę warunkiem „mniejsze” (`bcc`), a nie „różne” (`bne`) — licznik może przeskoczyć wartość graniczną i pętla z `bne` nigdy by się nie skończyła.

9.2.2 Licznik w pamięci

Rejestry X i Y często są potrzebne w ciele pętli. Wtedy licznik trzymamy w komórce pamięci (najlepiej na stronie zerowej):

```
        lda #25                 ; licznik := 25
        sta licznik             ; zapisz A do licznik
petla   jsr cialo               ; --- ciało: może swobodnie używać A, X i Y ---
        dec licznik             ; licznik := licznik - 1 (DEC ustawia znacznik Z)
        bne petla               ; licznik <> 0: powtórz
```

9.3 Pętle z licznikiem 16-bitowym

Rejestr ma 8 bitów, a często trzeba wykonać coś 720, 1024 czy 40 000 razy. Jest kilka technik; każda ma swoje zastosowania.

9.3.1 Licznik 16-bitowy w rejestrach X i Y, liczący w górę

Najczytelniejsza wersja: para rejestrów tworzy jedną liczbę 16-bitową — **X to młodszy bajt, Y to starszy**. Liczbę powtórzeń podajemy jedną stałą, więc zmiana z 720 na 1024 wymaga poprawienia jednej linii.

Plik: petla16.asm

```
; Pętla wykonująca ciało dokładnie ILE razy; licznik (Y:X) rośnie od 0 do ILE-1.
ILE     = 720                   ; liczba powtórzeń - zmień tutaj, np. na 1024

petla16 ldx #0                  ; X = młodszy bajt licznika = 0
        ldy #0                  ; Y = starszy bajt licznika = 0
p16_1   jsr cialo               ; --- ciało pętli (musi zachować X i Y!) ---
        inx                     ; zwiększ młodszy bajt licznika
        bne p16_2               ; nie przekręcił się przez 255 - starszy bajt bez zmian
        iny                     ; przekręcił się (255 -> 0): zwiększ starszy bajt
p16_2   cpx #<ILE               ; czy młodszy bajt licznika = młodszy bajt ILE ?
        bne p16_1               ; nie - powtórz
        cpy #>ILE               ; czy starszy bajt licznika = starszy bajt ILE ?
        bne p16_1               ; nie - powtórz
```

```
        rts                     ; tak - licznik = ILE, koniec pętli
```

Jak to działa: po każdym obiegu licznik rośnie o 1. Pętla kończy się dopiero wtedy, gdy **oba** bajty zgadzają się z wartością końcową. Dla 720 (`$02D0`) oznacza to X = `$D0` i Y = `$02`.

Jeżeli ciało pętli potrzebuje rejestrów, trzeba je zachować — albo przenieść licznik do pamięci.

9.3.2 Licznik 16-bitowy w pamięci, liczący w górę

Ta wersja zostawia wolne wszystkie rejestry, a bieżącą wartość licznika można wykorzystać w ciele pętli (na przykład jako numer sektora dyskietki).

```
        lda #0                  ; licznik := 0
        sta licznik             ; (młodszy bajt)
        sta licznik+1           ; (starszy bajt)
petla   jsr cialo               ; --- ciało pętli: A, X, Y do dowolnego użytku ---
        inc licznik             ; zwiększ młodszy bajt
        bne spr                 ; nie było przejścia przez zero
        inc licznik+1           ; było: zwiększ starszy bajt
spr     lda licznik             ; porównaj licznik z wartością końcową:
        cmp #<ILE               ; młodsze bajty
        bne petla               ; skocz do petla, gdy Z = 0 (różne / nie zero)
        lda licznik+1           ; A = bajt spod adresu licznik+1
        cmp #>ILE               ; starsze bajty
        bne petla               ; skocz do petla, gdy Z = 0 (różne / nie zero)
```

Gdy numeracja ma zaczynać się od 1 (jak sektory dyskietki, numerowane 1...720), wystarczy nadać licznikowi wartość początkową 1, a porównywać z `ILE+1`.

9.3.3 Licznik 16-bitowy liczący w dół

Jeśli wartość licznika w ciele pętli nie jest potrzebna, zliczanie w dół jest najkrótsze. Trzeba tylko poprawnie zmniejszać liczbę 16-bitową: starszy bajt maleje wtedy, gdy młodszy **jest zerem przed** zmniejszeniem.

```
        lda #<ILE               ; licznik := ILE
        sta licznik             ; zapisz A do licznik
        lda #>ILE               ; A = starszy bajt wartości ILE
        sta licznik+1           ; zapisz A pod adres licznik+1
petla   jsr cialo               ; --- ciało pętli ---
        lda licznik             ; czy młodszy bajt jest zerem?
        bne zmn                 ; nie - wystarczy zmniejszyć młodszy
        dec licznik+1           ; tak - "pożycz" ze starszego bajtu
zmn     dec licznik             ; zmniejsz młodszy bajt (z 0 robi się 255)
        lda licznik             ; czy cały licznik doszedł do zera?
        ora licznik+1           ; (zero tylko wtedy, gdy oba bajty są zerami)
        bne petla               ; nie - powtórz
```

9.3.4 Pętle zagnieżdżone: strony i reszta

Gdy ciało pętli przetwarza kolejne bajty pamięci, najszybsza jest konstrukcja dwupoziomowa: zewnętrzna pętla liczy pełne strony (po 256 bajtów), wewnętrzna — bajty na stronie. Tak działają procedury kopiowania i wypełniania z rozdziału 18. Tu szkielet dla 4 stron (1024 bajty):

```
        ldx #4                  ; X = liczba stron do przetworzenia
        ldy #0                  ; Y = indeks bajtu na stronie
petla   lda #0                  ; --- ciało: wyzeruj bajt (wsk),y ---
        sta (wsk),y             ; zapisz A pod adres (wsk) + Y
        iny                     ; następny bajt strony
        bne petla               ; Y <> 0: strona jeszcze nieskończona
        inc wsk+1               ; Y wrócił do 0: przejdź do następnej strony
        dex                     ; jedna strona mniej
        bne petla               ; są jeszcze strony - powtórz
```

9.4 Pętla WHILE

Pętla WHILE sprawdza warunek **przed** każdym obiegiem, więc może nie wykonać się ani razu.

```
   WHILE warunek DO            petla   sprawdź warunek
      ciało                            skocz do "koniec", jeśli FAŁSZYWY
   ENDWHILE                            ciało
                                       jmp petla
                               koniec
```

Przykład — pomiń spacje na początku napisu: `WHILE napis[y] = ' ' DO y := y + 1`

```
        ldy #0                 ; Y = indeks pierwszego znaku
petla   lda napis,y            ; A = bieżący znak
        cmp #' '               ; czy to spacja?
        bne koniec             ; nie - warunek fałszywy, wyjdź z pętli
        iny                    ; --- ciało: przejdź do następnego znaku ---
        jmp petla              ; wróć do sprawdzenia warunku
koniec                         ; tutaj Y wskazuje pierwszy znak niebędący spacją
```

Wariant oszczędniejszy przenosi sprawdzenie warunku na koniec pętli, a pierwsze wejście kieruje od razu do testu. W każdym obiegu wykonuje się wtedy tylko jeden skok:

```
        ldy #0                 ; Y = indeks pierwszego znaku
        jmp test               ; pierwsze wejście: od razu do sprawdzenia warunku
cialo   iny                    ; --- ciało ---
test    lda napis,y            ; --- warunek ---
        cmp #' '               ; porównaj A z: kod ATASCII znaku ' '
        beq cialo              ; spacja: wykonaj ciało i sprawdź ponownie
```

9.5 Pętla REPEAT … UNTIL

Sprawdza warunek **po** obiegu — ciało wykona się co najmniej raz. To najprostsza pętla do zapisania: jeden skok warunkowy na końcu.

```
   REPEAT                      petla   ciało
      ciało                            sprawdź warunek
   UNTIL warunek                       skocz do "petla", jeśli FAŁSZYWY
```

Przykład — czekaj na naciśnięcie klawisza START: `REPEAT UNTIL (CONSOL AND 1) = 0`

```
CONSOL  = $D01F                ; rejestr klawiszy konsoli; bit 0 = START (0 = wciśnięty)

petla   lda CONSOL             ; --- ciało: odczytaj stan klawiszy ---
        and #%00000001         ; zostaw tylko bit klawisza START
        bne petla              ; bit = 1: klawisz nie jest wciśnięty - powtarzaj
```

Przykład — czekaj na początek następnej ramki obrazu (1/50 sekundy). System zwiększa komórkę `$14` w każdym przerwaniu VBI:

```
RTCLOK  = $14                  ; najmłodszy bajt zegara systemowego

        lda RTCLOK             ; A = bieżąca wartość zegara
czekaj  cmp RTCLOK             ; czy zegar nadal ma tę samą wartość?
        beq czekaj             ; tak - czekaj dalej; nie - zaczęła się nowa ramka
```

9.6 Pętla nieskończona, BREAK i CONTINUE

Program gry czy narzędzia zwykle ma **pętlę główną**, która trwa aż do wyjścia:

```
glowna  jsr czytaj_klaw        ; odczytaj klawiaturę; wynik w A
        cmp #27                ; czy naciśnięto ESC?
        beq wyjscie            ; tak - BREAK: opuść pętlę
        cmp #0                 ; czy cokolwiek naciśnięto?
        beq glowna             ; nie - CONTINUE: zacznij następny obieg
```

```
        jsr obsluz              ; obsłuż klawisz
        jmp glowna              ; pętla nieskończona
wyjscie rts                     ; powrót z podprogramu
```

- **BREAK** to po prostu skok poza pętlę.
- **CONTINUE** to skok do miejsca, w którym pętla zwiększa licznik i sprawdza warunek (w pętli nieskończonej — na jej początek).

9.7 Pętle opóźniające

Najprostsze opóźnienie to pusta pętla. Jej czas zależy jednak od szybkości procesora i od tego, ile cykli zabiera układ ANTIC na rysowanie obrazu, więc jest niedokładna:

```
        ldy #0                  ; licznik zewnętrzny: 256 obiegów
        ldx #0                  ; licznik wewnętrzny: 256 obiegów
opoznij dex                     ; pętla wewnętrzna
        bne opoznij             ; skocz do opoznij, gdy Z = 0 (różne / nie zero)
        dey                     ; pętla zewnętrzna
        bne opoznij             ; razem 65 536 obiegów - około 1/5 sekundy
```

Na Atari znacznie lepiej odmierzać czas **zegarem systemowym**, który tyka co ramkę obrazu (w systemie PAL co 1/50 s):

Plik: pauza.asm

```
; Czeka A ramek obrazu (w systemie PAL: A/50 sekundy). Niszczy A i X.
RTCLOK  = $14                   ; najmłodszy bajt zegara, zwiększany przez system co ramkę

pauza   tax                     ; X = liczba ramek do odczekania
pau1    lda RTCLOK              ; A = bieżący stan zegara
pau2    cmp RTCLOK              ; czekaj, aż zegar się zmieni...
        beq pau2                ; ...czyli aż minie jedna ramka
        dex                     ; jedna ramka mniej
        bne pau1                ; odczekaj kolejną
        rts                     ; powrót z podprogramu
```

Wywołanie `lda #50` / `jsr pauza` daje pauzę równą jednej sekundzie.

9.8 Zestawienie

Konstrukcja	Szkielet w asemblerze
`FOR i:=N DOWNTO 1`	`ldx #N` / *ciało* / `dex` / `bne`
`FOR i:=N-1 DOWNTO 0`	`ldx #N-1` / *ciało* / `dex` / `bpl`
`FOR i:=0 TO N-1`	`ldx #0` / *ciało* / `inx` / `cpx #N` / `bne`
`WHILE w DO`	*test* / skok poza pętlę / *ciało* / `jmp`
`REPEAT UNTIL w`	*ciało* / *test* / skok wstecz
pętla nieskończona	*ciało* / `jmp`

Ćwiczenia

1. Napisz pętlę, która wypełnia 40 bajtów od adresu `bufor` wartością `$20`.
2. Zmień procedurę `petla16` tak, by licznik biegł od 1 do 720 (a nie od 0 do 719).
3. Ile razy wykona się ciało pętli: `ldx #0` / *ciało* / `inx` / `cpx #0` / `bne`? A ile, gdy usuniemy `cpx #0`?

Odpowiedzi: (2) `ldx #1`, `ldy #0`, porównania z `#<[ILE+1]` i `#>[ILE+1]`. (3) W obu przypadkach 256 razy — `cpx #0` jest zbędne, bo `inx` sam ustawia znacznik Z.

ROZDZIAŁ 10

Wybór wielowariantowy: CASE … OF

Instrukcja `CASE` wybiera jedną z wielu dróg na podstawie wartości zmiennej. W asemblerze mamy trzy techniki — od najprostszej do najszybszej.

10.1 Technika 1: łańcuch porównań

Najprostszy zapis: kolejne `cmp` i `beq`. Dobry, gdy wariantów jest kilka albo gdy wartości nie tworzą ciągłego zakresu (jak kody klawiszy).

```
   CASE klawisz OF
      'L': lewo
      'P': prawo
      'S': strzal
   ELSE  blad
   END
```

```
        lda klawisz             ; A = wartość sterująca; CMP jej nie zmienia
        cmp #'L'                ; wariant 'L' ?
        beq w_lewo              ; skocz do w_lewo, gdy Z = 1 (równe / zero)
        cmp #'P'                ; wariant 'P' ?
        beq w_prawo             ; skocz do w_prawo, gdy Z = 1 (równe / zero)
        cmp #'S'                ; wariant 'S' ?
        beq w_strzal            ; skocz do w_strzal, gdy Z = 1 (równe / zero)
        jsr blad                ; --- ELSE: żaden wariant nie pasuje ---
        jmp koniec              ; skocz do koniec

w_lewo  jsr lewo                ; --- wariant 'L' ---
        jmp koniec              ; każdy wariant kończy się skokiem za całą instrukcję
w_prawo jsr prawo               ; --- wariant 'P' ---
        jmp koniec              ; skocz do koniec
w_strzal jsr strzal             ; --- wariant 'S' ---
koniec
```

Gdy bloki wariantów są długie, skoki `beq` przestają sięgać i trzeba odwracać warunki:

```
        cmp #'L'                ; wariant 'L' ?
        bne nie_l               ; nie - sprawdzaj dalej
        jmp w_lewo              ; tak - skok daleki
nie_l   cmp #'P'                ; wariant 'P' ?
        bne nie_p               ; skocz do nie_p, gdy Z = 0 (różne / nie zero)
        jmp w_prawo             ; skocz do w_prawo
nie_p
```

10.1.1 Warianty będące zakresami

```
   CASE wiek OF  0..12: dziecko;  13..17: nastolatek;  ELSE dorosly
```

```
        lda wiek                ; A = wiek
        cmp #13                 ; wiek < 13 ?
        bcc dziecko             ; tak: zakres 0..12
        cmp #18                 ; wiek < 18 ?
        bcc nastolatek          ; tak: zakres 13..17 (bo wiemy już, że wiek >= 13)
                                ; tutaj: wiek >= 18 - dorosły
```

Sprawdzając progi po kolei rosnąco, na każdy zakres wystarcza jedno porównanie.

10.2 Technika 2: tablica skoków i `jmp (adres)`

Gdy wartość sterująca to **numer 0, 1, 2...** (pozycja w menu, numer stanu gry, numer polecenia), nie ma sensu jej z niczym porównywać. Traktujemy ją jako indeks do **tablicy adresów** i skaczemy pod wybrany adres. Czas wyboru jest stały, niezależnie od liczby wariantów.

Plik: case_jmp.asm

```
; CASE numer OF 0: proc0; 1: proc1; 2: proc2; 3: proc3 END
; Wejście: A = numer wariantu (0..3). Skacze do odpowiedniej procedury.
ILE_WAR = 4                     ; liczba wariantów
wektor  = $80                   ; komórki $80/$81: adres, pod który skoczymy

wybierz cmp #ILE_WAR            ; zabezpieczenie: czy numer mieści się w tablicy?
        bcs poza                ; numer >= 4: nie ma takiego wariantu
        tax                     ; X = numer wariantu - indeks do tablic
        lda adr_lo,x            ; młodszy bajt adresu procedury numer X
        sta wektor              ; zapisz A do wektor
        lda adr_hi,x            ; starszy bajt adresu procedury numer X
        sta wektor+1            ; zapisz A pod adres wektor+1
        jmp (wektor)            ; skocz pod adres zapisany w komórkach "wektor"
poza    rts                     ; wariant spoza zakresu - nic nie rób

; Tablica adresów podzielona na dwie części: osobno młodsze, osobno starsze bajty.
adr_lo  dta l(proc0,proc1,proc2,proc3)  ; młodsze bajty
adr_hi  dta h(proc0,proc1,proc2,proc3)  ; starsze bajty
```

Każda z procedur `proc0`...`proc3` kończy się rozkazem `rts`. Ponieważ do procedury wchodzimy skokiem `jmp`, a nie `jsr`, jej `rts` wraca od razu do tego, kto wywołał `wybierz` — o to właśnie chodzi.

Dlaczego tablica jest **podzielona** na `adr_lo` i `adr_hi`? Gdyby adresy leżały parami (`dta a(proc0,proc1...)`), numer wariantu trzeba by najpierw pomnożyć przez 2. Przy tablicach rozdzielonych numer jest od razu indeksem, a tablica może mieć do 256 pozycji zamiast 128.

> **Pułapka**
>
> **Zawsze sprawdzaj zakres** przed użyciem wartości jako indeksu tablicy skoków. Numer spoza tablicy pobierze jako „adres” przypadkowe bajty leżące za nią i procesor skoczy w nieznane.

10.3 Technika 3: sztuczka z RTS

Rozkaz `rts` zdejmuje ze stosu adres, dodaje 1 i tam skacze. Nic nie stoi na przeszkodzie, by ten adres odłożyć na stos **samodzielnie**. Dostajemy skok pośredni, który nie potrzebuje żadnej komórki na stronie zerowej:

Plik: case_rts.asm

```
; Wejście: A = numer wariantu (0..3). Skacze do odpowiedniej procedury.
wybierz2 cmp #4                 ; kontrola zakresu
        bcs poza2               ; numer >= 4: nic nie rób
        tax                     ; X = numer wariantu
        lda adr2_hi,x           ; NAJPIERW starszy bajt (adres - 1)...
        pha                     ; ...odłóż na stos
        lda adr2_lo,x           ; potem młodszy bajt (adres - 1)...
        pha                     ; ...odłóż na stos
        rts                     ; RTS zdejmie adres, doda 1 i skoczy do procedury
poza2   rts                     ; powrót z podprogramu

; W tablicy są adresy pomniejszone o 1, bo RTS dodaje jedynkę.
adr2_lo dta l(proc0-1,proc1-1,proc2-1,proc3-1)  ; młodsze bajty
adr2_hi dta h(proc0-1,proc1-1,proc2-1,proc3-1)  ; starsze bajty
```

Kolejność odkładania jest ważna: `jsr` odkłada najpierw starszy bajt, więc my też. I pamiętaj o **odjęciu jedynki** od każdego adresu w tablicy.

Tej samej techniki używa sam system operacyjny Atari: tablice procedur obsługi urządzeń (ekranu, klawiatury, drukarki) zawierają właśnie adresy pomniejszone o 1. Skorzystamy z tego w rozdziale 23.

10.4 Wywołanie podprogramu przez wskaźnik

Procesor nie ma rozkazu `jsr (adres)`. Zastępuje go mały pośrednik — „trampolina”: wywołujemy zwykłym `jsr` jednolinijkową procedurę, która wykonuje skok pośredni. Adres powrotu odłożony przez `jsr` zostaje na stosie, więc `rts` wywołanej procedury wróci tam, gdzie trzeba.

```
        jsr skocz               ; wywołaj procedurę wskazaną przez "wektor"
        lda #0                  ; <- tutaj wróci jej RTS
        rts                     ; powrót z podprogramu

skocz   jmp (wektor)            ; trampolina: skok pośredni, adres powrotu jest już na stosie
```

10.5 CASE jako tablica wartości

Bardzo często `CASE` służy tylko do **zamiany jednej wartości na inną**. Wtedy żadne skoki nie są potrzebne — wystarczy tablica:

```
    CASE poziom OF  0: predkosc := 1;  1: predkosc := 2;  2: predkosc := 4;  3: predkosc := 7  END
```

```
        ldx poziom              ; X = numer poziomu (0..3)
        lda tab_pred,x          ; A = prędkość odpowiadająca temu poziomowi
        sta predkosc            ; zapisz A do predkosc
        rts                     ; powrót z podprogramu

tab_pred dta 1,2,4,7            ; prędkości dla poziomów 0, 1, 2, 3
```

Trzy rozkazy zamiast kilkunastu. Zanim napiszesz łańcuch porównań, zawsze zadaj sobie pytanie: *czy nie wystarczy tablica?* Na 8-bitowym procesorze tablice zastępują nie tylko instrukcje `CASE`, ale też mnożenie, dzielenie, funkcje trygonometryczne i konwersje kodów.

10.6 Wyszukiwanie w tablicy kluczy

Gdy wartości sterujące nie są kolejnymi numerami (np. kody klawiszy), można połączyć obie idee: tablica **kluczy** do przeszukania i równoległa tablica adresów.

Plik: case_klucz.asm

```
; Wejście: A = kod klawisza. Jeśli jest w tablicy kluczy - skacze do procedury.
; Wyjście (gdy klucza nie znaleziono): C = 1.
wektor3 = $80                   ; adres procedury do wykonania

polecenie ldx #ILE_KL-1         ; X = indeks ostatniego klucza
szukaj  cmp klucze,x            ; czy A = klucz numer X ?
        beq znalaz              ; tak
        dex                     ; nie - sprawdź poprzedni klucz
        bpl szukaj              ; powtarzaj dla X = ILE_KL-1 ... 0
        sec                     ; nie znaleziono: C = 1 jako sygnał błędu
        rts                     ; powrót z podprogramu
znalaz  lda kl_lo,x             ; adres procedury odpowiadającej znalezionemu kluczowi
        sta wektor3             ; zapisz A do wektor3
        lda kl_hi,x             ; A = bajt spod adresu kl_hi+X
        sta wektor3+1           ; zapisz A pod adres wektor3+1
        jmp (wektor3)           ; skocz do procedury

klucze  dta c'LPSK'             ; klucze: litery poleceń
ILE_KL  = *-klucze              ; liczba kluczy = bieżący adres minus początek tablicy
kl_lo   dta l(lewo,prawo,strzal,koniec)  ; młodsze bajty
kl_hi   dta h(lewo,prawo,strzal,koniec)  ; starsze bajty
```

Zwróć uwagę na `ILE_KL = *-klucze`: asembler sam oblicza długość tablicy. Dopisanie nowego polecenia wymaga tylko dodania litery i dwóch adresów — kod wyszukiwania pozostaje bez zmian.

10.7 Którą technikę wybrać

Sytuacja	Technika
2–5 wariantów, dowolne wartości	łańcuch `cmp` / `beq`
warianty to zakresy liczb	łańcuch `cmp` / `bcc` z rosnącymi progami
warianty to kolejne numery 0...N	tablica skoków
wynik to tylko wartość	tablica wartości
wiele wariantów o nieregularnych wartościach	tablica kluczy + tablica skoków

ROZDZIAŁ 11

Podprogramy i stos

Podprogram (procedura) to fragment kodu, który można wywołać z wielu miejsc. W asemblerze nie ma deklaracji procedur ani list parametrów — jest tylko etykieta, rozkaz `jsr`, który do niej skacze, i rozkaz `rts`, który wraca. Wszystko inne to **umowa**, którą ustalasz sam i zapisujesz w komentarzu.

11.1 Jak działa JSR i RTS

Prześledźmy dokładnie, co dzieje się na stosie:

```
adres   kod           rozkaz
$2000   20 00 30      jsr $3000
$2003   ...           (następny rozkaz)
...
$3000   ...           (podprogram)
$3005   60            rts
```

```
przed JSR           po JSR                      po RTS
S = $FF             S = $FD                     S = $FF

$01FF: ??           $01FF: $20  starszy bajt    $01FF: $20  (dane zostają,
$01FE: ??           $01FE: $02  młodszy bajt    $01FE: $02   ale są "wolne")
$01FD: ??           $01FD: ??   <- S wskazuje tu

                    PC = $3000                  PC = $2002 + 1 = $2003
```

`jsr` odkłada adres **ostatniego bajtu własnego rozkazu** (`$2002`), a nie adres następnego rozkazu. Dlatego `rts` po zdjęciu adresu dodaje 1. Ten szczegół jest istotny tylko wtedy, gdy sami manipulujemy adresem powrotu — jak w sztuczce z `rts` z poprzedniego rozdziału i w przekazywaniu parametrów za rozkazem `jsr` (poniżej).

Wniosek praktyczny: **w chwili wykonania `rts` wskaźnik stosu musi mieć dokładnie tę samą wartość, co przy wejściu do podprogramu.** Każdy bajt odłożony w podprogramie musi zostać zdjęty.

11.2 Opis podprogramu

Dobrą praktyką jest poprzedzanie każdego podprogramu nagłówkiem opisującym umowę:

```
; ------------------------------------------------------------------------
; max8 - większa z dwóch liczb 8-bitowych bez znaku
; Wejście:  A = pierwsza liczba, X = druga liczba
; Wyjście:  A = większa z nich
; Niszczy:  znaczniki; komórkę "tmp"
; Zachowuje: X, Y
; ------------------------------------------------------------------------
max8    stx tmp                 ; odłóż X do pamięci (CMP nie potrafi porównać A z X wprost)
        cmp tmp                 ; porównaj A z drugą liczbą
        bcs max8_k              ; A >= X: w A jest już większa
        txa                     ; A < X: większa jest druga liczba
max8_k  rts                     ; powrót z podprogramu
```

Cztery pytania, na które nagłówek odpowiada: **co** procedura robi, **co dostaje**, **co zwraca**, **co psuje**. Ostatni punkt jest najważniejszy — większość trudnych błędów w asemblerze to procedura, która po cichu zmieniła rejestr potrzebny wywołującemu.

11.3 Przekazywanie parametrów

11.3.1 Przez rejestry

Najszybszy sposób, wystarczający dla większości procedur. Trzy rejestry mieszczą trzy bajty — na przykład jeden bajt danych i jeden adres 16-bitowy. Powszechna konwencja na Atari: **adres w parze rejestrów, A = młodszy bajt, Y (lub X) = starszy**.

```
        lda #<napis             ; A = młodszy bajt adresu napisu
        ldy #>napis             ; Y = starszy bajt adresu napisu
        jsr drukuj              ; wywołaj: procedura dostaje adres w A/Y
```

11.3.2 Przez zmienne w pamięci

Gdy parametrów jest więcej albo są wielobajtowe, wpisujemy je do ustalonych komórek — najlepiej na stronie zerowej. Tak działają procedury arytmetyczne z dalszych rozdziałów.

```
        lda #<1000              ; mnożna := 1000
        sta mnozna              ; zapisz A do mnozna
        lda #>1000              ; A = starszy bajt wartości 1000
        sta mnozna+1            ; zapisz A pod adres mnozna+1
        lda #25                 ; mnożnik := 25
        sta mnoznik             ; zapisz A do mnoznik
        jsr mnoz                ; wynik znajdzie się w zmiennej "iloczyn"
```

Wadą jest to, że taka procedura **nie może wywołać samej siebie** (każde wywołanie nadpisałoby parametry poprzedniego) i nie powinna być wywoływana jednocześnie z programu głównego i z przerwania.

11.3.3 Przez dane umieszczone za rozkazem JSR

Efektowna technika dla parametrów stałych — najczęściej napisów. Dane leżą w kodzie bezpośrednio za wywołaniem, a procedura odnajduje je dzięki adresowi powrotu ze stosu:

```
        jsr drukuj_tu           ; wypisz tekst leżący zaraz za tym rozkazem
        dta c'Wczytywanie...',$9B  ; tekst w kodach ATASCII
        lda #0                  ; <- wykonanie wraca tutaj, za tekst
```

Plik: drukuj_tu.asm

```
; drukuj_tu - wypisuje tekst umieszczony w kodzie tuż za wywołaniem JSR.
; Tekst musi kończyć się bajtem $9B (koniec linii), który także jest wypisywany.
; Niszczy: A, X, Y. Wymaga procedury "putchar" (wypisz znak z A).
dt_wsk  = $80                   ; wskaźnik na bieżący znak (2 bajty na stronie zerowej)

drukuj_tu pla                   ; zdejmij młodszy bajt adresu powrotu...
        sta dt_wsk              ; ...to adres ostatniego bajtu rozkazu JSR,
        pla                     ; czyli bajtu tuż PRZED naszym tekstem
        sta dt_wsk+1            ; zdejmij i zapamiętaj starszy bajt
dt_nast inc dt_wsk              ; przesuń wskaźnik na następny znak
        bne dt_1                ; (zwiększanie 16-bitowe: gdy młodszy bajt
        inc dt_wsk+1            ; przekręci się na 0, zwiększ starszy)
dt_1    ldy #0                  ; Y = 0: odczyt dokładnie spod wskaźnika
        lda (dt_wsk),y          ; A = bieżący znak tekstu
        pha                     ; zapamiętaj znak (putchar może zniszczyć A)
        jsr putchar             ; wypisz znak
        pla                     ; odtwórz znak
        cmp #$9B                ; czy to był znak końca linii?
        bne dt_nast             ; nie - wypisz następny znak
        lda dt_wsk+1            ; tak: wskaźnik pokazuje teraz OSTATNI bajt tekstu;
        pha                     ; odłóż go na stos jako nowy adres powrotu
        lda dt_wsk              ; (najpierw starszy bajt, potem młodszy -
        pha                     ; tak samo, jak robi to JSR)
        rts                     ; RTS doda 1 i wróci do rozkazu ZA tekstem
```

Procedura zdejmuje adres powrotu, używa go jako wskaźnika na dane, a na koniec odkłada adres **poprawiony** — wskazujący ostatni bajt danych. `rts` doda swoją jedynkę i wykonanie trafi dokładnie za tekst.

11.3.4 Przez stos

Języki wysokiego poziomu przekazują parametry przez stos. W 6502 jest to niewygodne (stos ma 256 bajtów, a dostęp wymaga `tsx` i adresowania `$0100,x`), więc w asemblerze pisanym ręcznie prawie się tego nie stosuje. Stos służy za to powszechnie do **przechowywania** wartości.

11.4 Zwracanie wyników

- **W rejestrach** — wynik 8-bitowy w A, 16-bitowy zwykle w A (młodszy) i X lub Y (starszy).
- **W zmiennych** — dla wyników wielobajtowych.
- **W znacznikach** — szczególnie przydatne dla informacji „udało się / błąd”:

```
; Wejście: A = kod znaku. Wyjście: C = 0, gdy znak jest cyfrą; C = 1, gdy nie.
czy_cyfra cmp #'0'              ; znak < '0' ?
        bcc cc_nie              ; tak - to nie cyfra
        cmp #'9'+1              ; znak <= '9' ? CMP od razu daje C = 0 dla cyfr
        rts                     ; C = 0: cyfra, C = 1: znak powyżej '9'
cc_nie  sec                     ; ustaw C = 1: to nie cyfra
        rts                     ; powrót z podprogramu

; --- użycie ---
        lda znak                ; A = znak
        jsr czy_cyfra           ; zbadaj znak
        bcs to_nie_cyfra        ; C = 1: nie jest cyfrą
```

Po powrocie znaczniki mają stan nadany przez ostatni rozkaz podprogramu — `rts` ich nie zmienia. Konwencję „C = 1 oznacza błąd” stosuje wiele procedur systemowych.

11.5 Zachowywanie rejestrów

Kto ma dbać o rejestry: wywołujący czy wywoływany? Obie szkoły są dobre, byle konsekwentnie:

Wywoływany zachowuje — procedura sama odkłada rejestry, których używa. Wywołania są proste, procedura nieco wolniejsza:

```
proc    pha                     ; zachowaj A
        txa                     ; A = X
        pha                     ; zachowaj X
        tya                     ; A = Y
        pha                     ; zachowaj Y
                                ; ... właściwa praca ...
        pla                     ; zdejmij A ze stosu
        tay                     ; odtwórz Y
        pla                     ; zdejmij A ze stosu
        tax                     ; odtwórz X
        pla                     ; odtwórz A
        rts                     ; powrót z podprogramu
```

Wywołujący zachowuje — procedury niczego nie gwarantują, a kto potrzebuje rejestru, ten go sam odkłada przed `jsr`. Kod jest szybszy, ale wymaga uwagi.

Zamiast stosu można użyć zwykłych komórek pamięci — to szybsze (3 cykle zamiast 3+4 i bez przekładania przez A):

```
        stx zapas_x             ; zachowaj X w zmiennej
        jsr niszczy_x           ; procedura zmieniająca X
        ldx zapas_x             ; odtwórz X
```

> **Uwaga**
>
> **Procedury systemowe Atari zwykle niszczą wszystkie rejestry.** Po wywołaniu CIO, SIO czy procedur zmiennoprzecinkowych nie zakładaj niczego o zawartości A, X i Y poza tym, co gwarantuje dokumentacja (np. CIO zwraca status w rejestrze Y).

11.6 Podprogramy zagnieżdżone i głębokość stosu

Podprogram może wywołać inny podprogram, ten kolejny i tak dalej. Każdy poziom to 2 bajty stosu, do tego dochodzą odłożone rejestry. Stos ma 256 bajtów, z czego część zajmuje system operacyjny i DOS, a każde przerwanie potrzebuje kilku–kilkunastu bajtów. W praktyce kilkanaście poziomów zagnieżdżenia jest całkowicie bezpieczne; kłopoty zaczynają się przy rekurencji.

11.6.1 Optymalizacja: `jsr` + `rts` = `jmp`

Jeżeli ostatnią czynnością podprogramu jest wywołanie innego podprogramu, parę `jsr/rts` można zastąpić jednym `jmp`. Wywołana procedura wróci swoim `rts` bezpośrednio do naszego wywołującego:

```
; zamiast:                      ; można napisać:
;       jsr putchar             ;       jmp putchar
;       rts
nowa_linia lda #$9B             ; A = kod końca linii
        jmp putchar             ; wypisz go i wróć (RTS z putchar wraca do naszego wywołującego)
```

Oszczędność: 1 bajt, 9 cykli i 2 bajty stosu.

11.7 Rekurencja

Podprogram wywołujący samego siebie musi przechowywać swoje dane lokalne na stosie, bo zwykłe zmienne zostałyby nadpisane przez kolejne wywołanie. Klasyczny przykład — wypisanie liczby dziesiętnie „od końca” — pojawi się w rozdziale 16 w wersji bez rekurencji, bo na 6502 rekurencja rzadko jest najlepszym rozwiązaniem. Tutaj krótka ilustracja samej zasady: suma liczb $1 + 2 + \ldots + n$.

Plik: suma_rek.asm

```
; suma_rek - rekurencyjnie oblicza 1 + 2 + ... + n (wynik musi być <= 255)
; Wejście: A = n.  Wyjście: A = suma.  Niszczy: komórkę "sr_tmp".
sr_tmp  = $80                   ; komórka pomocnicza

suma_rek cmp #0                 ; warunek zakończenia rekurencji: n = 0 ?
        beq sr_kon              ; tak: suma(0) = 0, A już zawiera zero
        pha                     ; odłóż n na stos - to nasza "zmienna lokalna"
        sec                     ; C = 1 (przed odejmowaniem)
        sbc #1                  ; A = n - 1
        jsr suma_rek            ; wywołanie rekurencyjne: A = suma(n-1)
        sta sr_tmp              ; zapamiętaj wynik wywołania
        pla                     ; zdejmij ze stosu nasze n
        clc                     ; C = 0 (przed dodawaniem)
        adc sr_tmp              ; A = n + suma(n-1)
sr_kon  rts                     ; powrót z podprogramu
```

Dla $n = 5$ stos w najgłębszym miejscu zawiera 5 odłożonych wartości i 6 adresów powrotu — razem 17 bajtów.

11.8 Biblioteka własnych procedur

W miarę nauki zbierzesz zestaw sprawdzonych procedur: wypisywanie znaku i liczby, mnożenie, dzielenie, kopiowanie pamięci, odczyt klawiatury. Trzymaj je w osobnych plikach i dołączaj dyrektywą `icl`:

```
        org $2000               ; adres asemblacji: $2000
start   jsr init                ; program główny
        rts                     ; powrót z podprogramu

        icl 'lib/print.asm'     ; procedury wypisywania
        icl 'lib/mul16.asm'     ; mnożenie 16-bitowe
        icl 'lib/memcpy.asm'    ; kopiowanie bloków pamięci
```

Wszystkie procedury z dalszych rozdziałów są napisane tak, by nadawały się do takiej biblioteki: mają nagłówek z opisem, własne nazwy zmiennych i nie zależą od reszty programu. O tym, jak uporządkować bibliotekę dyrek-

tywą `.proc` — z etykietami lokalnymi i automatycznym pomijaniem nieużywanych procedur — mówi rozdział 30.

Ćwiczenia

1. Napisz podprogram `zamien`, który zamienia miejscami zawartość rejestrów A i X.
2. Co się stanie, jeśli podprogram wykona `pha`, ale przed `rts` nie wykona `pla`?
3. Napisz podprogram zwracający C = 0, gdy znak w A jest literą (wielką lub małą), i C = 1 w przeciwnym razie.

Odpowiedzi: (1) `sta tmp` / `txa` / `ldx tmp` / `rts`. (2) `rts` zdejmie jako adres powrotu odłożony bajt i połowę prawdziwego adresu — procesor skoczy w przypadkowe miejsce.

Cześć III

Arytmetyka

ROZDZIAŁ 12

Dodawanie i odejmowanie: 8, 16 i 32 bity

Procesor 6502 potrafi dodać i odjąć dwa bajty. Wszystko ponad to — liczby 16-bitowe, 32-bitowe i dłuższe — składamy z działań na pojedynczych bajtach, połączonych znacznikiem przeniesienia C. Zasada jest dokładnie ta sama, co w szkolnym dodawaniu pisemnym: zaczynamy od najmłodszej pozycji i przenosimy „jeden dalej".

Zapamiętaj

Dwie żelazne reguły arytmetyki 6502:

1. **Przed dodawaniem `clc`, przed odejmowaniem `sec`** — ale tylko przed **pierwszym** (najmłodszym) bajtem.
2. **Między bajtami liczby wielobajtowej nie wolno zmienić znacznika C.** Rozkazy `lda`, `sta`, `inx`, `dex`, `iny`, `dey` go nie zmieniają. Rozkazy `cmp`, `cpx`, `cpy`, `asl`, `lsr`, `rol`, `ror` — zmieniają.

12.1 Liczby 8-bitowe

12.1.1 Dodawanie

```
; wynik := a + b
        lda a                   ; A = pierwszy składnik
        clc                     ; C = 0: nic nie doliczamy
        adc b                   ; A = a + b
        sta wynik               ; zapisz sumę (8 młodszych bitów)
                                ; C = 1 oznacza, że prawdziwa suma przekroczyła 255
```

Dodawanie stałej do zmiennej — odpowiednik `x := x + 5`:

```
        lda x                   ; A = x
        clc                     ; C = 0 (przed dodawaniem)
        adc #5                  ; A = x + 5
        sta x                   ; x := wynik
```

Jeżeli zwiększamy o 1 lub 2, szybciej i krócej jest użyć `inc x` (raz lub dwa razy) — pamiętając, że `inc` nie ustawia znacznika C.

12.1.2 Odejmowanie

```
; wynik := a - b
        lda a                   ; A = odjemna
        sec                     ; C = 1: brak pożyczki
        sbc b                   ; A = a - b
        sta wynik               ; zapisz różnicę
                                ; C = 0 oznacza, że b > a (wynik "zawinął się" poniżej zera)
```

Przykład: $5 - 7$ daje w bajcie 254 (`$FE`) i $C = 0$. Jako liczba ze znakiem `$FE` to -2 — poprawnie. Jako liczba bez znaku wynik jest błędny, o czym informuje właśnie $C = 0$.

12.1.3 Kiedy można pominąć `clc` i `sec`

Jeżeli **wiadomo na pewno**, jaki jest stan znacznika C, rozkaz ustawiający go jest zbędny. Typowa sytuacja — tuż po skoku warunkowym:

```
        cmp #10                 ; porównanie ustawia C
        bcc mniejsze            ; skok, gdy C = 0...
        sbc #10                 ; ...więc tutaj C = 1 na pewno: SEC nie jest potrzebny
mniejsze
```

Taką optymalizację zawsze opisz w komentarzu. Dopóki nie masz wprawy — pisz `clc` i `sec` zawsze; kosztują tylko 2 cykle.

12.2 Liczby 16-bitowe

Przyjmujemy, że zmienna 16-bitowa zajmuje dwie kolejne komórki: pod adresem `a` leży młodszy bajt, pod `a+1` — starszy.

12.2.1 Dodawanie 16 + 16 bitów

Plik: add16.asm

```
; add16 - dodawanie liczb 16-bitowych: w16 := a16 + b16
; Wyjście: C = 1, gdy suma przekroczyła 65535. Niszczy: A.
a16     = $80                   ; pierwszy składnik (2 bajty: $80 młodszy, $81 starszy)
b16     = $82                   ; drugi składnik (2 bajty)
w16     = $84                   ; wynik (2 bajty)

add16   clc                     ; C = 0 - tylko raz, przed najmłodszym bajtem
        lda a16                 ; młodszy bajt pierwszej liczby
        adc b16                 ; plus młodszy bajt drugiej; C = przeniesienie z tej pozycji
        sta w16                 ; zapisz młodszy bajt wyniku (STA nie zmienia C)
        lda a16+1               ; starszy bajt pierwszej liczby (LDA nie zmienia C)
        adc b16+1               ; plus starszy bajt drugiej PLUS przeniesienie z młodszych
        sta w16+1               ; zapisz starszy bajt wyniku
        rts                     ; C = przeniesienie z całego dodawania 16-bitowego
```

Prześledźmy `$12F0` + `$0320`:

```
   młodsze bajty:  $F0 + $20 + C(0) = $110  ->  zapisz $10, C = 1
   starsze bajty:  $12 + $03 + C(1) = $16   ->  zapisz $16, C = 0
   wynik: $1610
```

12.2.2 Dodawanie stałej 16-bitowej

```
; adres := adres + 40  (np. przejście do następnego wiersza ekranu)
        lda adres               ; młodszy bajt
        clc                     ; C = 0 (przed dodawaniem)
        adc #<40                ; plus młodszy bajt stałej (czyli 40)
        sta adres               ; zapisz A do adres
        lda adres+1             ; starszy bajt
        adc #>40                ; plus starszy bajt stałej (czyli 0) plus przeniesienie
        sta adres+1             ; zapisz A pod adres adres+1
```

Gdy dodawana wartość mieści się w jednym bajcie, starszy bajt wystarczy zwiększyć warunkowo. Jest to krótsze i szybsze:

```
; adres := adres + 40 - wersja oszczędna
        lda adres               ; młodszy bajt
        clc                     ; C = 0 (przed dodawaniem)
        adc #40                 ; dodaj 40
        sta adres               ; zapisz A do adres
        bcc bez_przen           ; nie było przeniesienia - starszy bajt bez zmian
        inc adres+1             ; było: starszy bajt + 1
bez_przen
```

12.2.3 Dodawanie bajtu z akumulatora do liczby 16-bitowej

Bardzo częsta operacja: przesunięcie wskaźnika o wartość obliczoną w programie.

```
; wsk := wsk + A
        clc                     ; C = 0 (przed dodawaniem)
        adc wsk                 ; A = A + młodszy bajt wskaźnika
        sta wsk                 ; zapisz A do wsk
        bcc gotowe              ; brak przeniesienia - koniec
        inc wsk+1               ; przeniesienie: zwiększ starszy bajt
gotowe
```

12.2.4 Zwiększanie i zmniejszanie o 1

```
; licznik := licznik + 1
        inc licznik             ; zwiększ młodszy bajt
        bne juz                 ; wynik <> 0: nie było przejścia 255 -> 0
        inc licznik+1           ; było: zwiększ starszy bajt
juz
```

```
; licznik := licznik - 1
        lda licznik             ; czy młodszy bajt jest zerem?
        bne tylko_ml            ; nie - starszy bajt zostaje
        dec licznik+1           ; tak: zmniejszenie "pożyczy" jedynkę ze starszego bajtu
tylko_ml dec licznik            ; zmniejsz młodszy bajt (z 0 robi się 255)
```

Zmniejszanie jest niesymetryczne względem zwiększania, bo `dec` nie potrafi zasygnalizować przejścia 0 → 255 znacznikiem Z. Dlatego zero sprawdzamy **przed** zmniejszeniem.

12.2.5 Odejmowanie 16 – 16 bitów

Plik: sub16.asm

```
; sub16 - odejmowanie liczb 16-bitowych: w16 := a16 - b16
; Wyjście: C = 0, gdy b16 > a16 (wynik ujemny). Niszczy: A.
a16     = $80                   ; odjemna (2 bajty)
b16     = $82                   ; odjemnik (2 bajty)
w16     = $84                   ; wynik (2 bajty)

sub16   sec                     ; C = 1 - tylko raz, przed najmłodszym bajtem
        lda a16                 ; młodszy bajt odjemnej
        sbc b16                 ; minus młodszy bajt odjemnika; C = 0, jeśli trzeba pożyczyć
        sta w16                 ; zapisz młodszy bajt wyniku
        lda a16+1               ; starszy bajt odjemnej
        sbc b16+1               ; minus starszy bajt odjemnika minus ewentualna pożyczka
        sta w16+1               ; zapisz starszy bajt wyniku
        rts                     ; C = 1: wynik nieujemny; C = 0: a16 < b16
```

Odejmowanie wartości jednobajtowej od liczby 16-bitowej — wersja oszczędna:

```
; adres := adres - 40
        lda adres               ; młodszy bajt
        sec                     ; C = 1 (przed odejmowaniem)
        sbc #40                 ; odejmij 40
        sta adres               ; zapisz A do adres
        bcs bez_poz             ; C = 1: nie było pożyczki
        dec adres+1             ; C = 0: pożyczka - starszy bajt - 1
bez_poz
```

12.3 Liczby 32-bitowe

Liczba 32-bitowa to cztery bajty i wartości do 4 294 967 295. Na Atari przydaje się rzadziej, niż mogłoby się wydawać — do punktacji w grach, długości plików, numerów sektorów dużych dysków, wyników mnożenia 16×16. Zasada pozostaje ta sama, łańcuch jest tylko dłuższy.

12.3.1 Dodawanie 32 + 32 bity — wersja rozwinięta

Plik: add32.asm

```
; add32 - dodawanie liczb 32-bitowych: w32 := a32 + b32
; Wyjście: C = 1, gdy suma nie zmieściła się w 32 bitach. Niszczy: A.
a32     = $80                   ; pierwszy składnik (4 bajty: $80 najmłodszy ... $83 najstarszy)
b32     = $84                   ; drugi składnik (4 bajty)
w32     = $88                   ; wynik (4 bajty)

add32   clc                     ; C = 0 przed najmłodszym bajtem
        lda a32                 ; bajt 0 (najmłodszy)
        adc b32                 ; A = A + b32 + C
        sta w32                 ; zapisz A do w32
        lda a32+1               ; bajt 1 - z przeniesieniem z bajtu 0
        adc b32+1               ; A = A + bajt spod adresu b32+1 + C
        sta w32+1               ; zapisz A pod adres w32+1
        lda a32+2               ; bajt 2 - z przeniesieniem z bajtu 1
        adc b32+2               ; A = A + bajt spod adresu b32+2 + C
        sta w32+2               ; zapisz A pod adres w32+2
        lda a32+3               ; bajt 3 (najstarszy) - z przeniesieniem z bajtu 2
        adc b32+3               ; A = A + bajt spod adresu b32+3 + C
        sta w32+3               ; zapisz A pod adres w32+3
        rts                     ; C = przeniesienie z całego dodawania
```

12.3.2 Odejmowanie 32 – 32 bity

Plik: sub32.asm

```
; sub32 - odejmowanie liczb 32-bitowych: w32 := a32 - b32
; Wyjście: C = 0, gdy b32 > a32. Niszczy: A.
a32     = $80                   ; odjemna (4 bajty)
b32     = $84                   ; odjemnik (4 bajty)
w32     = $88                   ; wynik (4 bajty)

sub32   sec                     ; C = 1 przed najmłodszym bajtem
        lda a32                 ; bajt 0 (najmłodszy)
        sbc b32                 ; A = A - b32 - pożyczka (gdy C = 0)
        sta w32                 ; zapisz A do w32
        lda a32+1               ; bajt 1 - z pożyczką z bajtu 0
        sbc b32+1               ; A = A - bajt spod adresu b32+1 - pożyczka (gdy C = 0)
        sta w32+1               ; zapisz A pod adres w32+1
        lda a32+2               ; bajt 2
        sbc b32+2               ; A = A - bajt spod adresu b32+2 - pożyczka (gdy C = 0)
        sta w32+2               ; zapisz A pod adres w32+2
        lda a32+3               ; bajt 3 (najstarszy)
        sbc b32+3               ; A = A - bajt spod adresu b32+3 - pożyczka (gdy C = 0)
        sta w32+3               ; zapisz A pod adres w32+3
        rts                     ; powrót z podprogramu
```

12.3.3 Wersja w pętli — dla liczb dowolnej długości

Powtarzające się trójki rozkazów aż proszą się o pętlę. Trzeba tylko dopilnować, żeby **sterowanie pętlą nie zepsuło znacznika C**. Rozkazy `inx` i `dey` go nie ruszają; `cpx` — tak, więc nie możemy porównywać indeksu z wartością końcową. Rozwiązanie: osobny licznik malejący do zera.

Plik: addn.asm

```
; addn - dodawanie liczb o długości N bajtów: an := an + bn
; Wyjście: C = przeniesienie. Niszczy: A, X, Y.
N       = 4                     ; długość liczb w bajtach (4 = 32 bity; można zmienić np. na 8)
an      = $80                   ; pierwsza liczba i jednocześnie wynik (N bajtów)
bn      = $88                   ; druga liczba (N bajtów)

addn    ldx #0                  ; X = indeks bajtu: zaczynamy od najmłodszego
        ldy #N                  ; Y = licznik bajtów pozostałych do dodania
        clc                     ; C = 0 przed pierwszym dodawaniem
addn_1  lda an,x                ; bajt numer X pierwszej liczby
        adc bn,x                ; plus bajt numer X drugiej, plus przeniesienie
```

```
        sta an,x                ; zapisz bajt wyniku
        inx                     ; następny bajt (INX nie zmienia C!)
        dey                     ; licznik - 1 (DEY nie zmienia C!)
        bne addn_1              ; powtarzaj, aż licznik dojdzie do zera
        rts                     ; powrót z podprogramu
```

Wersja w pętli jest krótsza, ale około dwóch razy wolniejsza od rozwiniętej. Dla 16 bitów zawsze pisz wersję rozwiniętą; dla 32 — zależnie od tego, czy bardziej brakuje pamięci, czy czasu.

12.3.4 Zwiększanie liczby 32-bitowej o 1

Plik: inc32.asm

```
; inc32 - zwiększenie liczby 32-bitowej o 1. Nie zmienia A, X, Y.
l32     = $80                   ; liczba (4 bajty)

inc32   inc l32                 ; zwiększ bajt 0
        bne inc32_k             ; nie przekręcił się - koniec
        inc l32+1               ; przekręcił się: zwiększ bajt 1
        bne inc32_k             ; skocz do inc32_k, gdy Z = 0 (różne / nie zero)
        inc l32+2               ; i tak dalej...
        bne inc32_k             ; skocz do inc32_k, gdy Z = 0 (różne / nie zero)
        inc l32+3               ; zwiększ o 1 bajt spod adresu l32+3
inc32_k rts                     ; powrót z podprogramu
```

W 255 przypadkach na 256 wykona się tylko pierwszy `inc` i jeden skok — to bardzo szybki licznik.

12.3.5 Dodawanie liczby 8- lub 16-bitowej do 32-bitowej

Krótszą liczbę uzupełniamy w myśli zerami: do starszych bajtów dodajemy zero **wraz z przeniesieniem**.

Plik: add32_16.asm

```
; add32_16 - dodanie liczby 16-bitowej do 32-bitowej: l32 := l32 + d16. Niszczy: A.
l32     = $80                   ; liczba 32-bitowa (4 bajty)
d16     = $84                   ; dodawana liczba 16-bitowa (2 bajty)

add32_16 clc                    ; C = 0 (przed dodawaniem)
        lda l32                 ; bajt 0
        adc d16                 ; A = A + d16 + C
        sta l32                 ; zapisz A do l32
        lda l32+1               ; bajt 1
        adc d16+1               ; A = A + bajt spod adresu d16+1 + C
        sta l32+1               ; zapisz A pod adres l32+1
        bcc a3216_k             ; nie ma przeniesienia - starsze bajty bez zmian
        inc l32+2               ; jest: zwiększ bajt 2
        bne a3216_k             ; nie przekręcił się - koniec
        inc l32+3               ; przekręcił się: zwiększ bajt 3
a3216_k rts                     ; powrót z podprogramu
```

12.4 Zmiana znaku (negacja)

Liczbę przeciwną w kodzie U2 najprościej otrzymać, odejmując ją od zera. Dla liczb wielobajtowych to zwykłe odejmowanie z przeniesieniem:

Plik: neg16.asm

```
; neg16 - zmiana znaku liczby 16-bitowej: l16 := 0 - l16. Niszczy: A.
l16     = $80                   ; liczba (2 bajty)

neg16   sec                     ; C = 1 przed odejmowaniem
        lda #0                  ; 0 - młodszy bajt
        sbc l16                 ; A = A - l16 - pożyczka (gdy C = 0)
        sta l16                 ; zapisz A do l16
        lda #0                  ; 0 - starszy bajt - pożyczka
        sbc l16+1               ; A = A - bajt spod adresu l16+1 - pożyczka (gdy C = 0)
        sta l16+1               ; zapisz A pod adres l16+1
```

```
        rts                     ; powrót z podprogramu
```

Dla jednego bajtu w akumulatorze używa się zwykle postaci „odwróć bity i dodaj 1”: `eor #$FF` / `clc` / `adc #1`. Negacja liczby 32-bitowej to cztery pary `lda #0` / `sbc`.

12.5 Liczby ze znakiem

Dodawanie i odejmowanie liczb w kodzie U2 wykonuje się **tymi samymi rozkazami** co dla liczb bez znaku. Różnice są dwie.

Wykrywanie przepełnienia. Zamiast znacznika C badamy V — po ostatnim (najstarszym) bajcie:

```
        clc                     ; C = 0 (przed dodawaniem)
        lda a16                 ; młodsze bajty - zwyczajnie
        adc b16                 ; A = A + b16 + C
        sta w16                 ; zapisz A do w16
        lda a16+1               ; starsze bajty
        adc b16+1               ; A = A + bajt spod adresu b16+1 + C
        sta w16+1               ; zapisz A pod adres w16+1
        bvs nadmiar             ; V = 1: wynik poza zakresem -32768..+32767
```

Rozszerzanie znaku. Gdy do liczby 16-bitowej dodajemy 8-bitową liczbę ze znakiem (np. prędkość −3...+3 do współrzędnej), nie wolno uzupełnić jej zerami: −1 to `$FF`, a w 16 bitach musi stać się `$FFFF`. Starszy bajt uzupełnienia to `$00` dla liczb nieujemnych i `$FF` dla ujemnych:

Plik: add16s8.asm

```
; add16s8 - dodanie 8-bitowej liczby ZE ZNAKIEM do liczby 16-bitowej: poz := poz + delta
; Niszczy: A, X.
poz     = $80                   ; pozycja (2 bajty)
delta   = $82                   ; przesunięcie -128..+127 (1 bajt)

add16s8 ldx #0                  ; X = starszy bajt rozszerzenia: załóż $00 (liczba nieujemna)
        lda delta               ; A = przesunięcie; N = jego znak
        bpl a16s_1              ; nieujemne: rozszerzenie to $00
        dex                     ; ujemne: X = $FF
a16s_1  clc                     ; C = 0 (przed dodawaniem)
        adc poz                 ; młodszy bajt: delta + poz
        sta poz                 ; zapisz A do poz
        txa                     ; A = rozszerzenie znaku ($00 lub $FF)
        adc poz+1               ; starszy bajt: rozszerzenie + poz + przeniesienie
        sta poz+1               ; zapisz A pod adres poz+1
        rts                     ; powrót z podprogramu
```

12.6 Najczęstsze błędy

Objaw	Przyczyna
wynik bywa większy o 1	brak `clc` przed dodawaniem
wynik bywa mniejszy o 1	brak `sec` przed odejmowaniem
starszy bajt błędny	między bajtami użyto rozkazu zmieniającego C (np. `cmp`, `asl`) albo wstawiono drugie `clc`
po `inc` zły skok `bcc/bcs`	`inc` i `dec` nie zmieniają znacznika C
wynik zupełnie bez sensu	włączony tryb dziesiętny (zapomniane `cld`)
liczba ujemna „skacze” o 256	brak rozszerzenia znaku przy dodawaniu liczby 8-bitowej do 16-bitowej

Ćwiczenia

1. Napisz fragment obliczający `w16 := a16 + b16 + c16` (trzy liczby 16-bitowe).
2. Napisz `dec32` — zmniejszenie liczby 32-bitowej o 1. Wskazówka: najprościej odjąć stałą 1 czterema parami `lda/sbc/sta`.

3. Dlaczego w procedurze `addn` nie można zakończyć pętli parą `cpx #N` / `bne`?

ROZDZIAŁ 13

Przesunięcia, obroty i operacje na bitach

Cztery rozkazy — `asl`, `lsr`, `rol`, `ror` — zastępują w 6502 mnożenie i dzielenie przez potęgi dwójki, służą do budowy pełnego mnożenia i dzielenia, do rozbierania bajtów na bity i składania ich z powrotem. Ten rozdział zbiera najważniejsze techniki.

13.1 Mnożenie przez 2, 4, 8…

Każde przesunięcie w lewo podwaja liczbę:

```
        lda x                   ; A = x
        asl @                   ; A = x * 2
        asl @                   ; A = x * 4
        asl @                   ; A = x * 8
        sta wynik               ; poprawne, dopóki wynik mieści się w bajcie (x <= 31)
```

Po każdym `asl` znacznik C zawiera bit, który „wypadł”. Jeśli wynik może przekroczyć 255, trzeba liczyć na 16 bitach:

```
; w16 := x * 8 dla dowolnego x (0..255); wynik 16-bitowy
        lda x                   ; młodszy bajt wyniku := x
        sta w16                 ; zapisz A do w16
        lda #0                  ; starszy bajt wyniku := 0
        sta w16+1               ; zapisz A pod adres w16+1
        asl w16                 ; przesuń młodszy bajt; bit 7 trafia do C
        rol w16+1               ; wsuń C do starszego bajtu           (razem: * 2)
        asl w16                 ; przesuń w16 o 1 bit w lewo (bit 7 -> C)
        rol w16+1               ;                                     (razem: * 4)
        asl w16                 ; przesuń w16 o 1 bit w lewo (bit 7 -> C)
        rol w16+1               ;                                     (razem: * 8)
```

Para `asl młodszy` / `rol starszy` to **przesunięcie 16-bitowe w lewo**. Dla 32 bitów łańcuch się wydłuża: `asl` na bajcie 0 i `rol` na bajtach 1, 2, 3.

13.2 Dzielenie przez 2, 4, 8…

Przesunięcie w prawo dzieli liczbę bez znaku przez 2, odrzucając resztę:

```
        lda x                   ; A = x
        lsr @                   ; A = x / 2
        lsr @                   ; A = x / 4
        lsr @                   ; A = x / 8
        sta iloraz              ; zapisz A do iloraz
```

Resztę z dzielenia przez potęgę dwójki daje maska AND — to po prostu młodsze bity:

```
        lda x                   ; A = x
        and #%00000111          ; A = x mod 8 (trzy najmłodsze bity)
        sta reszta              ; zapisz A do reszta
```

Przesunięcie 16-bitowe w prawo zaczyna się od **starszego** bajtu:

```
        lsr w16+1               ; przesuń starszy bajt; jego bit 0 trafia do C
        ror w16                 ; wsuń C do bitu 7 młodszego bajtu      (razem: / 2)
```

Dla 32 bitów: `lsr` na bajcie 3, potem `ror` na bajtach 2, 1, 0.

13.2.1 Dzielenie liczb ze znakiem

`lsr` wstawia zero do bitu 7, więc psuje liczby ujemne (z −4 zrobiłby +126). Potrzebne jest **przesunięcie arytmetyczne**, które powiela bit znaku. 6502 nie ma takiego rozkazu, ale wystarczą dwa:

```
        lda x                   ; A = liczba ze znakiem
        cmp #$80                ; C = 1, gdy A >= $80, czyli gdy bit 7 (znak) jest ustawiony
        ror @                   ; wsuń kopię bitu znaku do bitu 7: A = x / 2 z zachowaniem znaku
```

Rozkaz `cmp #$80` to zgrabny sposób skopiowania bitu 7 do znacznika C bez zmiany akumulatora.

13.3 Przesunięcie o zmienną liczbę bitów

Gdy liczba przesunięć nie jest znana z góry, używamy pętli:

Plik: shl16.asm

```
; shl16 - przesunięcie liczby 16-bitowej w lewo o X bitów (X = 0..16). Niszczy: X.
w16     = $80                   ; liczba (2 bajty)

shl16   cpx #0                  ; przesunięcie o zero bitów?
        beq shl16_k             ; tak - nic nie rób
shl16_1 asl w16                 ; przesuń o jeden bit
        rol w16+1               ; obróć bajt spod adresu w16+1 w lewo przez C
        dex                     ; jedno przesunięcie mniej
        bne shl16_1             ; skocz do shl16_1, gdy Z = 0 (różne / nie zero)
shl16_k rts                     ; powrót z podprogramu
```

Przesunięcie o 8 bitów to po prostu przepisanie bajtu: `lda w16` / `sta w16+1` / `lda #0` / `sta w16`. Warto o tym pamiętać — mnożenie przez 256 jest w 6502 „za darmo".

13.4 Obrót bajtu bez udziału przeniesienia

`rol` i `ror` obracają 9 bitów (bajt i znacznik C). Czasem potrzebny jest obrót samych 8 bitów — bit wychodzący z jednej strony ma wejść z drugiej:

```
; obrót A w lewo o 1 bit (bit 7 przechodzi na pozycję bitu 0)
        cmp #$80                ; C = bit 7 akumulatora
        rol @                   ; przesuń w lewo, wsuwając dawny bit 7 na pozycję 0

; obrót A w prawo o 1 bit (bit 0 przechodzi na pozycję bitu 7)
        lsr @                   ; bit 0 -> C, bit 7 := 0
        bcc obr_k               ; bit 0 był zerem - gotowe
        ora #$80                ; bit 0 był jedynką - ustaw bit 7
obr_k
```

13.5 Mnożenie przez stałą

Mnożenie przez dowolną stałą rozkłada się na przesunięcia i dodawania. Wystarczy zapisać stałą jako sumę potęg dwójki:

- $x \cdot 3 = x \cdot 2 + x$
- $x \cdot 5 = x \cdot 4 + x$
- $x \cdot 10 = x \cdot 8 + x \cdot 2$
- $x \cdot 40 = x \cdot 32 + x \cdot 8$

Plik: mul10.asm

```
; mul10 - mnożenie przez 10: A := A * 10. Poprawne dla A = 0..25. Niszczy: komórkę "m10_tmp".
m10_tmp = $80                   ; komórka pomocnicza

mul10   asl @                   ; A = x * 2
        sta m10_tmp             ; zapamiętaj x * 2
```

```
        asl @                   ; A = x * 4
        asl @                   ; A = x * 8
        clc                     ; C = 0 (przed dodawaniem)
        adc m10_tmp             ; A = x * 8 + x * 2 = x * 10
        rts                     ; powrót z podprogramu
```

Mnożenie przez 40 jest na Atari szczególnie ważne: ekran tekstowy ma 40 znaków w wierszu, więc adres wiersza numer y to początek ekranu plus $y \cdot 40$.

Plik: mul40.asm

```
; mul40 - mnożenie przez 40 z wynikiem 16-bitowym: w40 := A * 40 (A = 0..255)
; Niszczy: A.
w40     = $80                   ; wynik (2 bajty)
t40     = $82                   ; zmienna pomocnicza (2 bajty)

mul40   sta w40                 ; w40 := x (młodszy bajt)
        lda #0                  ; A = 0
        sta w40+1               ; starszy bajt := 0
        asl w40                 ; w40 := x * 2
        rol w40+1               ; obróć bajt spod adresu w40+1 w lewo przez C
        asl w40                 ; w40 := x * 4
        rol w40+1               ; obróć bajt spod adresu w40+1 w lewo przez C
        asl w40                 ; w40 := x * 8
        rol w40+1               ; obróć bajt spod adresu w40+1 w lewo przez C
        lda w40                 ; t40 := x * 8 (zapamiętaj wynik pośredni)
        sta t40                 ; zapisz A do t40
        lda w40+1               ; A = bajt spod adresu w40+1
        sta t40+1               ; zapisz A pod adres t40+1
        asl w40                 ; w40 := x * 16
        rol w40+1               ; obróć bajt spod adresu w40+1 w lewo przez C
        asl w40                 ; w40 := x * 32
        rol w40+1               ; obróć bajt spod adresu w40+1 w lewo przez C
        clc                     ; w40 := x * 32 + x * 8 = x * 40
        lda w40                 ; A = w40
        adc t40                 ; A = A + t40 + C
        sta w40                 ; zapisz A do w40
        lda w40+1               ; A = bajt spod adresu w40+1
        adc t40+1               ; A = A + bajt spod adresu t40+1 + C
        sta w40+1               ; zapisz A pod adres w40+1
        rts                     ; powrót z podprogramu
```

Wskazówka

Jeśli mnożnik jest mały i znany (np. numer wiersza 0–23), jeszcze szybsza jest **tablica gotowych wyników**: `lda wiersz_lo,y` i `lda wiersz_hi,y`. Zajmuje 48 bajtów, a mnożenie sprowadza do dwóch rozkazów. Zobaczysz ją w rozdziale 22.

13.6 Ustawianie, kasowanie i badanie bitów

13.6.1 Bit wybrany stałą

```
        lda flagi               ; --- ustaw bit 3 ---
        ora #%00001000          ; A = A OR %00001000
        sta flagi               ; zapisz A do flagi

        lda flagi               ; --- skasuj bit 3 ---
        and #%11110111          ; maska: zero tylko na pozycji 3
        sta flagi               ; zapisz A do flagi

        lda flagi               ; --- odwróć bit 3 ---
        eor #%00001000          ; A = A XOR %00001000
        sta flagi               ; zapisz A do flagi

        lda flagi               ; --- zbadaj bit 3 ---
        and #%00001000          ; A = A AND %00001000
        bne bit_ustawiony       ; wynik <> 0: bit 3 jest jedynką
```

MADS pomaga pisać maski czytelniej. Zamiast liczyć zera, można zapisać `#1<<3` (jedynka przesunięta o 3 pozycje) i `#[1<<3]^$FF` (jej negacja).

13.6.2 Bit o numerze w rejestrze

Gdy numer bitu jest obliczany w programie, potrzebna jest **tablica masek**:

Plik: bity.asm

```
; Operacje na bicie o numerze X (0..7) w komórce "flagi". Niszczą: A.
flagi   = $80                   ; bajt, którego bity zmieniamy

ustaw_bit lda flagi             ; A = dotychczasowa wartość
        ora maski,x             ; ustaw bit wskazany przez maskę numer X
        sta flagi               ; zapisz A do flagi
        rts                     ; powrót z podprogramu

kasuj_bit lda maski,x           ; A = maska z jedynką na wybranej pozycji
        eor #$FF                ; odwróć: zero na wybranej pozycji, reszta jedynki
        and flagi               ; skasuj wybrany bit, pozostałe bez zmian
        sta flagi               ; zapisz A do flagi
        rts                     ; powrót z podprogramu

badaj_bit lda flagi             ; A = wartość
        and maski,x             ; zostaw tylko badany bit; Z = 1, gdy jest zerem
        rts                     ; wywołujący używa BEQ (bit = 0) lub BNE (bit = 1)

maski   dta %00000001,%00000010,%00000100,%00001000  ; dane: wzór bitów
        dta %00010000,%00100000,%01000000,%10000000  ; dane: wzór bitów
```

Rozszerzenie na wielobajtową **mapę bitową** jest proste: numer bitu dzielimy przez 8 (trzy `lsr`), co daje numer bajtu, a reszta (`and #7`) to numer bitu w bajcie. Tak właśnie DOS zapisuje w sektorze VTOC, które sektory dyskietki są wolne.

13.7 Półbajty

Rozdzielanie bajtu na dwie cyfry szesnastkowe i ponowne składanie to operacja potrzebna przy wyświetlaniu liczb, kodzie BCD i kolorach Atari:

```
; --- rozdziel bajt na półbajty ---
        lda bajt                ; A = %ssssmmmm
        and #$0F                ; A = %0000mmmm
        sta mlodszy             ; młodszy półbajt (0..15)
        lda bajt                ; A = %ssssmmmm
        lsr @                   ; cztery przesunięcia w prawo
        lsr @                   ; przesuń A o 1 bit w prawo (bit 0 -> C)
        lsr @                   ; przesuń A o 1 bit w prawo (bit 0 -> C)
        lsr @                   ; A = %0000ssss
        sta starszy             ; starszy półbajt (0..15)

; --- złóż bajt z półbajtów ---
        lda starszy             ; A = %0000ssss
        asl @                   ; cztery przesunięcia w lewo
        asl @                   ; przesuń A o 1 bit w lewo (bit 7 -> C)
        asl @                   ; przesuń A o 1 bit w lewo (bit 7 -> C)
        asl @                   ; A = %ssss0000
        ora mlodszy             ; A = %ssssmmmm
        sta bajt                ; zapisz A do bajt
```

13.8 Zliczanie i przeglądanie bitów

Typowy schemat „zrób coś dla każdego bitu”: wysuwamy bity po kolei do znacznika C i badamy je skokiem `bcc/bcs`.

Plik: licz_bity.asm

```
; licz_bity - zlicza bity równe 1 w akumulatorze.
; Wejście: A = bajt.  Wyjście: X = liczba jedynek (0..8), A = 0.
licz_bity ldx #0                ; X = licznik jedynek
lb_1    asl @                   ; wysuń najstarszy bit do C; reszta przesuwa się w lewo
        bcc lb_2                ; bit był zerem - nie licz
        inx                     ; bit był jedynką - policz
lb_2    bne lb_1                ; czy w A zostały jeszcze jakieś jedynki? tak - powtarzaj
        rts                     ; A = 0: wszystkie bity przejrzane
```

Zwróć uwagę na elegancki warunek zakończenia: pętla nie liczy do ośmiu, tylko kończy się, gdy w akumulatorze nie ma już żadnej jedynki. Skok `bne lb_1` bada znacznik Z ustawiony przez ostatni rozkaz, który go zmienił. Jeżeli `inx` został pominięty, jest to `asl` — czyli $Z = 1$ dokładnie wtedy, gdy akumulator się wyzerował. Jeżeli `inx` się wykonał, $Z = 0$ (bo $X \geq 1$) i pętla wykona jeszcze jeden obieg; `asl` na samych zerach da wtedy $Z = 1$ i pętla się skończy.

13.9 Bajt jako zestaw flag

Osiem zmiennych logicznych mieści się w jednym bajcie. Rozkaz `bit` pozwala zbadać dwie z nich (bity 7 i 6) bez użycia akumulatora, a przesunięcia — dowolną:

```
F_DZWIEK = %10000000            ; bit 7: dźwięk włączony
F_PAUZA = %01000000             ; bit 6: gra wstrzymana
F_DEMO  = %00000001             ; bit 0: tryb demonstracyjny

        bit stan                ; N = bit 7 (dźwięk), V = bit 6 (pauza)
        bvs wstrzymana          ; pauza włączona
        bpl bez_dzwieku         ; dźwięk wyłączony

        lda stan                ; włącz pauzę
        ora #F_PAUZA            ; A = A OR F_PAUZA
        sta stan                ; zapisz A do stan

        lda stan                ; wyłącz pauzę
        and #F_PAUZA^$FF        ; maska odwrócona przez asembler
        sta stan                ; zapisz A do stan
```

Ćwiczenia

1. Napisz fragment obliczający `A := A * 6` (dla $A \leq 42$).
2. Napisz przesunięcie liczby 32-bitowej `l32` o jeden bit w prawo.
3. Jak jednym rozkazem sprawdzić, czy liczba w A jest podzielna przez 4?

Odpowiedzi: (1) `asl @` / `sta tmp` / `asl @` / `clc` / `adc tmp` ($x \cdot 4 + x \cdot 2$). (2) `lsr l32+3` / `ror l32+2` / `ror l32+1` / `ror l32`. (3) `and #%00000011` — wynik zero oznacza podzielność.

ROZDZIAŁ 14

Mnożenie

Procesor 6502 nie ma rozkazu mnożenia. Trzeba je zaprogramować — i jest to znakomite ćwiczenie, bo algorytm jest tym samym mnożeniem pisemnym, którego uczono nas w szkole, tylko w systemie dwójkowym, gdzie staje się zaskakująco prosty.

14.1 Najprostszy sposób: wielokrotne dodawanie

$5 \cdot 3$ to $5 + 5 + 5$. Dla małych mnożników to w zupełności wystarcza:

Plik: mul_dod.asm

```
; mul_dod - mnożenie przez wielokrotne dodawanie: A := A * X (wynik musi być <= 255)
; Niszczy: X, komórkę "md_tmp".
md_tmp  = $80                   ; komórka pomocnicza: mnożna

mul_dod sta md_tmp              ; zapamiętaj mnożną
        lda #0                  ; A = 0: tu będzie rosła suma
        cpx #0                  ; mnożnik = 0 ?
        beq md_k                ; tak: wynik 0
        clc                     ; C = 0; przy wyniku <= 255 żadne dodawanie go nie ustawi
md_1    adc md_tmp              ; A = A + mnożna
        dex                     ; jedno dodawanie mniej
        bne md_1                ; powtarzaj X razy
md_k    rts                     ; powrót z podprogramu
```

Czas działania rośnie wraz z mnożnikiem: mnożenie przez 200 to 200 obiegów pętli. Potrzebujemy czegoś lepszego.

14.2 Algorytm „przesuń i dodaj”

Przypomnijmy mnożenie pisemne w systemie dziesiętnym:

```
   123
 x 205
 -----
   615      123 * 5
  000       123 * 0, przesunięte o jedną pozycję
 246        123 * 2, przesunięte o dwie pozycje
 -----
 25215
```

Dla każdej cyfry mnożnika mnożymy przez nią mnożną i dodajemy wynik przesunięty o odpowiednią liczbę pozycji. W systemie dwójkowym cyfry to tylko 0 i 1, więc „mnożenie przez cyfrę” znika: **albo dodajemy przesuniętą mnożną (cyfra 1), albo nie dodajemy nic (cyfra 0)**.

```
   1101      (13)  mnożna
 x 0101      (5)   mnożnik
 ------
   1101      bit 0 mnożnika = 1: dodaj mnożną
  0000       bit 1 mnożnika = 0: nic nie dodawaj
 1101        bit 2 mnożnika = 1: dodaj mnożną przesuniętą o 2 pozycje
0000         bit 3 mnożnika = 0: nic nie dodawaj
--------
1000001      (65)
```

Algorytm dla mnożnika n-bitowego:

1. Wyzeruj iloczyn.

2. Powtórz n razy:
   - wysuń najmłodszy bit mnożnika (przesunięcie w prawo, bit trafia do C),
   - jeśli był jedynką — dodaj mnożną do iloczynu,
   - przesuń mnożną o jeden bit w lewo (następny bit ma dwukrotnie większą wagę).

Iloczyn dwóch liczb n-bitowych może mieć do 2n bitów: 8 × 8 daje 16 bitów, 16 × 16 daje 32.

14.3 Mnożenie 8 × 8 = 16 bitów — wersja czytelna

Ta wersja wprost realizuje opisany algorytm. Mnożną rozszerzamy do 16 bitów, bo będzie przesuwana w lewo.

Plik: mul8.asm

```
; mul8 - mnożenie liczb 8-bitowych bez znaku: iloczyn (16 bitów) := mnozna * mnoznik
; Wejście: mnozna, mnoznik (po 1 bajcie). Wyjście: iloczyn (2 bajty).
; Niszczy: A, X oraz zmienne mnozna i mnoznik.
mnozna  = $80                   ; mnożna: 2 bajty (drugi służy do przesuwania w lewo)
mnoznik = $82                   ; mnożnik: 1 bajt
iloczyn = $83                   ; wynik: 2 bajty

mul8    lda #0                  ; A = 0
        sta iloczyn             ; iloczyn := 0
        sta iloczyn+1           ; zapisz A pod adres iloczyn+1
        sta mnozna+1            ; rozszerz mnożną do 16 bitów (starszy bajt := 0)
        ldx #8                  ; X = 8: tyle bitów ma mnożnik
mul8_1  lsr mnoznik             ; wysuń najmłodszy bit mnożnika do znacznika C
        bcc mul8_2              ; bit = 0: nie dodawaj
        clc                     ; bit = 1: iloczyn := iloczyn + mnożna (dodawanie 16-bitowe)
        lda iloczyn             ; A = iloczyn
        adc mnozna              ; A = A + mnozna + C
        sta iloczyn             ; zapisz A do iloczyn
        lda iloczyn+1           ; A = bajt spod adresu iloczyn+1
        adc mnozna+1            ; A = A + bajt spod adresu mnozna+1 + C
        sta iloczyn+1           ; zapisz A pod adres iloczyn+1
mul8_2  asl mnozna              ; mnożna := mnożna * 2 (waga następnego bitu jest 2x większa)
        rol mnozna+1            ; obróć bajt spod adresu mnozna+1 w lewo przez C
        dex                     ; następny bit mnożnika
        bne mul8_1              ; powtórz 8 razy
        rts                     ; powrót z podprogramu
```

Prześledźmy 13 · 5 (mnożnik = `%00000101`):

Obieg	Bit mnożnika	Mnożna (przesunięta)	Iloczyn po obiegu
1	1	13	13
2	0	26	13
3	1	52	65
4–8	0	104, 208…	65

Wskazówka

Pętlę można zakończyć wcześniej: gdy po `lsr mnoznik` mnożnik stał się zerem, dalsze obiegi niczego już nie dodadzą. Dla małych mnożników daje to duże przyspieszenie. Wystarczy zamienić `dex / bne mul8_1` na badanie, czy mnożnik jest jeszcze niezerowy (`lda mnoznik / bne mul8_1`) — i wtedy licznik X jest w ogóle zbędny.

14.4 Mnożenie 8 × 8 = 16 bitów — wersja szybka

Doświadczeni programiści 6502 stosują sprytniejszy układ: zamiast przesuwać mnożną w lewo, **przesuwają iloczyn w prawo**. Dodawanie jest wtedy zawsze 8-bitowe, a starsza część iloczynu mieszka w akumulatorze. Co więcej, bity mnożnika „wychodzą" z jego komórki w tym samym tempie, w jakim „wchodzą" do niej młodsze bity iloczynu — więc mnożnik i młodszy bajt wyniku **dzielą tę samą komórkę**.

Plik: mul8f.asm

```
; mul8f - szybkie mnożenie 8 x 8 = 16 bitów bez znaku
; Wejście:  fmnozna, fmnoznik (po 1 bajcie)
; Wyjście:  A = starszy bajt iloczynu, fmnoznik = młodszy bajt iloczynu
; Niszczy:  X. Zachowuje: fmnozna.
fmnozna = $80                   ; mnożna
fmnoznik = $81                  ; mnożnik, a po zakończeniu młodszy bajt iloczynu

mul8f   lda #0                  ; A = starsza część iloczynu, na początku 0
        ldx #8                  ; X = licznik bitów
        lsr fmnoznik            ; wysuń pierwszy (najmłodszy) bit mnożnika do C
mul8f_1 bcc mul8f_2             ; bit = 0: nie dodawaj (C = 0 zostaje do przesunięcia)
        clc                     ; C = 0 (przed dodawaniem)
        adc fmnozna             ; bit = 1: dodaj mnożną do starszej części; C = przeniesienie
mul8f_2 ror @                   ; przesuń starszą część w prawo; z lewej wchodzi C (9. bit sumy),
                                ; a najmłodszy bit wypada do C
        ror fmnoznik            ; ten bit wchodzi z lewej do młodszej części iloczynu,
                                ; a z prawej wypada do C NASTĘPNY bit mnożnika
        dex                     ; następny bit
        bne mul8f_1             ; powtórz 8 razy
        rts                     ; powrót z podprogramu
```

Ta wersja jest mniej więcej dwa razy szybsza od czytelnej i o połowę krótsza. Warto poświęcić chwilę na zrozumienie, jak para `ror` / `ror` jednocześnie przesuwa 16-bitowy iloczyn i podaje kolejny bit mnożnika — ten sam pomysł wraca w mnożeniu 16- i 32-bitowym.

14.5 Mnożenie 16 × 16 = 32 bity

Ten sam algorytm w wersji szybkiej, rozszerzony na dwa bajty. Starsza połowa iloczynu (bajty 2 i 3) przyjmuje dodawania, a cały 32-bitowy iloczyn jest przesuwany w prawo.

Plik: mul16.asm

```
; mul16 - mnożenie liczb 16-bitowych bez znaku: il32 (32 bity) := mna16 * mnk16
; Wejście: mna16, mnk16 (po 2 bajty). Wyjście: il32 (4 bajty, od najmłodszego).
; Niszczy: A, X oraz zmienną mnk16. Zachowuje: mna16.
mna16   = $80                   ; mnożna (2 bajty)
mnk16   = $82                   ; mnożnik (2 bajty) - zostaje wyzerowany
il32    = $84                   ; iloczyn (4 bajty)

mul16   lda #0                  ; A = 0
        sta il32+2              ; wyzeruj starszą połowę iloczynu
        sta il32+3              ; (młodsza połowa zapełni się sama podczas przesuwania)
        ldx #16                 ; X = 16: tyle bitów ma mnożnik
mul16_1 lsr mnk16+1             ; przesuń 16-bitowy mnożnik w prawo:
        ror mnk16               ; najmłodszy bit trafia do C
        bcc mul16_2             ; bit = 0: nie dodawaj
        lda il32+2              ; bit = 1: dodaj mnożną do STARSZEJ połowy iloczynu
        clc                     ; C = 0 (przed dodawaniem)
        adc mna16               ; A = A + mna16 + C
        sta il32+2              ; zapisz A pod adres il32+2
        lda il32+3              ; A = bajt spod adresu il32+3
        adc mna16+1             ; A = najstarszy bajt iloczynu; C = przeniesienie z dodawania
mul16_2 ror @                   ; przesuń cały iloczyn w prawo o 1 bit, od najstarszego bajtu
        sta il32+3              ; (jest w A); z lewej wchodzi przeniesienie C
        ror il32+2              ; obróć bajt spod adresu il32+2 w prawo przez C
        ror il32+1              ; obróć bajt spod adresu il32+1 w prawo przez C
        ror il32                ; obróć il32 w prawo przez C
        dex                     ; następny bit mnożnika
        bne mul16_1             ; powtórz 16 razy
        rts                     ; powrót z podprogramu
```

Subtelny punkt: gdy dodawanie jest pomijane (`bcc mul16_2`), w akumulatorze musi znajdować się bieżąca wartość `il32+3`. I tak jest — pozostała tam po `sta il32+3` z poprzedniego obiegu (a w pierwszym obiegu jest to zero z początku procedury). Rozkazy `ror` na pamięci oraz `dex` nie zmieniają akumulatora.

Jeżeli potrzebny jest tylko wynik 16-bitowy (bo wiadomo, że się zmieści), wystarczy odczytać `il32` i `il32+1`.

14.6 Mnożenie 32 × 32 = 64 bity

Dla porządku — pełne mnożenie 32-bitowe. Algorytm bez zmian; dla przejrzystości wszystkie bajty są obsługiwane jawnie.

Plik: mul32.asm

```
; mul32 - mnożenie liczb 32-bitowych bez znaku: il64 (64 bity) := mna32 * mnk32
; Wejście: mna32, mnk32 (po 4 bajty). Wyjście: il64 (8 bajtów, od najmłodszego).
; Niszczy: A, Y oraz zmienną mnk32. Zachowuje: mna32, X.
mna32   = $80                   ; mnożna (4 bajty)
mnk32   = $84                   ; mnożnik (4 bajty) - zostaje wyzerowany
il64    = $88                   ; iloczyn (8 bajtów)

mul32   lda #0                  ; A = 0
        sta il64+4              ; wyzeruj starszą połowę iloczynu
        sta il64+5              ; zapisz A pod adres il64+5
        sta il64+6              ; zapisz A pod adres il64+6
        sta il64+7              ; zapisz A pod adres il64+7
        ldy #32                 ; Y = 32: tyle bitów ma mnożnik
mul32_1 lsr mnk32+3             ; przesuń 32-bitowy mnożnik w prawo;
        ror mnk32+2             ; obróć bajt spod adresu mnk32+2 w prawo przez C
        ror mnk32+1             ; obróć bajt spod adresu mnk32+1 w prawo przez C
        ror mnk32               ; najmłodszy bit trafia do C
        bcc mul32_2             ; bit = 0: nie dodawaj (C = 0 wejdzie do iloczynu)
        clc                     ; bit = 1: dodaj mnożną do starszej połowy iloczynu
        lda il64+4              ; A = bajt spod adresu il64+4
        adc mna32               ; A = A + mna32 + C
        sta il64+4              ; zapisz A pod adres il64+4
        lda il64+5              ; A = bajt spod adresu il64+5
        adc mna32+1             ; A = A + bajt spod adresu mna32+1 + C
        sta il64+5              ; zapisz A pod adres il64+5
        lda il64+6              ; A = bajt spod adresu il64+6
        adc mna32+2             ; A = A + bajt spod adresu mna32+2 + C
        sta il64+6              ; zapisz A pod adres il64+6
        lda il64+7              ; A = bajt spod adresu il64+7
        adc mna32+3             ; A = A + bajt spod adresu mna32+3 + C
        sta il64+7              ; C = przeniesienie z dodawania (65. bit)
mul32_2 ror il64+7              ; przesuń cały 64-bitowy iloczyn w prawo; z lewej wchodzi C
        ror il64+6              ; obróć bajt spod adresu il64+6 w prawo przez C
        ror il64+5              ; obróć bajt spod adresu il64+5 w prawo przez C
        ror il64+4              ; obróć bajt spod adresu il64+4 w prawo przez C
        ror il64+3              ; obróć bajt spod adresu il64+3 w prawo przez C
        ror il64+2              ; obróć bajt spod adresu il64+2 w prawo przez C
        ror il64+1              ; obróć bajt spod adresu il64+1 w prawo przez C
        ror il64                ; obróć il64 w prawo przez C
        dey                     ; następny bit mnożnika
        bne mul32_1             ; powtórz 32 razy
        rts                     ; powrót z podprogramu
```

To mnożenie trwa kilka tysięcy cykli — rząd dwóch–trzech milisekund. Na 8-bitowym procesorze 32-bitowe mnożenie w pętli gry to luksus, na który rzadko można sobie pozwolić.

14.7 Mnożenie liczb ze znakiem

Najprostsza i najpewniejsza metoda: **pomnóż wartości bezwzględne, a na końcu nadaj wynikowi właściwy znak**. Wynik jest ujemny, gdy czynniki mają różne znaki — co sprawdza jedno `eor` na ich bitach znaku.

Plik: mul16s.asm

```
; mul16s - mnożenie liczb 16-bitowych ZE ZNAKIEM: il32 := mna16 * mnk16
; Wymaga procedury mul16 (i jej zmiennych). Niszczy: A, X, mna16, mnk16, komórkę "znak".
znak    = $88                   ; bit 7 = znak wyniku

mul16s  lda mna16+1             ; starszy bajt mnożnej (zawiera bit znaku)
        eor mnk16+1             ; XOR ze starszym bajtem mnożnika:
        sta znak                ; bit 7 = 1, gdy znaki czynników są RÓŻNE (wynik ujemny)

        lda mna16+1             ; czy mnożna jest ujemna?
```

```
        bpl ms_1                ; nie
        sec                     ; tak: mna16 := 0 - mna16 (wartość bezwzględna)
        lda #0                  ; A = 0
        sbc mna16               ; A = A - mna16 - pożyczka (gdy C = 0)
        sta mna16               ; zapisz A do mna16
        lda #0                  ; A = 0
        sbc mna16+1             ; A = A - bajt spod adresu mna16+1 - pożyczka (gdy C = 0)
        sta mna16+1             ; zapisz A pod adres mna16+1
ms_1    lda mnk16+1             ; czy mnożnik jest ujemny?
        bpl ms_2                ; nie
        sec                     ; tak: mnk16 := 0 - mnk16
        lda #0                  ; A = 0
        sbc mnk16               ; A = A - mnk16 - pożyczka (gdy C = 0)
        sta mnk16               ; zapisz A do mnk16
        lda #0                  ; A = 0
        sbc mnk16+1             ; A = A - bajt spod adresu mnk16+1 - pożyczka (gdy C = 0)
        sta mnk16+1             ; zapisz A pod adres mnk16+1
ms_2    jsr mul16               ; pomnóż wartości bezwzględne (bez znaku)

        bit znak                ; czy wynik ma być ujemny?
        bpl ms_3                ; nie - gotowe
        sec                     ; tak: il32 := 0 - il32 (negacja 32-bitowa)
        lda #0                  ; A = 0
        sbc il32                ; A = A - il32 - pożyczka (gdy C = 0)
        sta il32                ; zapisz A do il32
        lda #0                  ; A = 0
        sbc il32+1              ; A = A - bajt spod adresu il32+1 - pożyczka (gdy C = 0)
        sta il32+1              ; zapisz A pod adres il32+1
        lda #0                  ; A = 0
        sbc il32+2              ; A = A - bajt spod adresu il32+2 - pożyczka (gdy C = 0)
        sta il32+2              ; zapisz A pod adres il32+2
        lda #0                  ; A = 0
        sbc il32+3              ; A = A - bajt spod adresu il32+3 - pożyczka (gdy C = 0)
        sta il32+3              ; zapisz A pod adres il32+3
ms_3    rts                     ; powrót z podprogramu
```

14.8 Szybkie mnożenie tablicowe

W grach, demach i wszędzie tam, gdzie mnożenie wykonuje się tysiące razy na sekundę, stosuje się sztuczkę opartą na wzorze skróconego mnożenia:

```
a * b  =  [ (a + b)^2 - (a - b)^2 ] / 4
```

Wystarczy mieć w pamięci tablicę wartości $n^2 / 4$ dla n od 0 do 510. Mnożenie sprowadza się wtedy do jednego dodawania, jednego odejmowania, dwóch odczytów z tablicy i jeszcze jednego odejmowania — około 40–50 cykli zamiast 150–200. Ceną jest około 2 KB pamięci na tablice (po dwa bajty na pozycję, podzielone na części młodszą i starszą) oraz jednorazowe ich wygenerowanie przy starcie programu. To klasyczna wymiana **pamięci na czas** — jedna z podstawowych technik programowania maszyn 8-bitowych.

14.9 Którą procedurę wybrać

Potrzeba	Rozwiązanie
mnożenie przez stałą (2, 4, 10, 40…)	przesunięcia i dodawania — rozdział 13
mały mnożnik z ograniczonego zakresu	tablica gotowych wyników
8 × 8 bitów	`mul8f`
16 × 16 bitów	`mul16`
liczby ze znakiem	`mul16s`
tysiące mnożeń na sekundę	tablica kwadratów

Ćwiczenia

1. Prześledź na kartce działanie `mul8f` dla $3 \cdot 2$. Zapisuj po każdym obiegu zawartość A, `fmnoznik` i znacznika C.
2. Zmodyfikuj `mul8` tak, by kończyła pętlę, gdy mnożnik stanie się zerem.
3. Oblicz adres komórki ekranu dla kolumny x i wiersza y: `adres := SAVMSC + y * 40 + x`, używając procedury `mul40` z rozdziału 13 i dodawania 16-bitowego z rozdziału 12.

ROZDZIAŁ 15

Dzielenie

Dzielenie jest lustrzanym odbiciem mnożenia: tam przesuwaliśmy i dodawaliśmy, tu **przesuwamy i odejmujemy**. To także dokładna kopia dzielenia pisemnego — w wersji dwójkowej.

Przypomnijmy nazwy: **dzielna** : **dzielnik** = **iloraz** i **reszta**. Zawsze zachodzi: dzielna = iloraz · dzielnik + reszta, przy czym reszta < dzielnik.

15.1 Najprostszy sposób: wielokrotne odejmowanie

Iloraz mówi, ile razy dzielnik „mieści się” w dzielnej. Można to po prostu policzyć:

Plik: div_odejm.asm

```
; div_odejm - dzielenie przez wielokrotne odejmowanie
; Wejście:  A = dzielna, komórka do_dzielnik = dzielnik (różny od zera!)
; Wyjście:  X = iloraz, A = reszta
do_dzielnik = $80               ; dzielnik

div_odejm ldx #0                ; X = iloraz: licznik wykonanych odejmowań
do_1    cmp do_dzielnik         ; czy dzielnik jeszcze mieści się w tym, co zostało?
        bcc do_k                ; nie (A < dzielnik): koniec, w A jest reszta
        sbc do_dzielnik         ; tak: odejmij go (C = 1 po CMP, więc SEC jest zbędny)
        inx                     ; iloraz + 1
        bne do_1                ; powtarzaj (skok zawsze - iloraz nie przekroczy 255)
do_k    rts                     ; powrót z podprogramu
```

Ta metoda jest dobra, gdy iloraz jest mały — na przykład przy rozbijaniu liczby 0–99 na dziesiątki i jedności. Dla dużych ilorazów jest powolna: 250 : 1 to 250 obiegów.

15.2 Algorytm „przesuń i odejmij”

Dzielenie pisemne w systemie dwójkowym, na przykładzie 13 : 3:

```
   dzielna 1101 (13), dzielnik 11 (3)

   bierzemy bity dzielnej od lewej, dopisując je do "bieżącej reszty":

   reszta   1       1 <  11  ->  bit ilorazu 0
   reszta   11     11 >= 11  ->  bit ilorazu 1, reszta := 11 - 11 = 0
   reszta   00      0 <  11  ->  bit ilorazu 0
   reszta   001     1 <  11  ->  bit ilorazu 0

   iloraz 0100 (4), reszta 1          sprawdzenie: 4 * 3 + 1 = 13
```

W systemie dwójkowym dzielnik mieści się w bieżącej reszcie **albo zero razy, albo jeden raz** — nie trzeba zgadywać cyfry ilorazu, wystarczy jedno porównanie. Algorytm dla dzielnej n-bitowej:

1. Wyzeruj resztę.
2. Powtórz n razy:
   - przesuń dzielną w lewo; jej najstarszy bit wsuń z prawej strony do reszty,
   - jeśli reszta ≥ dzielnik: odejmij dzielnik od reszty i zapisz 1 jako kolejny bit ilorazu; w przeciwnym razie zapisz 0.

Sprytne spostrzeżenie: przesuwając dzielną w lewo, zwalniamy jej najmłodszy bit — i właśnie tam wpisujemy kolejny bit ilorazu. Po n obiegach w komórce dzielnej znajduje się gotowy **iloraz**.

Uwaga

Dzielenie przez zero. Żadna z poniższych procedur nie sprawdza, czy dzielnik jest zerem. Dzielenie przez zero nie zawiesi programu (pętla ma stałą liczbę obiegów), ale da bezwartościowy wynik. Jeśli dzielnik może być zerem, sprawdź to **przed** wywołaniem.

15.3 Dzielenie 8 : 8 bitów

Plik: div8.asm

```
; div8 - dzielenie liczb 8-bitowych bez znaku
; Wejście:  dzielna, dzielnik (po 1 bajcie; dzielnik różny od zera)
; Wyjście:  dzielna = iloraz, A = reszta (zapisana też w zmiennej reszta)
; Niszczy:  X. Zachowuje: dzielnik.
dzielna = $80                   ; dzielna, a po zakończeniu iloraz
dzielnik = $81                  ; dzielnik
reszta  = $82                   ; reszta

div8    lda #0                  ; A = bieżąca reszta, na początku 0
        ldx #8                  ; X = 8: tyle bitów ma dzielna
div8_1  asl dzielna             ; przesuń dzielną w lewo: najstarszy bit -> C, bit 0 := 0
        rol @                   ; wsuń ten bit do reszty (reszta = reszta * 2 + bit)
        bcs div8_2              ; reszta przekroczyła 255 - na pewno jest >= dzielnik
        cmp dzielnik            ; czy reszta >= dzielnik ?
        bcc div8_3              ; nie: bit ilorazu = 0 (już tam jest po ASL)
div8_2  sbc dzielnik            ; tak: reszta := reszta - dzielnik (C = 1 na obu ścieżkach)
        inc dzielna             ; bit ilorazu = 1 (ustaw bit 0 zwolniony przez ASL)
div8_3  dex                     ; następny bit
        bne div8_1              ; powtórz 8 razy
        sta reszta              ; zapisz resztę
        rts                     ; powrót z podprogramu
```

Dwa szczegóły zasługują na komentarz.

Brak sec przed `sbc`. Do rozkazu `sbc dzielnik` prowadzą dwie drogi: skok `bcs` (wykonywany tylko, gdy C = 1) i przejście przez `bcc`, który nie skoczył (czyli też C = 1). Znacznik jest więc zawsze ustawiony.

Skok `bcs div8_2`. Obsługuje dzielniki większe od 128. Reszta jest zawsze mniejsza od dzielnika, ale po podwojeniu może przekroczyć 255 — wtedy dziewiąty bit ląduje w C. Taka reszta jest na pewno większa od dzielnika, więc odejmujemy bez porównywania (ośmiobitowy wynik odejmowania jest poprawny). Wiele procedur dzielenia spotykanych w starych książkach pomija ten przypadek i myli się dla dużych dzielników.

15.4 Dzielenie 16 : 8 bitów

Najczęściej potrzebne dzielenie w praktyce: duża liczba przez małą — choćby przez 10 przy zamianie na cyfry dziesiętne. Reszta mieści się w bajcie, więc może mieszkać w akumulatorze.

Plik: div16_8.asm

```
; div16_8 - dzielenie liczby 16-bitowej przez 8-bitową, bez znaku
; Wejście:  dzielna16 (2 bajty), dzielnik8 (1 bajt, różny od zera)
; Wyjście:  dzielna16 = iloraz (16 bitów), A = reszta
; Niszczy:  X. Zachowuje: dzielnik8.
dzielna16 = $80                 ; dzielna, a po zakończeniu iloraz (2 bajty)
dzielnik8 = $82                 ; dzielnik (1 bajt)

div16_8 lda #0                  ; A = bieżąca reszta
        ldx #16                 ; X = 16: tyle bitów ma dzielna
d168_1  asl dzielna16           ; przesuń 16-bitową dzielną w lewo...
        rol dzielna16+1         ; ...jej najstarszy bit trafia do C
        rol @                   ; wsuń ten bit do reszty
        bcs d168_2              ; reszta > 255: na pewno odejmujemy
        cmp dzielnik8           ; czy reszta >= dzielnik ?
        bcc d168_3              ; nie: bit ilorazu = 0
d168_2  sbc dzielnik8           ; tak: reszta := reszta - dzielnik (C = 1)
        inc dzielna16           ; bit ilorazu = 1
d168_3  dex                     ; następny bit
```

```
        bne d168_1              ; powtórz 16 razy
        rts                     ; A = reszta
```

15.5 Dzielenie 16 : 16 bitów

Gdy dzielnik ma 16 bitów, reszta też musi mieć 16 bitów i nie zmieści się w akumulatorze.

Plik: div16.asm

```
; div16 - dzielenie liczb 16-bitowych bez znaku
; Wejście:  dna16 (dzielna), dnk16 (dzielnik, różny od zera) - po 2 bajty
; Wyjście:  dna16 = iloraz, rsz16 = reszta
; Niszczy:  A, X. Zachowuje: dnk16.
dna16   = $80                   ; dzielna, a po zakończeniu iloraz (2 bajty)
dnk16   = $82                   ; dzielnik (2 bajty)
rsz16   = $84                   ; reszta (2 bajty)

div16   lda #0                  ; A = 0
        sta rsz16               ; reszta := 0
        sta rsz16+1             ; zapisz A pod adres rsz16+1
        ldx #16                 ; X = 16: tyle bitów ma dzielna
div16_1 asl dna16               ; przesuń dzielną w lewo; najstarszy bit -> C
        rol dna16+1             ; obróć bajt spod adresu dna16+1 w lewo przez C
        rol rsz16               ; wsuń ten bit do 16-bitowej reszty
        rol rsz16+1             ; C = siedemnasty bit reszty (zwykle 0)
        bcs div16_2             ; reszta > 65535: na pewno odejmujemy
        lda rsz16               ; porównaj resztę z dzielnikiem (16-bitowo):
        cmp dnk16               ; młodsze bajty...
        lda rsz16+1             ; A = bajt spod adresu rsz16+1
        sbc dnk16+1             ; ...i starsze z pożyczką
        bcc div16_3             ; reszta < dzielnik: bit ilorazu = 0
div16_2 lda rsz16               ; reszta := reszta - dzielnik (C = 1 na obu ścieżkach)
        sbc dnk16               ; A = A - dnk16 - pożyczka (gdy C = 0)
        sta rsz16               ; zapisz A do rsz16
        lda rsz16+1             ; A = bajt spod adresu rsz16+1
        sbc dnk16+1             ; A = A - bajt spod adresu dnk16+1 - pożyczka (gdy C = 0)
        sta rsz16+1             ; zapisz A pod adres rsz16+1
        inc dna16               ; bit ilorazu = 1
div16_3 dex                     ; następny bit
        bne div16_1             ; powtórz 16 razy
        rts                     ; powrót z podprogramu
```

Porównanie i odejmowanie wykonują tu tę samą pracę dwukrotnie. Wersję szybszą otrzymamy, zapamiętując wynik odejmowania próbnego (młodszy bajt w rejestrze Y, starszy w A) i zapisując go tylko wtedy, gdy nie było pożyczki. Zyskuje się kilkanaście procent czasu kosztem przejrzystości.

15.6 Dzielenie 32 : 32 bity

Plik: div32.asm

```
; div32 - dzielenie liczb 32-bitowych bez znaku
; Wejście:  dna32 (dzielna), dnk32 (dzielnik, różny od zera) - po 4 bajty
; Wyjście:  dna32 = iloraz, rsz32 = reszta
; Niszczy:  A, Y. Zachowuje: dnk32, X.
dna32   = $80                   ; dzielna, a po zakończeniu iloraz (4 bajty)
dnk32   = $84                   ; dzielnik (4 bajty)
rsz32   = $88                   ; reszta (4 bajty)

div32   lda #0                  ; A = 0
        sta rsz32               ; reszta := 0
        sta rsz32+1             ; zapisz A pod adres rsz32+1
        sta rsz32+2             ; zapisz A pod adres rsz32+2
        sta rsz32+3             ; zapisz A pod adres rsz32+3
        ldy #32                 ; Y = 32: tyle bitów ma dzielna
div32_1 asl dna32               ; przesuń 32-bitową dzielną w lewo
        rol dna32+1             ; obróć bajt spod adresu dna32+1 w lewo przez C
        rol dna32+2             ; obróć bajt spod adresu dna32+2 w lewo przez C
        rol dna32+3             ; najstarszy bit -> C
```

```
        rol rsz32               ; wsuń go do 32-bitowej reszty
        rol rsz32+1             ; obróć bajt spod adresu rsz32+1 w lewo przez C
        rol rsz32+2             ; obróć bajt spod adresu rsz32+2 w lewo przez C
        rol rsz32+3             ; C = trzydziesty trzeci bit reszty
        bcs div32_2             ; reszta nie mieści się w 32 bitach: na pewno odejmujemy
        lda rsz32               ; porównaj resztę z dzielnikiem (odejmowanie na próbę)
        cmp dnk32               ; porównaj A z: dnk32
        lda rsz32+1             ; A = bajt spod adresu rsz32+1
        sbc dnk32+1             ; A = A - bajt spod adresu dnk32+1 - pożyczka (gdy C = 0)
        lda rsz32+2             ; A = bajt spod adresu rsz32+2
        sbc dnk32+2             ; A = A - bajt spod adresu dnk32+2 - pożyczka (gdy C = 0)
        lda rsz32+3             ; A = bajt spod adresu rsz32+3
        sbc dnk32+3             ; A = A - bajt spod adresu dnk32+3 - pożyczka (gdy C = 0)
        bcc div32_3             ; reszta < dzielnik: bit ilorazu = 0
div32_2 lda rsz32               ; reszta := reszta - dzielnik (C = 1)
        sbc dnk32               ; A = A - dnk32 - pożyczka (gdy C = 0)
        sta rsz32               ; zapisz A do rsz32
        lda rsz32+1             ; A = bajt spod adresu rsz32+1
        sbc dnk32+1             ; A = A - bajt spod adresu dnk32+1 - pożyczka (gdy C = 0)
        sta rsz32+1             ; zapisz A pod adres rsz32+1
        lda rsz32+2             ; A = bajt spod adresu rsz32+2
        sbc dnk32+2             ; A = A - bajt spod adresu dnk32+2 - pożyczka (gdy C = 0)
        sta rsz32+2             ; zapisz A pod adres rsz32+2
        lda rsz32+3             ; A = bajt spod adresu rsz32+3
        sbc dnk32+3             ; A = A - bajt spod adresu dnk32+3 - pożyczka (gdy C = 0)
        sta rsz32+3             ; zapisz A pod adres rsz32+3
        inc dna32               ; bit ilorazu = 1
div32_3 dey                     ; następny bit
        bne div32_1             ; powtórz 32 razy
        rts                     ; powrót z podprogramu
```

Procedura ta podzieli też liczbę 32-bitową przez 16-bitową — wystarczy wyzerować dwa starsze bajty dzielnika.

15.7 Dzielenie liczb ze znakiem

Podobnie jak przy mnożeniu: dzielimy wartości bezwzględne, a znaki nadajemy na końcu. Przyjęta w większości języków programowania konwencja mówi:

- **iloraz** jest ujemny, gdy dzielna i dzielnik mają różne znaki,
- **reszta** ma znak **dzielnej**.

Na przykład $-7 : 2 = -3$, reszta -1 (bo $-3 \cdot 2 + (-1) = -7$).

Plik: div16s.asm

```
; div16s - dzielenie liczb 16-bitowych ZE ZNAKIEM: dna16 := dna16 / dnk16, rsz16 := reszta
; Wymaga procedury div16 (i jej zmiennych). Niszczy: A, X, dnk16, komórki zn_il i zn_rs.
zn_il   = $86                   ; bit 7 = znak ilorazu
zn_rs   = $87                   ; bit 7 = znak reszty (taki jak znak dzielnej)

div16s  lda dna16+1             ; starszy bajt dzielnej: jego bit 7 to znak
        sta zn_rs               ; reszta będzie miała znak dzielnej
        eor dnk16+1             ; XOR ze znakiem dzielnika
        sta zn_il               ; iloraz ujemny, gdy znaki są różne

        lda dna16+1             ; dzielna ujemna?
        bpl ds_1                ; skocz do ds_1, gdy N = 0 (bit 7 skasowany)
        jsr ds_neg_a            ; tak: dna16 := -dna16
ds_1    lda dnk16+1             ; dzielnik ujemny?
        bpl ds_2                ; skocz do ds_2, gdy N = 0 (bit 7 skasowany)
        sec                     ; tak: dnk16 := 0 - dnk16
        lda #0                  ; A = 0
        sbc dnk16               ; A = A - dnk16 - pożyczka (gdy C = 0)
        sta dnk16               ; zapisz A do dnk16
        lda #0                  ; A = 0
        sbc dnk16+1             ; A = A - bajt spod adresu dnk16+1 - pożyczka (gdy C = 0)
        sta dnk16+1             ; zapisz A pod adres dnk16+1
ds_2    jsr div16               ; podziel wartości bezwzględne
```

```
        bit zn_il               ; iloraz ma być ujemny?
        bpl ds_3                ; skocz do ds_3, gdy N = 0 (bit 7 skasowany)
        jsr ds_neg_a            ; tak: dna16 := -dna16
ds_3    bit zn_rs               ; reszta ma być ujemna?
        bpl ds_4                ; skocz do ds_4, gdy N = 0 (bit 7 skasowany)
        sec                     ; tak: rsz16 := 0 - rsz16
        lda #0                  ; A = 0
        sbc rsz16               ; A = A - rsz16 - pożyczka (gdy C = 0)
        sta rsz16               ; zapisz A do rsz16
        lda #0                  ; A = 0
        sbc rsz16+1             ; A = A - bajt spod adresu rsz16+1 - pożyczka (gdy C = 0)
        sta rsz16+1             ; zapisz A pod adres rsz16+1
ds_4    rts                     ; powrót z podprogramu

ds_neg_a sec                    ; podprogram pomocniczy: dna16 := 0 - dna16
        lda #0                  ; A = 0
        sbc dna16               ; A = A - dna16 - pożyczka (gdy C = 0)
        sta dna16               ; zapisz A do dna16
        lda #0                  ; A = 0
        sbc dna16+1             ; A = A - bajt spod adresu dna16+1 - pożyczka (gdy C = 0)
        sta dna16+1             ; zapisz A pod adres dna16+1
        rts                     ; powrót z podprogramu
```

15.8 Dzielenie przez stałą

Potęgi dwójki — przesunięcia w prawo (rozdział 13); reszta to maska AND.

Dzielenie przez 10 pojawia się przy każdym wypisywaniu liczby. Dla bajtu wygodne jest odejmowanie wielokrotne, które od razu rozbija liczbę 0–99 na cyfry:

Plik: dziel10.asm

```
; dziel10 - rozbicie liczby 0..255 na liczbę dziesiątek i jedności
; Wejście: A = liczba.  Wyjście: X = liczba / 10 (0..25), A = liczba mod 10 (0..9).
dziel10 ldx #0                  ; X = licznik dziesiątek
        sec                     ; C = 1 przed pierwszym odejmowaniem
dz10_1  sbc #10                 ; odejmij 10
        bcc dz10_2              ; pożyczka: odjęliśmy o jeden raz za dużo
        inx                     ; nie było pożyczki: policz dziesiątkę
        bne dz10_1              ; powtarzaj (skok zawsze; C = 1 nadal)
dz10_2  adc #10                 ; cofnij ostatnie odejmowanie (C = 0, więc ADC doda dokładnie 10)
        rts                     ; powrót z podprogramu
```

Inne stałe. Gdy dzielnik jest stały, a szybkość krytyczna, stosuje się tablicę wyników albo mnożenie przez odwrotność (np. dzielenie przez 3 jako mnożenie przez 85 i odrzucenie młodszego bajtu — z niewielkim błędem, który trzeba skorygować). W zwykłych programach wystarcza ogólna procedura dzielenia.

15.9 Reszta z dzielenia (modulo)

Każda z naszych procedur zwraca resztę razem z ilorazem. Jeśli potrzebujesz samej operacji `a MOD b`, wywołaj dzielenie i zignoruj iloraz. Przykład: zawijanie pozycji na ekranie o szerokości 40 znaków to `x := x MOD 40`.

Dla małych wartości wystarczy często jedno porównanie i jedno odejmowanie:

```
; x := (x + 1) MOD 40 - przesuń pozycję z zawijaniem
        inc x                   ; x := x + 1
        lda x                   ; A = x
        cmp #40                 ; doszliśmy do 40 ?
        bcc w_zakresie          ; nie
        lda #0                  ; tak: wróć na początek
        sta x                   ; zapisz A do x
w_zakresie
```

Ćwiczenia

1. Prześledź na kartce działanie `div8` dla 13 : 3.

2. Napisz procedurę obliczającą średnią arytmetyczną dwóch bajtów bez ryzyka przepełnienia. Wskazówka: po dodaniu dziewiąty bit sumy jest w znaczniku C — jeden rozkaz `ror @` dzieli całość przez 2.
3. Uzupełnij `div16` o sprawdzenie dzielnika: jeśli jest zerem, procedura ma wrócić z C = 1, w przeciwnym razie z C = 0.

ROZDZIAŁ 16

Liczby dziesiętne: kod BCD i konwersje

Procesor liczy dwójkowo, człowiek czyta dziesiętnie. Każdy program, który pokazuje wynik, licznik punktów czy numer sektora, musi zamienić liczbę na cyfry — a program przyjmujący dane od użytkownika musi zrobić rzecz odwrotną. Ten rozdział to komplet narzędzi do tego celu.

16.1 Znaki cyfr

W kodzie ATASCII (tak jak w ASCII) cyfry `0`–`9` mają kody `$30`–`$39`. Zamiana wartości 0–9 na znak to dodanie `$30`, a ponieważ młodszy półbajt `$30` jest zerem, można też użyć `ora`:

```
        lda cyfra               ; A = wartość 0..9
        ora #'0'                ; A = kod ATASCII znaku cyfry ($30..$39)

        lda znak                ; A = kod znaku '0'..'9'
        and #$0F                ; A = wartość 0..9 (albo: SEC / SBC #'0')
```

Jeśli cyfry trafiają bezpośrednio do pamięci ekranu, potrzebne są **kody ekranowe**: cyfry to `$10`–`$19`. W MADS zapisuje się to wygodnie jako `ora #"0"` (cudzysłów zamiast apostrofu).

16.2 Liczba 8-bitowa na trzy cyfry

Dla jednego bajtu najprościej odejmować setki, potem dziesiątki:

Plik: bin8dec.asm

```
; bin8dec - zamiana bajtu na trzy cyfry dziesiętne ATASCII
; Wejście:  A = liczba 0..255
; Wyjście:  cyfry8 (3 bajty): setki, dziesiątki, jedności jako znaki '0'..'9'
; Niszczy:  A, X.
cyfry8  = $80                   ; bufor na 3 znaki

bin8dec ldx #'0'                ; X = znak cyfry setek, na początku '0'
b8_1    cmp #100                ; czy zostało co najmniej 100 ?
        bcc b8_2                ; nie - cyfra setek gotowa
        sbc #100                ; tak: odejmij 100 (C = 1 po CMP)
        inx                     ; cyfra setek + 1
        bne b8_1                ; powtarzaj (skok zawsze)
b8_2    stx cyfry8              ; zapisz znak setek
        ldx #'0'                ; X = znak cyfry dziesiątek
b8_3    cmp #10                 ; czy zostało co najmniej 10 ?
        bcc b8_4                ; nie - cyfra dziesiątek gotowa
        sbc #10                 ; tak: odejmij 10
        inx                     ; cyfra dziesiątek + 1
        bne b8_3                ; powtarzaj (skok zawsze)
b8_4    stx cyfry8+1            ; zapisz znak dziesiątek
        ora #'0'                ; w A zostały jedności 0..9: zamień na znak
        sta cyfry8+2            ; zapisz znak jedności
        rts                     ; powrót z podprogramu
```

16.3 Liczba 16-bitowa na pięć cyfr

Ta sama metoda dla liczb do 65535: odejmujemy kolejno 10000, 1000, 100 i 10, licząc, ile razy się udało. Potęgi dziesiątki trzymamy w tablicy rozdzielonej na młodsze i starsze bajty.

Plik: bin16dec.asm

```
; bin16dec - zamiana liczby 16-bitowej na pięć cyfr dziesiętnych ATASCII
; Wejście:  liczba16 (2 bajty)
; Wyjście:  cyfry16 (5 bajtów): znaki '0'..'9', od dziesiątek tysięcy do jedności
; Niszczy:  A, X, Y oraz zmienną liczba16.
liczba16 = $80                  ; zamieniana liczba (2 bajty) - zostaje zniszczona
cyfry16 = $82                   ; bufor na 5 znaków

bin16dec ldx #0                 ; X = numer cyfry i jednocześnie numer potęgi dziesiątki
b16_1    ldy #'0'               ; Y = znak bieżącej cyfry, zaczynamy od '0'
b16_2    lda liczba16           ; porównaj liczbę z bieżącą potęgą dziesiątki:
         cmp pot_lo,x           ; młodsze bajty...
         lda liczba16+1         ; A = bajt spod adresu liczba16+1
         sbc pot_hi,x           ; ...i starsze z pożyczką; A = starszy bajt różnicy
         bcc b16_3              ; liczba < potęga: cyfra gotowa
         sta liczba16+1         ; liczba >= potęga: zapisz starszy bajt różnicy,
         lda liczba16           ; a młodszy oblicz jeszcze raz
         sbc pot_lo,x           ; (C = 1, więc odejmowanie jest dokładne)
         sta liczba16           ; zapisz A do liczba16
         iny                    ; cyfra + 1
         bne b16_2              ; spróbuj odjąć jeszcze raz (skok zawsze)
b16_3    sty cyfry16,x          ; zapisz znak cyfry na pozycji X
         inx                    ; następna cyfra, następna potęga
         cpx #4                 ; cztery cyfry gotowe?
         bne b16_1              ; nie - licz następną
         lda liczba16           ; zostały jedności (0..9)
         ora #'0'               ; zamień na znak
         sta cyfry16+4          ; zapisz piątą cyfrę
         rts                    ; powrót z podprogramu

pot_lo   dta l(10000,1000,100,10)  ; młodsze bajty potęg dziesiątki
pot_hi   dta h(10000,1000,100,10)  ; starsze bajty potęg dziesiątki
```

Każda cyfra wymaga najwyżej 9 odejmowań, więc cała zamiana to co najwyżej 36 obiegów pętli — szybciej niż pięciokrotne wywołanie ogólnej procedury dzielenia przez 10.

16.4 Liczba 32-bitowa na dziesięć cyfr

Zasada bez zmian; tablica zawiera dziewięć potęg dziesiątki po cztery bajty. Rejestr Y indeksuje tablicę (skacząc co 4), a X numeruje cyfry.

Plik: bin32dec.asm

```
; bin32dec - zamiana liczby 32-bitowej na dziesięć cyfr dziesiętnych ATASCII
; Wejście:  liczba32 (4 bajty)
; Wyjście:  cyfry32 (10 bajtów): znaki '0'..'9', od miliardów do jedności
; Niszczy:  A, X, Y oraz zmienną liczba32.
liczba32 = $80                  ; zamieniana liczba (4 bajty) - zostaje zniszczona
cyfry32 = $84                   ; bufor na 10 znaków

bin32dec ldx #0                 ; X = numer cyfry (0..9)
         ldy #0                 ; Y = przesunięcie w tablicy potęg (0, 4, 8, ...)
b32_1    lda #'0'               ; wpisz '0' jako początkową wartość cyfry
         sta cyfry32,x          ; zapisz A pod adres cyfry32+X
b32_2    lda liczba32           ; porównaj liczbę z bieżącą potęgą dziesiątki
         cmp pot32,y            ; (32-bitowe odejmowanie na próbę)
         lda liczba32+1         ; A = bajt spod adresu liczba32+1
         sbc pot32+1,y          ; A = A - bajt spod adresu pot32+1+Y - pożyczka (gdy C = 0)
         lda liczba32+2         ; A = bajt spod adresu liczba32+2
         sbc pot32+2,y          ; A = A - bajt spod adresu pot32+2+Y - pożyczka (gdy C = 0)
         lda liczba32+3         ; A = bajt spod adresu liczba32+3
         sbc pot32+3,y          ; A = A - bajt spod adresu pot32+3+Y - pożyczka (gdy C = 0)
         bcc b32_3              ; liczba < potęga: cyfra gotowa
         lda liczba32           ; liczba := liczba - potęga (C = 1)
         sbc pot32,y            ; A = A - bajt spod adresu pot32+Y - pożyczka (gdy C = 0)
         sta liczba32           ; zapisz A do liczba32
         lda liczba32+1         ; A = bajt spod adresu liczba32+1
         sbc pot32+1,y          ; A = A - bajt spod adresu pot32+1+Y - pożyczka (gdy C = 0)
         sta liczba32+1         ; zapisz A pod adres liczba32+1
         lda liczba32+2         ; A = bajt spod adresu liczba32+2
```

```
        sbc pot32+2,y           ; A = A - bajt spod adresu pot32+2+Y - pożyczka (gdy C = 0)
        sta liczba32+2          ; zapisz A pod adres liczba32+2
        lda liczba32+3          ; A = bajt spod adresu liczba32+3
        sbc pot32+3,y           ; A = A - bajt spod adresu pot32+3+Y - pożyczka (gdy C = 0)
        sta liczba32+3          ; zapisz A pod adres liczba32+3
        inc cyfry32,x           ; cyfra + 1 (znak '0' staje się '1' itd.)
        bne b32_2               ; spróbuj odjąć jeszcze raz (skok zawsze)
b32_3   iny                     ; przesuń się do następnej potęgi: 4 bajty dalej
        iny                     ; Y = Y + 1
        iny                     ; Y = Y + 1
        iny                     ; Y = Y + 1
        inx                     ; następna cyfra
        cpx #9                  ; dziewięć cyfr gotowych?
        bne b32_1               ; nie - licz następną
        lda liczba32            ; zostały jedności
        ora #'0'                ; A = A OR kod ATASCII znaku '0'
        sta cyfry32+9           ; zapisz dziesiątą cyfrę
        rts                     ; powrót z podprogramu

pot32   dta f(1000000000)       ; tablica potęg dziesiątki, każda jako liczba 32-bitowa
        dta f(100000000)        ; liczba 32-bitowa
        dta f(10000000)         ; liczba 32-bitowa
        dta f(1000000)          ; liczba 32-bitowa
        dta f(100000)           ; liczba 32-bitowa
        dta f(10000)            ; liczba 32-bitowa
        dta f(1000)             ; liczba 32-bitowa
        dta f(100)              ; liczba 32-bitowa
        dta f(10)               ; liczba 32-bitowa
```

16.5 Usuwanie zer wiodących

Liczba „00042” wygląda źle. Zera z przodu zamieniamy na spacje — ale ostatnią cyfrę zostawiamy zawsze, żeby zero było wypisane jako „0”, a nie jako pusty napis.

Plik: bez_zer.asm

```
; bez_zer - zamienia zera wiodące na spacje w buforze cyfr ATASCII
; Wejście:  bz_wsk = adres bufora (2 bajty na stronie zerowej), Y = liczba cyfr (2..255)
; Niszczy:  A, Y, komórkę bz_ost.
bz_wsk  = $80                   ; wskaźnik na bufor cyfr
bz_ost  = $82                   ; indeks ostatniej cyfry

bez_zer dey                     ; Y = indeks ostatniej cyfry (liczba cyfr - 1)
        sty bz_ost              ; zapamiętaj: tej cyfry nigdy nie zamieniamy
        ldy #0                  ; Y = indeks bieżącej cyfry, od lewej
bz_1    lda (bz_wsk),y          ; A = bieżący znak
        cmp #'0'                ; czy to zero?
        bne bz_k                ; nie - koniec zer wiodących
        lda #' '                ; tak: zastąp spacją
        sta (bz_wsk),y          ; zapisz A pod adres (bz_wsk) + Y
        iny                     ; następny znak
        cpy bz_ost              ; doszliśmy do ostatniej cyfry?
        bne bz_1                ; nie - sprawdzaj dalej
bz_k    rts                     ; powrót z podprogramu
```

16.6 Kod BCD i tryb dziesiętny procesora

W kodzie BCD każdy półbajt przechowuje jedną cyfrę dziesiętną: bajt `$59` oznacza pięćdziesiąt dziewięć. Po włączeniu trybu dziesiętnego rozkazem `sed` rozkazy `adc` i `sbc` same pilnują, żeby wynik pozostał poprawną liczbą BCD:

```
        sed                     ; włącz tryb dziesiętny
        lda #$59                ; A = 59 (BCD)
        clc                     ; C = 0 (przed dodawaniem)
        adc #$03                ; A = $62: sześćdziesiąt dwa (dwójkowo byłoby $5C)
        cld                     ; wyłącz tryb dziesiętny
```

Przeniesienie C działa jak zwykle, tyle że oznacza przekroczenie 99. Można więc budować liczby BCD dowolnej długości tak samo jak dwójkowe. Najważniejsze zastosowanie to **licznik punktów**: przechowywany w BCD jest od razu gotowy do wyświetlenia, bez żadnego dzielenia.

Plik: punkty.asm

```
; dodaj_pkt - dodaje do sześciocyfrowego licznika punktów (BCD) wartość 0..99 (BCD)
; Wejście:  A = dodawana wartość w kodzie BCD (np. $25 = dwadzieścia pięć punktów)
; Wyjście:  punkty (3 bajty BCD; punkty+0 = dwie najmłodsze cyfry)
punkty  = $80                   ; licznik: 3 bajty = 6 cyfr dziesiętnych

dodaj_pkt sed                   ; tryb dziesiętny
        clc                     ; C = 0 (przed dodawaniem)
        adc punkty              ; dodaj do dwóch najmłodszych cyfr
        sta punkty              ; zapisz A do punkty
        lda punkty+1            ; dwie środkowe cyfry...
        adc #0                  ; ...plus przeniesienie
        sta punkty+1            ; zapisz A pod adres punkty+1
        lda punkty+2            ; dwie najstarsze cyfry...
        adc #0                  ; ...plus przeniesienie
        sta punkty+2            ; zapisz A pod adres punkty+2
        cld                     ; KONIECZNIE wyłącz tryb dziesiętny
        rts                     ; powrót z podprogramu
```

Wyświetlenie bajtu BCD to rozdzielenie półbajtów:

Plik: bcd2asc.asm

```
; bcd2asc - zamiana bajtu BCD na dwa znaki ATASCII
; Wejście:  A = bajt BCD (np. $47).  Wyjście: A = znak dziesiątek, X = znak jedności.
bcd2asc pha                     ; zapamiętaj bajt
        and #$0F                ; młodszy półbajt: jedności
        ora #'0'                ; zamień na znak
        tax                     ; X = znak jedności
        pla                     ; odtwórz bajt
        lsr @                   ; przesuń starszy półbajt na miejsce młodszego
        lsr @                   ; przesuń A o 1 bit w prawo (bit 0 -> C)
        lsr @                   ; przesuń A o 1 bit w prawo (bit 0 -> C)
        lsr @                   ; przesuń A o 1 bit w prawo (bit 0 -> C)
        ora #'0'                ; A = znak dziesiątek
        rts                     ; powrót z podprogramu
```

Pułapka

Trzy pułapki trybu dziesiętnego.

1. **Zapomniane `cld`** psuje całą dalszą arytmetykę programu. Zawsze zamykaj fragment BCD.
2. Po `adc`/`sbc` w trybie dziesiętnym wiarygodny jest tylko znacznik **C**. Znaczniki Z, N i V w klasycznym 6502 nie odpowiadają wynikowi dziesiętnemu.
3. **`inc` i `dec` nie znają trybu dziesiętnego** — zawsze liczą dwójkowo. `inc` na bajcie `$09` da `$0A`, czyli niepoprawną liczbę BCD. Licznik BCD zwiększaj przez `adc #1`.

System operacyjny Atari w swoich procedurach obsługi przerwań sam wyłącza tryb dziesiętny, więc przerwanie, które nadejdzie między `sed` a `cld`, niczego nie zepsuje. Własna procedura przerwania DLI nie przechodzi jednak przez ten fragment systemu — jeśli używa `adc` lub `sbc`, powinna zacząć od `cld` (rozkaz `rti` i tak przywróci przerwanemu programowi jego stan znacznika D).

16.7 Zamiana liczby dwójkowej na BCD

Tryb dziesiętny pozwala na zaskakująco krótki algorytm zamiany liczby dwójkowej na BCD. Pomysł: wynik budujemy, wsuwając bity liczby od najstarszego i za każdym razem **podwajając wynik w arytmetyce dziesiętnej** (wynik := wynik · 2 + bit). Podwojenie to dodanie liczby do samej siebie.

Plik: bin16bcd.asm

```
; bin16bcd - zamiana liczby 16-bitowej na 5 cyfr w kodzie BCD
; Wejście:  bin16 (2 bajty)
; Wyjście:  bcd24 (3 bajty BCD: bcd24+0 = dwie najmłodsze cyfry, bcd24+2 = najstarsza)
```

```
; Niszczy:  A, X oraz zmienną bin16.
bin16   = $80                   ; liczba dwójkowa (2 bajty) - zostaje wyzerowana
bcd24   = $82                   ; wynik BCD (3 bajty)

bin16bcd lda #0                 ; A = 0
        sta bcd24               ; wynik := 0
        sta bcd24+1             ; zapisz A pod adres bcd24+1
        sta bcd24+2             ; zapisz A pod adres bcd24+2
        sed                     ; tryb dziesiętny
        ldx #16                 ; X = 16 bitów do przetworzenia
bb_1    asl bin16               ; wysuń najstarszy bit liczby do C
        rol bin16+1             ; obróć bajt spod adresu bin16+1 w lewo przez C
        lda bcd24               ; wynik := wynik + wynik + C (dziesiętnie),
        adc bcd24               ; czyli wynik * 2 + wysunięty bit
        sta bcd24               ; zapisz A do bcd24
        lda bcd24+1             ; A = bajt spod adresu bcd24+1
        adc bcd24+1             ; A = A + bajt spod adresu bcd24+1 + C
        sta bcd24+1             ; zapisz A pod adres bcd24+1
        lda bcd24+2             ; A = bajt spod adresu bcd24+2
        adc bcd24+2             ; A = A + bajt spod adresu bcd24+2 + C
        sta bcd24+2             ; zapisz A pod adres bcd24+2
        dex                     ; następny bit
        bne bb_1                ; skocz do bb_1, gdy Z = 0 (różne / nie zero)
        cld                     ; wyłącz tryb dziesiętny
        rts                     ; powrót z podprogramu
```

Szesnaście obiegów, żadnych tablic, żadnego dzielenia. Wynik wyświetlamy procedurą `bcd2asc`.

16.8 Wypisywanie szesnastkowe

Do diagnostyki i narzędzi systemowych najwygodniejszy jest zapis szesnastkowy. Najprostsza i najszybsza zamiana półbajtu na znak korzysta z tablicy:

Plik: hex8.asm

```
; hex8 - zamiana bajtu na dwa znaki szesnastkowe ATASCII
; Wejście:  A = bajt.  Wyjście: A = znak starszego półbajtu, X = znak młodszego.
; Niszczy:  Y.
hex8    pha                     ; zapamiętaj bajt
        and #$0F                ; młodszy półbajt (0..15)
        tay                     ; jako indeks do tablicy znaków
        ldx hex_tab,y           ; X = znak młodszego półbajtu
        pla                     ; odtwórz bajt
        lsr @                   ; starszy półbajt na miejsce młodszego
        lsr @                   ; przesuń A o 1 bit w prawo (bit 0 -> C)
        lsr @                   ; przesuń A o 1 bit w prawo (bit 0 -> C)
        lsr @                   ; przesuń A o 1 bit w prawo (bit 0 -> C)
        tay                     ; jako indeks
        lda hex_tab,y           ; A = znak starszego półbajtu
        rts                     ; powrót z podprogramu

hex_tab dta c'0123456789ABCDEF'  ; znak dla każdej wartości półbajtu
```

Liczbę 16-bitową wypisujemy, wywołując `hex8` dla starszego, a potem dla młodszego bajtu.

16.9 Zamiana tekstu dziesiętnego na liczbę

Operacja odwrotna: użytkownik wpisał „1234”, a my potrzebujemy liczby. Czytamy cyfry od lewej; przed dołączeniem każdej następnej dotychczasowy wynik mnożymy przez 10:

```
   "1234":   0 * 10 + 1 = 1
             1 * 10 + 2 = 12
            12 * 10 + 3 = 123
           123 * 10 + 4 = 1234
```

Plik: dec2bin.asm

```
; dec2bin - zamiana napisu z cyframi dziesiętnymi na liczbę 16-bitową
; Wejście:  db_wsk = adres napisu; zamiana kończy się na pierwszym znaku niebędącym cyfrą
; Wyjście:  wart16 = liczba, Y = liczba przetworzonych cyfr (0 = napis nie zaczynał się cyfrą)
;           C = 0: w porządku;  C = 1: liczba nie mieści się w 16 bitach
; Niszczy:  A, komórki db_tmp.
db_wsk  = $80                   ; wskaźnik na napis (2 bajty)
wart16  = $82                   ; wynik (2 bajty)
db_tmp  = $84                   ; zmienne pomocnicze (3 bajty)

dec2bin ldy #0                  ; Y = indeks bieżącego znaku
        sty wart16              ; wynik := 0
        sty wart16+1            ; zapisz Y pod adres wart16+1
db_1    lda (db_wsk),y          ; A = bieżący znak
        sec                     ; C = 1 (przed odejmowaniem)
        sbc #'0'                ; A = znak - '0': dla cyfr wychodzi 0..9
        cmp #10                 ; czy to cyfra? (znaki poniżej '0' dają tu wartości >= 246)
        bcs db_ok               ; nie: koniec liczby
        sta db_tmp+2            ; zapamiętaj wartość cyfry

        asl wart16              ; wynik := wynik * 2
        rol wart16+1            ; obróć bajt spod adresu wart16+1 w lewo przez C
        bcs db_nadm             ; bit wypadł poza 16 bitów: przepełnienie
        lda wart16              ; db_tmp := wynik * 2 (zapamiętaj)
        sta db_tmp              ; zapisz A do db_tmp
        lda wart16+1            ; A = bajt spod adresu wart16+1
        sta db_tmp+1            ; zapisz A pod adres db_tmp+1
        asl wart16              ; wynik := wynik * 4
        rol wart16+1            ; obróć bajt spod adresu wart16+1 w lewo przez C
        bcs db_nadm             ; skocz do db_nadm, gdy C = 1
        asl wart16              ; wynik := wynik * 8
        rol wart16+1            ; obróć bajt spod adresu wart16+1 w lewo przez C
        bcs db_nadm             ; skocz do db_nadm, gdy C = 1
        clc                     ; wynik := wynik * 8 + wynik * 2 = wynik * 10
        lda wart16              ; A = wart16
        adc db_tmp              ; A = A + db_tmp + C
        sta wart16              ; zapisz A do wart16
        lda wart16+1            ; A = bajt spod adresu wart16+1
        adc db_tmp+1            ; A = A + bajt spod adresu db_tmp+1 + C
        sta wart16+1            ; zapisz A pod adres wart16+1
        bcs db_nadm             ; skocz do db_nadm, gdy C = 1

        lda wart16              ; wynik := wynik + cyfra (C = 0 po poprzednim skoku)
        adc db_tmp+2            ; A = A + bajt spod adresu db_tmp+2 + C
        sta wart16              ; zapisz A do wart16
        bcc db_2                ; bez przeniesienia
        inc wart16+1            ; przeniesienie do starszego bajtu
        beq db_nadm             ; starszy bajt przekręcił się na zero: przepełnienie
db_2    iny                     ; następny znak
        bne db_1                ; powtarzaj (najwyżej 255 znaków)
db_ok   clc                     ; C = 0: zamiana udana
        rts                     ; powrót z podprogramu
db_nadm sec                     ; C = 1: liczba za duża
        rts                     ; powrót z podprogramu
```

Procedura rozpoznaje koniec liczby po pierwszym znaku, który nie jest cyfrą — może to być koniec linii `$9B`, spacja, przecinek. Wywołujący sprawdza po powrocie znacznik C oraz rejestr Y (zero oznacza, że nie było żadnej cyfry).

16.10 Pakiet zmiennoprzecinkowy systemu Atari

Dla porządku warto wiedzieć, że Atari ma w pamięci ROM (`$D800`–`$DFFF`) pakiet arytmetyki zmiennoprzecinkowej, z którego korzysta BASIC. Liczby mają w nim postać sześciobajtową w kodzie BCD, a pakiet oferuje między innymi zamianę napisu na liczbę (AFP, `$D800`), liczby na napis (FASC, `$D8E6`), liczby całkowitej 16-bitowej na zmiennoprzecinkową (IFP, `$D9AA`) i odwrotnie (FPI, `$D9D2`) oraz cztery działania. Parą wywołań IFP i FASC można wypisać liczbę 16-bitową bez własnej procedury konwersji.

Pakiet jest jednak powolny, zajmuje sporą część strony zerowej (`$D4`–`$FF`) i znika, gdy wyłączymy ROM systemu, żeby odzyskać pamięć. Własne procedury całkowitoliczbowe — takie jak w tym rozdziale — są krótsze, szybsze

i niezależne od systemu.

Ćwiczenia

1. Napisz procedurę wypisującą liczbę 16-bitową szesnastkowo (cztery znaki) do bufora.
2. Zmodyfikuj `bin16dec` tak, by zapisywała kody ekranowe zamiast ATASCII.
3. Napisz procedurę odejmującą punkty od licznika BCD, która nie pozwala zejść poniżej zera.

Wskazówka do 2: wystarczy zamienić `'0'` na `"0"` w trzech miejscach.

Cześć IV

Dane

ROZDZIAŁ 17

Tablice, wskaźniki i napisy

W asemblerze nie ma typów danych. Jest tylko pamięć — ciąg ponumerowanych bajtów — i sposób, w jaki postanowisz ją interpretować. Tablica, rekord, napis czy wskaźnik to wyłącznie **konwencje**. Ten rozdział przedstawia te, które sprawdziły się najlepiej.

17.1 Zmienne

Zmienna to nazwany adres. Mamy dwa sposoby jej utworzenia.

Nazwanie komórki strony zerowej — bez generowania czegokolwiek w pliku:

```
licznik = $80                   ; zmienna 8-bitowa pod adresem $80
wsk     = $81                   ; zmienna 16-bitowa: zajmuje $81 i $82
suma    = $83                   ; zmienna 32-bitowa: zajmuje $83..$86
```

Asembler nie pilnuje, ile bajtów zajmuje zmienna. To **ty** musisz zadbać, by następna nazwa dostała adres o odpowiednią liczbę bajtów dalej. Pomyłka — dwie zmienne zachodzące na siebie — daje błędy bardzo trudne do znalezienia. Wygodnym zabezpieczeniem jest liczenie adresów względem poprzedniej zmiennej:

```
licznik = $80                   ; 1 bajt
wsk     = licznik+1             ; 2 bajty - zaraz za licznikiem
suma    = wsk+2                 ; 4 bajty - zaraz za wskaźnikiem
nast    = suma+4                ; następna wolna komórka
```

Zarezerwowanie miejsca w obrębie programu — dyrektywą `dta` (z wartością początkową) lub `.ds` (bez wartości):

```
zycia   dta 3                   ; zmienna 8-bitowa z wartością początkową 3
rekord  dta a(0)                ; zmienna 16-bitowa z wartością początkową 0
bufor   .ds 40                  ; 40 bajtów bez wartości początkowej
```

> **Uwaga**
>
> Wartość nadana przez `dta` jest wpisywana **tylko raz — przy ładowaniu programu z dysku**. Jeśli program uruchamia się ponownie bez ładowania (np. po powrocie do menu gry), zmienna będzie miała wartość z poprzedniego przebiegu. Zmienne, które muszą startować od znanej wartości, ustawiaj rozkazami na początku programu.

17.2 Tablice bajtów

Tablica to ciąg bajtów pod wspólną etykietą. Element numer *i* leży pod adresem `tablica+i`:

```
        ldx #2                  ; X = indeks (liczymy od zera)
        lda dni,x               ; A = 31 (trzeci element)
        rts                     ; powrót z podprogramu

dni     dta 31,28,31,30,31,30,31,31,30,31,30,31  ; liczba dni w miesiącach
ILE_MIES = *-dni                ; liczba elementów: bieżący adres minus początek = 12
```

Zapis `ILE_MIES = *-dni` to idiom wart zapamiętania: asembler sam liczy długość tablicy i nie trzeba poprawiać stałej po dopisaniu elementów.

Zapis do tablicy wygląda tak samo: `sta tablica,x`. Indeksem może być X lub Y; wybierz ten, który jest wolny.

17.3 Tablice słów

Element 16-bitowy zajmuje dwa bajty, więc indeks trzeba pomnożyć przez 2:

```
        lda numer               ; A = numer elementu (0..127)
        asl @                   ; A = numer * 2: przesunięcie w bajtach
        tax                     ; X = przesunięcie
        lda progi,x             ; młodszy bajt elementu
        sta wynik               ; zapisz A do wynik
        lda progi+1,x           ; starszy bajt elementu
        sta wynik+1             ; zapisz A pod adres wynik+1
        rts                     ; powrót z podprogramu

progi   dta a(100,500,1000,5000,10000)  ; pięć liczb 16-bitowych
```

Lepszym rozwiązaniem są zwykle **tablice rozdzielone** — osobna dla młodszych bajtów, osobna dla starszych. Numer elementu jest wtedy od razu indeksem, a tablica może mieć 256 elementów:

```
        ldx numer               ; X = numer elementu (0..255) - bez mnożenia
        lda progi_lo,x          ; młodszy bajt
        sta wynik               ; zapisz A do wynik
        lda progi_hi,x          ; starszy bajt
        sta wynik+1             ; zapisz A pod adres wynik+1
        rts                     ; powrót z podprogramu

progi_lo dta l(100,500,1000,5000,10000)  ; młodsze bajty pięciu liczb
progi_hi dta h(100,500,1000,5000,10000)  ; starsze bajty tych samych liczb
```

17.4 Rekordy, czyli tablice struktur

Załóżmy, że gra ma 8 przeciwników, a każdy ma współrzędne x, y, rodzaj i energię. W języku wysokiego poziomu byłaby to tablica rekordów. W 6502 znacznie wygodniej odwrócić układ i utworzyć **osobną tablicę dla każdego pola**:

```
ILE_WR  = 8                     ; liczba wrogów

; --- odwołanie do wroga numer X: po prostu ten sam indeks we wszystkich tablicach ---
        ldx #3                  ; X = numer wroga
        inc wr_x,x              ; przesuń go o 1 w prawo
        lda wr_energia,x        ; A = jego energia
        beq martwy              ; zero: wróg nie żyje
martwy  rts                     ; powrót z podprogramu

wr_x    .ds ILE_WR              ; współrzędne x wszystkich wrogów
wr_y    .ds ILE_WR              ; współrzędne y
wr_rodzaj .ds ILE_WR            ; rodzaje
wr_energia .ds ILE_WR           ; energie
```

Numer obiektu trzymamy w rejestrze X przez całą procedurę obsługi i sięgamy do dowolnego pola jednym rozkazem. Nie potrzeba mnożenia ani wskaźników. Ten układ („struktura tablic") jest podstawą większości gier na 6502.

17.5 Tablice dwuwymiarowe

Pamięć jest jednowymiarowa, więc tablicę o W kolumnach zapisujemy wiersz po wierszu. Element (kolumna x, wiersz y) leży pod adresem `baza + y * W + x`. Mnożenia można uniknąć na dwa sposoby.

Szerokość będąca potęgą dwójki — mnożenie zastępują przesunięcia. Dla planszy 16 × 16 pól:

```
; A := plansza[x, y] dla planszy 16 kolumn x 16 wierszy
        lda y                   ; A = numer wiersza (0..15)
        asl @                   ; * 2
        asl @                   ; * 4
        asl @                   ; * 8
        asl @                   ; * 16: przesunięcie początku wiersza
```

```
        ora x                   ; dołącz numer kolumny (0..15) w młodszym półbajcie
        tax                     ; X = indeks elementu (0..255)
        lda plansza,x           ; A = zawartość pola
```

Tablica adresów wierszy — rozwiązanie ogólne i najszybsze. Dla ekranu 40 × 24 znaki:

```
; Ustawia wskaźnik "wsk" na początek wiersza numer Y bufora o wymiarach 40 x 24
        lda wiersz_lo,y         ; młodszy bajt adresu wiersza Y
        sta wsk                 ; zapisz A do wsk
        lda wiersz_hi,y         ; starszy bajt adresu wiersza Y
        sta wsk+1               ; zapisz A pod adres wsk+1
        ldy kolumna             ; Y = numer kolumny (0..39)
        lda (wsk),y             ; A = znak w tej kolumnie i tym wierszu
```

Tablicę adresów wierszy może wygenerować asembler — pokażemy to w rozdziale 22.

17.6 Wskaźniki

Wskaźnik to zmienna 16-bitowa **na stronie zerowej**, zawierająca adres. Przez wskaźnik sięgamy do danych trybem `(wsk),y`. Podstawowe operacje:

```
wsk     = $80                   ; wskaźnik: komórki $80 (młodszy bajt) i $81 (starszy)

; --- ustawienie wskaźnika na znany adres ---
        lda #<dane              ; młodszy bajt adresu
        sta wsk                 ; zapisz A do wsk
        lda #>dane              ; starszy bajt adresu
        sta wsk+1               ; zapisz A pod adres wsk+1

; --- odczyt i zapis przez wskaźnik ---
        ldy #0                  ; Y = 0: dokładnie tam, gdzie wskazuje wskaźnik
        lda (wsk),y             ; odczyt
        sta (wsk),y             ; zapis
        ldy #5                  ; Y = 5: pięć bajtów dalej - jak pole rekordu
        lda (wsk),y             ; A = bajt spod adresu (wsk) + Y

; --- przesunięcie wskaźnika o 1 bajt ---
        inc wsk                 ; zwiększ młodszy bajt
        bne w1                  ; nie przekręcił się - gotowe
        inc wsk+1               ; przekręcił się: zwiększ starszy
w1

; --- przesunięcie wskaźnika o N bajtów ---
        lda wsk                 ; A = wsk
        clc                     ; C = 0 (przed dodawaniem)
        adc #40                 ; np. o 40: następny wiersz ekranu
        sta wsk                 ; zapisz A do wsk
        bcc w2                  ; bez przeniesienia - gotowe
        inc wsk+1               ; z przeniesieniem: starszy bajt + 1
w2
```

Przy przeglądaniu danych po kolei mamy wybór: zwiększać wskaźnik (dowolna długość danych) albo zwiększać rejestr Y przy nieruchomym wskaźniku (szybciej, ale tylko do 256 bajtów). Procedury z następnego rozdziału łączą obie metody.

17.7 Napisy

Napis to tablica znaków — ale skąd wiadomo, gdzie się kończy? Stosuje się kilka konwencji:

Konwencja	Przykład w MADS	Zalety i wady
znak końca `$9B` (EOL)	`dta c'Tekst',$9B`	standard Atari: tak kończy wiersze system operacyjny
bajt zerowy na końcu	`dta c'Tekst',0`	łatwy test (`beq`); zero nie może wystąpić w tekście

Konwencja	Przykład w MADS	Zalety i wady
długość na początku	`dta 5,c'Tekst'`	długość znana od razu; najwyżej 255 znaków
stała długość	`dta c'Tekst   '`	najprostsze; marnuje miejsce
bit 7 w ostatnim znaku	`dta c'Teks','t'+$80`	oszczędza bajt; kłopotliwe w obsłudze

W tej książce stosujemy konwencję systemu Atari: napis kończy się znakiem **EOL** o kodzie `$9B` (155). Zaletą jest to, że takie napisy można wprost przekazywać procedurom systemowym.

17.7.1 Długość napisu

Plik: strlen.asm

```
; strlen - długość napisu zakończonego znakiem EOL ($9B)
; Wejście:  s_wsk = adres napisu
; Wyjście:  Y = liczba znaków bez znaku EOL (0..255)
; Niszczy:  A.
EOL     = $9B                   ; kod końca linii w systemie Atari
s_wsk   = $80                   ; wskaźnik na napis (2 bajty)

strlen  ldy #0                  ; Y = licznik znaków i jednocześnie indeks
sl_1    lda (s_wsk),y           ; A = kolejny znak
        cmp #EOL                ; koniec napisu?
        beq sl_k                ; tak: Y zawiera długość
        iny                     ; nie: licz dalej
        bne sl_1                ; (zabezpieczenie: najwyżej 255 znaków)
sl_k    rts                     ; powrót z podprogramu
```

17.7.2 Kopiowanie napisu

Plik: strcpy.asm

```
; strcpy - kopiuje napis zakończony EOL (razem ze znakiem EOL)
; Wejście:  s_zr = adres napisu źródłowego, s_cel = adres miejsca docelowego
; Wyjście:  Y = długość napisu (bez EOL)
; Niszczy:  A.
EOL     = $9B                   ; znak końca linii
s_zr    = $80                   ; wskaźnik źródła (2 bajty)
s_cel   = $82                   ; wskaźnik celu (2 bajty)

strcpy  ldy #0                  ; Y = indeks znaku
scp_1   lda (s_zr),y            ; pobierz znak ze źródła
        sta (s_cel),y           ; zapisz do celu (STA nie zmienia znaczników ani A)
        cmp #EOL                ; czy skopiowany znak to EOL?
        beq scp_k               ; tak: koniec
        iny                     ; nie: następny znak
        bne scp_1               ; (najwyżej 255 znaków)
scp_k   rts                     ; powrót z podprogramu
```

17.7.3 Porównywanie napisów

Plik: strcmp.asm

```
; strcmp - porównuje dwa napisy zakończone EOL
; Wejście:  s_a, s_b = adresy napisów
; Wyjście:  Z = 1: napisy identyczne
;           Z = 0: różne; wtedy C = 0, gdy pierwszy jest "wcześniej w alfabecie",
;                  a C = 1, gdy "później"
; Niszczy:  A, Y.
EOL     = $9B                   ; znak końca linii
s_a     = $80                   ; wskaźnik pierwszego napisu (2 bajty)
s_b     = $82                   ; wskaźnik drugiego napisu (2 bajty)

strcmp  ldy #0                  ; Y = indeks porównywanego znaku
```

```
scm_1   lda (s_a),y             ; znak z pierwszego napisu
        cmp (s_b),y             ; porównaj ze znakiem z drugiego
        bne scm_k               ; różne: znaczniki Z i C opisują wynik - koniec
        cmp #EOL                ; takie same; czy to koniec obu napisów?
        beq scm_k               ; tak: Z = 1 - napisy identyczne
        iny                     ; nie: następna para znaków
        bne scm_1               ; skocz do scm_1, gdy Z = 0 (różne / nie zero)
scm_k   rts                     ; powrót z podprogramu
```

Ponieważ kod EOL (`$9B`) jest większy od kodów liter i cyfr, krótszy napis będący początkiem dłuższego („ALA" i „ALAN") zostanie uznany za *późniejszy*. Jeżeli potrzebujesz porządku słownikowego, zakończ napisy bajtem zerowym i porównuj z zerem.

17.7.4 Zamiana na wielkie litery

Plik: upcase.asm

```
; upcase - zamienia w napisie małe litery a..z na wielkie A..Z
; Wejście:  s_wsk = adres napisu zakończonego EOL.  Niszczy: A, Y.
EOL     = $9B                   ; znak końca linii
s_wsk   = $80                   ; wskaźnik na napis (2 bajty)

upcase  ldy #0                  ; Y = indeks znaku
uc_1    lda (s_wsk),y           ; A = znak
        cmp #EOL                ; koniec napisu?
        beq uc_k                ; skocz do uc_k, gdy Z = 1 (równe / zero)
        cmp #'a'                ; znak < 'a' ?
        bcc uc_2                ; tak: to nie mała litera
        cmp #'z'+1              ; znak > 'z' ?
        bcs uc_2                ; tak: to nie mała litera
        and #%11011111          ; mała litera: skasuj bit 5 (kody różnią się o 32)
        sta (s_wsk),y           ; zapisz wielką literę
uc_2    iny                     ; następny znak
        bne uc_1                ; skocz do uc_1, gdy Z = 0 (różne / nie zero)
uc_k    rts                     ; powrót z podprogramu
```

17.7.5 Wyszukiwanie znaku

Plik: strchr.asm

```
; strchr - szuka znaku w napisie zakończonym EOL
; Wejście:  A = szukany znak, s_wsk = adres napisu
; Wyjście:  C = 0: znaleziono, Y = pozycja;  C = 1: nie znaleziono
; Niszczy:  komórkę s_znak.
EOL     = $9B                   ; znak końca linii
s_wsk   = $80                   ; wskaźnik na napis (2 bajty)
s_znak  = $82                   ; szukany znak

strchr  sta s_znak              ; zapamiętaj szukany znak
        ldy #0                  ; Y = indeks
sch_1   lda (s_wsk),y           ; A = znak z napisu
        cmp s_znak              ; czy to szukany znak?
        beq sch_jest            ; tak
        cmp #EOL                ; koniec napisu?
        beq sch_brak            ; tak: nie ma takiego znaku
        iny                     ; szukaj dalej
        bne sch_1               ; skocz do sch_1, gdy Z = 0 (różne / nie zero)
sch_brak sec                    ; C = 1: nie znaleziono
        rts                     ; powrót z podprogramu
sch_jest clc                    ; C = 0: znaleziono na pozycji Y
        rts                     ; powrót z podprogramu
```

17.8 Tablice jako zamienniki obliczeń

Na 8-bitowym procesorze najszybszym sposobem obliczenia funkcji jest… nieobliczanie jej. Jeśli argument mieści się w bajcie, wynik można wcześniej zapisać w tablicy 256-bajtowej i odczytać jednym rozkazem `lda tablica,x`.

W ten sposób realizuje się:

- zamianę kodów znaków (ATASCII na ekranowe, małe litery na wielkie),
- odwracanie kolejności bitów w bajcie (lustrzane odbicie grafiki),
- funkcje sinus i cosinus dla ruchu po okręgu,
- mnożenie przez stałą, kwadraty liczb, pierwiastki,
- adresy wierszy ekranu, maski bitów, kształty kursorów.

Tablicę można wpisać ręcznie, wygenerować zewnętrznym programem i dołączyć dyrektywą `ins`, albo zbudować krótką procedurą przy starcie programu.

Ćwiczenia

1. Napisz procedurę `strcat`, która dokleja napis `s_zr` na koniec napisu `s_cel` (wskazówka: najpierw znajdź EOL w `s_cel`, potem kopiuj).
2. Napisz procedurę odwracającą kolejność znaków w napisie (bez znaku EOL).
3. Dla tablicy `wr_energia` z przykładu o wrogach napisz pętlę zliczającą żywych przeciwników.

ROZDZIAŁ 18

Kopiowanie, wypełnianie i porównywanie bloków pamięci

Przenoszenie danych to najczęstsza czynność programu: przepisanie bufora na ekran, wyczyszczenie tablicy, przesunięcie zawartości ekranu, skopiowanie zestawu znaków z ROM-u do RAM-u. Procesor 6502 nie ma do tego specjalnych rozkazów — każdy bajt przechodzi przez akumulator. Tym ważniejsze jest, by pętla kopiująca była dobrze napisana.

18.1 Bloki do 256 bajtów

Gdy adresy są znane w czasie asemblacji, a blok ma najwyżej 256 bajtów, wystarczy indeksowanie rejestrem.

18.1.1 Kopiowanie w dół (najkrótsze)

```
; Kopiuje 40 bajtów z "zrodlo" do "cel"
        ldx #39                 ; X = indeks ostatniego bajtu (długość - 1)
kop     lda zrodlo,x            ; pobierz bajt
        sta cel,x               ; zapisz bajt
        dex                     ; poprzedni bajt
        bpl kop                 ; powtarzaj dla X = 39 ... 0
```

Wersja z `bpl` działa dla bloków do 128 bajtów. Dla większych używamy adresów pomniejszonych o 1 i skoku `bne`:

```
; Kopiuje 200 bajtów z "zrodlo" do "cel"
        ldx #200                ; X = długość (1..255)
kop2    lda zrodlo-1,x          ; X = 200 sięga do bajtu 199, X = 1 do bajtu 0
        sta cel-1,x             ; zapisz A pod adres cel-1+X
        dex                     ; X = X - 1
        bne kop2                ; powtarzaj do X = 0
```

18.1.2 Dokładnie 256 bajtów (jedna strona)

```
        ldx #0                  ; X = 0
kop3    lda zrodlo,x            ; kopiuj bajt X
        sta cel,x               ; zapisz A pod adres cel+X
        inx                     ; następny bajt
        bne kop3                ; X wróci do zera po 256 bajtach
```

18.1.3 Wypełnianie

```
; Zeruje 256 bajtów od adresu "bufor"
        lda #0                  ; A = wartość wypełniająca
        tax                     ; X = 0 (TAX jest krótszy niż LDX #0)
wyp     sta bufor,x             ; zapisz bajt
        inx                     ; X = X + 1
        bne wyp                 ; 256 obiegów
```

18.2 Bloki dowolnej długości

Dla bloków większych niż 256 bajtów — albo gdy adresy nie są znane z góry — potrzebne są wskaźniki na stronie zerowej i tryb `(wsk),y`. Najszybszy schemat dzieli pracę na dwie części:

1. **pełne strony** (po 256 bajtów): wewnętrzna pętla z rejestrem Y, a po każdej stronie zwiększenie starszych bajtów wskaźników,
2. **reszta** (0–255 bajtów): krótka pętla końcowa.

18.2.1 Kopiowanie od początku bloku

Plik: memcpy.asm

```
; memcpy - kopiuje blok pamięci dowolnej długości, zaczynając od najniższego adresu
; Wejście:  mc_zr  = adres źródła (2 bajty)
;           mc_cel = adres celu (2 bajty)
;           mc_dl  = długość bloku w bajtach (2 bajty; może być zerem)
; Niszczy:  A, X, Y; zmienia starsze bajty mc_zr i mc_cel.
; Uwaga:    jeżeli bloki nachodzą na siebie, cel musi leżeć PONIŻEJ źródła.
mc_zr   = $80                   ; wskaźnik źródła
mc_cel  = $82                   ; wskaźnik celu
mc_dl   = $84                   ; długość

memcpy  ldy #0                  ; Y = indeks bajtu na stronie
        ldx mc_dl+1             ; X = liczba pełnych stron (starszy bajt długości)
        beq mc_3                ; brak pełnych stron: przejdź do reszty
mc_1    lda (mc_zr),y           ; --- kopiowanie pełnej strony ---
        sta (mc_cel),y          ; zapisz A pod adres (mc_cel) + Y
        iny                     ; następny bajt
        bne mc_1                ; powtarzaj, aż Y wróci do zera (256 bajtów)
        inc mc_zr+1             ; przesuń oba wskaźniki o całą stronę
        inc mc_cel+1            ; zwiększ o 1 bajt spod adresu mc_cel+1
        dex                     ; jedna strona mniej
        bne mc_1                ; są jeszcze pełne strony
mc_3    ldx mc_dl               ; X = liczba pozostałych bajtów (młodszy bajt długości)
        beq mc_k                ; zero: koniec
mc_4    lda (mc_zr),y           ; --- kopiowanie reszty; Y = 0 na wejściu ---
        sta (mc_cel),y          ; zapisz A pod adres (mc_cel) + Y
        iny                     ; następny bajt
        dex                     ; jeden bajt mniej
        bne mc_4                ; skocz do mc_4, gdy Z = 0 (różne / nie zero)
mc_k    rts                     ; powrót z podprogramu
```

Wewnętrzna pętla to cztery rozkazy i 16 cykli na bajt. Skopiowanie 1 KB zajmuje około 17 000 cykli, czyli mniej więcej połowę czasu trwania jednej ramki obrazu.

Przykład wywołania — skopiowanie zestawu znaków (1024 bajty) z pamięci ROM do RAM:

```
        lda #<$E000             ; źródło: zestaw znaków w ROM-ie
        sta mc_zr               ; zapisz A do mc_zr
        lda #>$E000             ; A = starszy bajt wartości $E000
        sta mc_zr+1             ; zapisz A pod adres mc_zr+1
        lda #<nowe_znaki        ; cel: nasz bufor
        sta mc_cel              ; zapisz A do mc_cel
        lda #>nowe_znaki        ; A = starszy bajt adresu nowe_znaki
        sta mc_cel+1            ; zapisz A pod adres mc_cel+1
        lda #<1024              ; długość: 1024 bajty
        sta mc_dl               ; zapisz A do mc_dl
        lda #>1024              ; A = starszy bajt wartości 1024
        sta mc_dl+1             ; zapisz A pod adres mc_dl+1
        jsr memcpy              ; wywołaj podprogram memcpy
```

18.2.2 Problem nakładających się bloków

Co się stanie, gdy przesuwamy blok o kilka bajtów **w górę** pamięci, a obszary źródła i celu częściowo się pokrywają?

```
Przesunięcie 6 bajtów o 2 pozycje w górę, kopiowanie OD POCZĄTKU - źle:

adres:      0   1   2   3   4   5   6   7
przed:      A   B   C   D   E   F   .   .
krok 1:     A   B   A   D   E   F   .   .     bajt 0 -> 2   (C zniszczone!)
krok 2:     A   B   A   B   E   F   .   .     bajt 1 -> 3   (D zniszczone!)
krok 3:     A   B   A   B   A   F   .   .     bajt 2 -> 4   (kopiujemy już kopię)
...
wynik:      A   B   A   B   A   B   A   B     zamiast  A B A B C D E F
```

Kopiowanie od początku niszczy dane, zanim zdąży je przeczytać. Rozwiązanie: w takim przypadku kopiować **od końca**. Reguła:

Zapamiętaj

- cel **poniżej** źródła (przesuwanie w dół) → kopiuj **od początku**,
- cel **powyżej** źródła (przesuwanie w górę) → kopiuj **od końca**,
- bloki rozłączne → kierunek obojętny.

18.2.3 Kopiowanie od końca bloku

Plik: memcpy_k.asm

```
; memcpy_k - kopiuje blok pamięci dowolnej długości, zaczynając od NAJWYŻSZEGO adresu
; Wejście:  mc_zr, mc_cel, mc_dl - jak w procedurze memcpy
; Niszczy:  A, X, Y; zmienia starsze bajty mc_zr i mc_cel.
; Uwaga:    jeżeli bloki nachodzą na siebie, cel musi leżeć POWYŻEJ źródła.
mc_zr   = $80                   ; wskaźnik źródła
mc_cel  = $82                   ; wskaźnik celu
mc_dl   = $84                   ; długość

memcpy_k clc                    ; ustaw wskaźniki na ostatnią, niepełną stronę bloku:
        lda mc_zr+1             ; do starszego bajtu każdego wskaźnika
        adc mc_dl+1             ; dodaj liczbę pełnych stron
        sta mc_zr+1             ; zapisz A pod adres mc_zr+1
        clc                     ; C = 0 (przed dodawaniem)
        lda mc_cel+1            ; A = bajt spod adresu mc_cel+1
        adc mc_dl+1             ; A = A + bajt spod adresu mc_dl+1 + C
        sta mc_cel+1            ; zapisz A pod adres mc_cel+1
        ldy mc_dl               ; Y = liczba bajtów w niepełnej stronie (reszta)
        beq mk_3                ; zero: od razu pełne strony
mk_1    dey                     ; --- kopiowanie reszty od końca ---
        lda (mc_zr),y           ; pobierz bajt
        sta (mc_cel),y          ; zapisz bajt
        tya                     ; czy Y doszedł do zera? (TYA ustawia znacznik Z)
        bne mk_1                ; nie: kopiuj poprzedni bajt
mk_3    ldx mc_dl+1             ; X = liczba pełnych stron
        beq mk_k                ; zero: koniec
mk_4    dec mc_zr+1             ; cofnij oba wskaźniki o jedną stronę
        dec mc_cel+1            ; zmniejsz o 1 bajt spod adresu mc_cel+1
mk_5    dey                     ; --- kopiowanie pełnej strony od końca; Y: 255 ... 0 ---
        lda (mc_zr),y           ; A = bajt spod adresu (mc_zr) + Y
        sta (mc_cel),y          ; zapisz A pod adres (mc_cel) + Y
        tya                     ; Y = 0 ?
        bne mk_5                ; nie: kopiuj dalej
        dex                     ; jedna strona mniej
        bne mk_4                ; skocz do mk_4, gdy Z = 0 (różne / nie zero)
mk_k    rts                     ; powrót z podprogramu
```

18.2.4 Bezpieczne przesuwanie: wybór kierunku

Procedura, która sama wybiera właściwy kierunek, porównuje adresy źródła i celu:

Plik: memmove.asm

```
; memmove - kopiuje blok pamięci, poprawnie także wtedy, gdy bloki się nakładają
; Wejście:  mc_zr, mc_cel, mc_dl - jak w procedurze memcpy
```

```
; Wymaga:   procedur memcpy i memcpy_k.
memmove lda mc_cel              ; porównaj 16-bitowo: cel z adresem źródła
        cmp mc_zr               ; (młodsze bajty)
        lda mc_cel+1            ; A = bajt spod adresu mc_cel+1
        sbc mc_zr+1             ; (starsze bajty z pożyczką)
        bcs mm_1                ; cel >= źródło: kopiuj od końca
        jmp memcpy              ; cel < źródło: kopiuj od początku
mm_1    jmp memcpy_k            ; skocz do memcpy_k
```

18.3 Wypełnianie bloku dowolnej długości

Plik: memset.asm

```
; memset - wypełnia blok pamięci jedną wartością
; Wejście:  A = wartość wypełniająca
;           ms_cel = adres bloku (2 bajty), ms_dl = długość (2 bajty; może być zerem)
; Niszczy:  X, Y; zmienia starszy bajt ms_cel. Zachowuje: A.
ms_cel  = $80                   ; wskaźnik bloku
ms_dl   = $82                   ; długość

memset  ldy #0                  ; Y = indeks bajtu na stronie
        ldx ms_dl+1             ; X = liczba pełnych stron
        beq ms_3                ; brak: przejdź do reszty
ms_1    sta (ms_cel),y          ; --- wypełnianie pełnej strony ---
        iny                     ; Y = Y + 1
        bne ms_1                ; 256 bajtów
        inc ms_cel+1            ; następna strona
        dex                     ; X = X - 1
        bne ms_1                ; skocz do ms_1, gdy Z = 0 (różne / nie zero)
ms_3    ldx ms_dl               ; X = liczba pozostałych bajtów
        beq ms_k                ; zero: koniec
ms_4    sta (ms_cel),y          ; --- wypełnianie reszty ---
        iny                     ; Y = Y + 1
        dex                     ; X = X - 1
        bne ms_4                ; skocz do ms_4, gdy Z = 0 (różne / nie zero)
ms_k    rts                     ; powrót z podprogramu
```

18.4 Porównywanie bloków

Plik: memcmp.asm

```
; memcmp - porównuje dwa bloki pamięci
; Wejście:  cm_a, cm_b = adresy bloków (po 2 bajty), cm_dl = długość (2 bajty)
; Wyjście:  Z = 1: bloki identyczne
;           Z = 0: różne; C = 0, gdy w miejscu pierwszej różnicy bajt bloku A jest mniejszy
; Niszczy:  A, X, Y; zmienia starsze bajty cm_a i cm_b.
cm_a    = $80                   ; wskaźnik pierwszego bloku
cm_b    = $82                   ; wskaźnik drugiego bloku
cm_dl   = $84                   ; długość

memcmp  ldy #0                  ; Y = indeks bajtu na stronie
        ldx cm_dl+1             ; X = liczba pełnych stron
        beq cm_3                ; skocz do cm_3, gdy Z = 1 (równe / zero)
cm_1    lda (cm_a),y            ; --- porównywanie pełnej strony ---
        cmp (cm_b),y            ; porównaj A z: bajt spod adresu (cm_b) + Y
        bne cm_k                ; pierwsza różnica: koniec, znaczniki opisują wynik
        iny                     ; Y = Y + 1
        bne cm_1                ; skocz do cm_1, gdy Z = 0 (różne / nie zero)
        inc cm_a+1              ; następna strona
        inc cm_b+1              ; zwiększ o 1 bajt spod adresu cm_b+1
        dex                     ; X = X - 1
        bne cm_1                ; skocz do cm_1, gdy Z = 0 (różne / nie zero)
cm_3    ldx cm_dl               ; X = liczba pozostałych bajtów; Z = 1, gdy zero
        beq cm_k                ; nic więcej do porównania: bloki identyczne (Z = 1)
cm_4    lda (cm_a),y            ; --- porównywanie reszty ---
        cmp (cm_b),y            ; porównaj A z: bajt spod adresu (cm_b) + Y
        bne cm_k                ; różnica
        iny                     ; Y = Y + 1
```

```
        dex                     ; X = X - 1
        bne cm_4                ; po ostatnim bajcie DEX daje zero, więc Z = 1
cm_k    rts                     ; powrót z podprogramu
```

Typowe zastosowanie: weryfikacja zapisu na dyskietce — po zapisaniu sektora odczytujemy go do drugiego bufora i porównujemy oba bufory.

18.5 Szybciej: kod samomodyfikujący się

Tryb `(wsk),y` kosztuje 5–6 cykli na dostęp. Tryb `adres,x` — tylko 4–5 i nie zajmuje strony zerowej, ale adres jest wpisany w rozkaz na stałe. Chyba że... zmienimy sam rozkaz. Program w pamięci RAM może **modyfikować własny kod**: wpisać nowy adres do dwóch bajtów argumentu rozkazu.

Plik: kopiuj_strony.asm

```
; kopiuj_strony - szybkie kopiowanie X pełnych stron (po 256 bajtów)
; Wejście:  ks_zr+1, ks_zr+2  = adres źródła, wpisany w argument rozkazu LDA
;           ks_cel+1, ks_cel+2 = adres celu, wpisany w argument rozkazu STA
;           X = liczba stron do skopiowania (1..255)
; Niszczy:  A, X, Y.
kopiuj_strony ldy #0            ; Y = indeks bajtu na stronie
ks_zr   lda $FFFF,y             ; adres $FFFF zostanie podmieniony przed wywołaniem
ks_cel  sta $FFFF,y             ; ten również
        iny                     ; Y = Y + 1
        bne ks_zr               ; kopiuj 256 bajtów
        inc ks_zr+2             ; zwiększ starszy bajt adresu WEWNĄTRZ rozkazu LDA
        inc ks_cel+2            ; zwiększ starszy bajt adresu wewnątrz rozkazu STA
        dex                     ; jedna strona mniej
        bne ks_zr               ; skocz do ks_zr, gdy Z = 0 (różne / nie zero)
        rts                     ; powrót z podprogramu
```

Wywołanie polega na wpisaniu adresów do wnętrza rozkazów:

```
        lda #<$E000             ; adres źródła -> argument rozkazu LDA
        sta ks_zr+1             ; (bajt +1 to młodszy bajt argumentu,
        lda #>$E000             ; bajt +2 to starszy)
        sta ks_zr+2             ; zapisz A pod adres ks_zr+2
        lda #<bufor             ; adres celu -> argument rozkazu STA
        sta ks_cel+1            ; zapisz A pod adres ks_cel+1
        lda #>bufor             ; A = starszy bajt adresu bufor
        sta ks_cel+2            ; zapisz A pod adres ks_cel+2
        ldx #4                  ; 4 strony = 1024 bajty
        jsr kopiuj_strony       ; wywołaj podprogram kopiuj_strony
```

Etykieta `ks_zr` wskazuje kod rozkazu `lda`, więc `ks_zr+1` i `ks_zr+2` to dwa bajty jego argumentu. Adres zastępczy `$FFFF` wymusza na asemblerze tryb absolutny (trzybajtowy) — gdyby stało tam `$00`, MADS wybrałby krótszy tryb strony zerowej i nie byłoby miejsca na starszy bajt.

> **Uwaga**
>
> Kod samomodyfikujący się jest w świecie 6502 techniką powszechną i w pełni uprawnioną, ale ma ograniczenia: nie działa w pamięci ROM (np. na kartridżu), utrudnia czytanie programu i wymaga, by procedura nie była wywoływana jednocześnie z programu głównego i z przerwania. Każde takie miejsce wyraźnie oznacz komentarzem.

18.6 Przykład: przewijanie ekranu tekstowego

Przesunięcie zawartości ekranu o jeden wiersz w górę to kopiowanie 23 wierszy (920 bajtów) o 40 bajtów w dół pamięci. Cel leży poniżej źródła, więc kopiujemy od początku. Ostatni wiersz czyścimy.

Plik: przewin.asm

```
; ------------------------------------------------------------------------
; przewin.asm - przewija ekran tekstowy o jeden wiersz w górę co 1/5 sekundy,
;               aż do naciśnięcia dowolnego klawisza
; ------------------------------------------------------------------------
```

```
SAVMSC  = $58                   ; adres początku pamięci ekranu (2 bajty)
RTCLOK  = $14                   ; zegar systemowy - najmłodszy bajt
CH      = $02FC                 ; kod naciśniętego klawisza ($FF = żaden)

zr      = $80                   ; wskaźnik źródła
cel     = $82                   ; wskaźnik celu
dl      = $84                   ; długość (2 bajty)

        org $2000               ; adres asemblacji: $2000

start   lda #$FF                ; skasuj ewentualny wcześniejszy klawisz
        sta CH                  ; zapisz A do CH - kod klawisza
petla   jsr przewin             ; przewiń ekran o wiersz
        ldx #10                 ; odczekaj 10 ramek (1/5 sekundy w systemie PAL)
czekaj  lda RTCLOK              ; A = bieżący stan zegara
cz1     cmp RTCLOK              ; czekaj na zmianę zegara...
        beq cz1                 ; ...czyli na następną ramkę
        dex                     ; X = X - 1
        bne czekaj              ; skocz do czekaj, gdy Z = 0 (różne / nie zero)
        lda CH                  ; czy naciśnięto klawisz?
        cmp #$FF                ; porównaj A z: $FF
        beq petla               ; nie: przewijaj dalej
        lda #$FF                ; tak: skasuj klawisz
        sta CH                  ; zapisz A do CH - kod klawisza
        rts                     ; i wróć do DOS-u

; --- przewin: przesuwa 23 wiersze o jeden w górę, ostatni wiersz czyści -----
przewin lda SAVMSC              ; cel := początek ekranu (wiersz 0)
        sta cel                 ; zapisz A do cel
        clc                     ; C = 0 (przed dodawaniem)
        adc #40                 ; źródło := początek ekranu + 40 (wiersz 1)
        sta zr                  ; zapisz A do zr
        lda SAVMSC+1            ; A = bajt spod adresu SAVMSC+1
        sta cel+1               ; zapisz A pod adres cel+1
        adc #0                  ; starszy bajt z przeniesieniem
        sta zr+1                ; zapisz A pod adres zr+1
        lda #<[23*40]           ; długość := 23 wiersze po 40 znaków = 920 bajtów
        sta dl                  ; zapisz A do dl
        lda #>[23*40]           ; A = starszy bajt wartości [23*40]
        sta dl+1                ; zapisz A pod adres dl+1

        ldy #0                  ; --- kopiowanie od początku (jak w procedurze memcpy) ---
        ldx dl+1                ; X = liczba pełnych stron
        beq p3                  ; skocz do p3, gdy Z = 1 (równe / zero)
p1      lda (zr),y              ; A = bajt spod adresu (zr) + Y
        sta (cel),y             ; zapisz A pod adres (cel) + Y
        iny                     ; Y = Y + 1
        bne p1                  ; skocz do p1, gdy Z = 0 (różne / nie zero)
        inc zr+1                ; zwiększ o 1 bajt spod adresu zr+1
        inc cel+1               ; zwiększ o 1 bajt spod adresu cel+1
        dex                     ; X = X - 1
        bne p1                  ; skocz do p1, gdy Z = 0 (różne / nie zero)
p3      ldx dl                  ; X = reszta bajtów
        beq p5                  ; skocz do p5, gdy Z = 1 (równe / zero)
p4      lda (zr),y              ; A = bajt spod adresu (zr) + Y
        sta (cel),y             ; zapisz A pod adres (cel) + Y
        iny                     ; Y = Y + 1
        dex                     ; X = X - 1
        bne p4                  ; skocz do p4, gdy Z = 0 (różne / nie zero)

p5      lda #0                  ; --- czyszczenie ostatniego wiersza ---
        ldx #40                 ; X = 40 znaków do wyczyszczenia
p6      sta (cel),y             ; po skopiowaniu 920 bajtów para (cel),y wskazuje bajt numer 920
        iny                     ; ekranu, czyli początek ostatniego (24.) wiersza - kontynuujemy
        dex                     ; zapis od tego miejsca, wpisując zera (kod ekranowy spacji)
        bne p6                  ; skocz do p6, gdy Z = 0 (różne / nie zero)
        rts                     ; powrót z podprogramu

        org $02E0               ; wektor RUNAD: adres uruchomienia programu
        dta a(start)            ; adres uruchomienia: start
```

Zwróć uwagę na czyszczenie ostatniego wiersza: nie ustawiamy wskaźnika od nowa, tylko korzystamy z tego, że po zakończeniu kopiowania wskaźnik `cel` wraz z rejestrem Y wskazuje dokładnie pierwszy bajt za skopiowanym

blokiem — a to właśnie początek ostatniego wiersza ekranu.

Ćwiczenia

1. Napisz program przewijający ekran o jeden wiersz **w dół**. Który kierunek kopiowania jest tu potrzebny?
2. Napisz procedurę `memswap`, która zamienia miejscami zawartość dwóch bloków pamięci o tej samej długości (do 256 bajtów).
3. Ile cykli na bajt zajmuje pętla wewnętrzna procedury `kopiuj_strony`, a ile `memcpy`? (Skorzystaj z tabel w rozdziale 6.)

Odpowiedzi: (1) Od końca — cel leży powyżej źródła. (3) `lda adres,y` 4 + `sta adres,y` 5 + `iny` 2 + `bne` 3 = 14 cykli wobec 16 w `memcpy`.

Cześć V

Atari XL/XE

ROZDZIAŁ 19

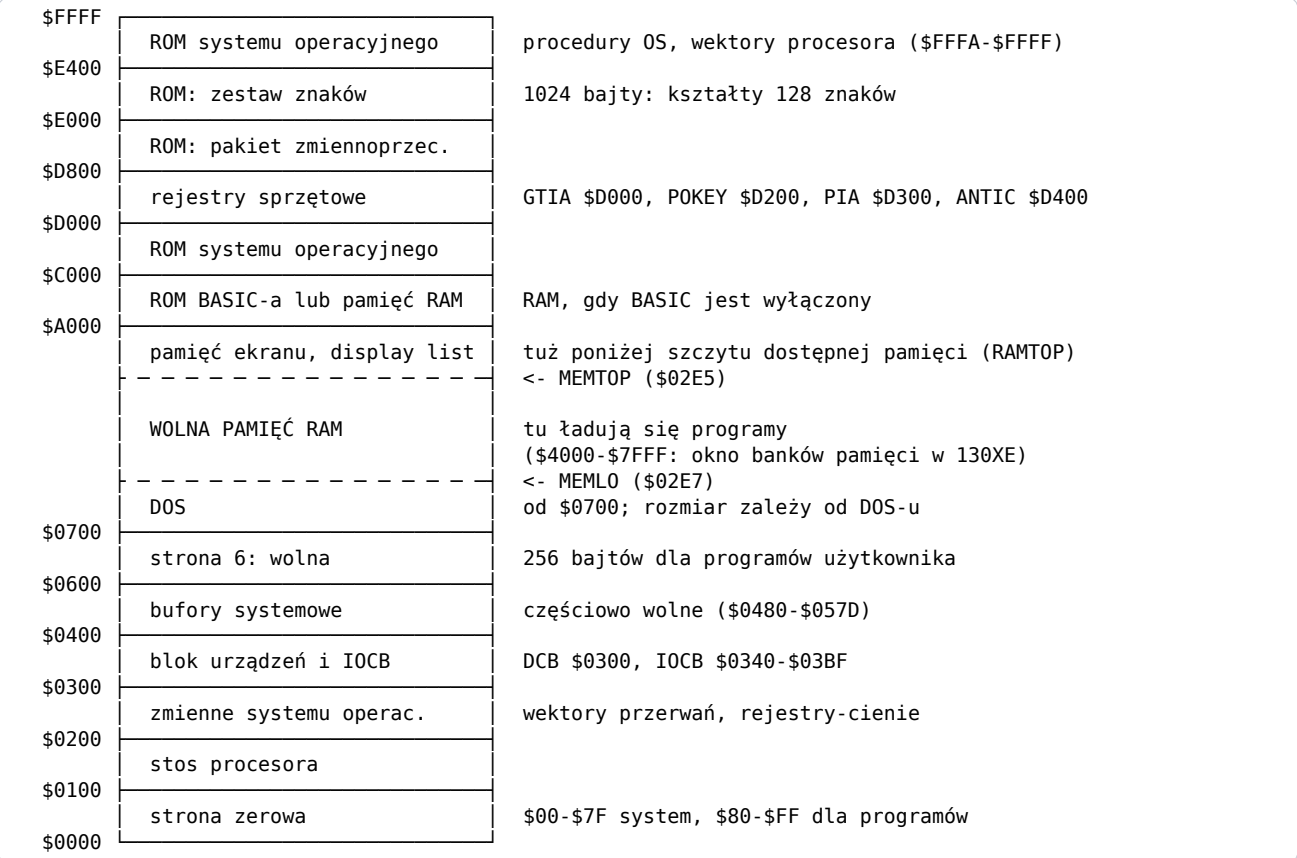
Mapa pamięci i rejestry-cienie

Do tej pory procesor był dla nas abstrakcyjną maszyną. Teraz osadzamy go w konkretnym komputerze. Atari XL/XE to — oprócz procesora — cztery wyspecjalizowane układy oraz system operacyjny w pamięci ROM:

Układ	Zadania	Adresy rejestrów
ANTIC	generowanie obrazu według programu zwanego *display list*, przerwania NMI	`$D400`–`$D40F`
GTIA	kolory, duszki (grafika PMG), klawisze konsoli	`$D000`–`$D01F`
POKEY	dźwięk, klawiatura, transmisja szeregowa (stacja dysków), liczniki, liczby losowe	`$D200`–`$D20F`
PIA	porty joysticków, sterowanie konfiguracją pamięci (PORTB)	`$D300`–`$D303`

Układy te widać w przestrzeni adresowej procesora jako zwykłe komórki pamięci. Zapis pod adres `$D01A` nie zapamiętuje bajtu w pamięci RAM — ustawia kolor tła w układzie GTIA. To się nazywa **wejście/wyjście odwzorowane w pamięci**; 6502 nie ma osobnych rozkazów do obsługi urządzeń.

19.1 Mapa pamięci



Kilka uwag do tej mapy.

RAM leży także „pod" ROM-em. Atari 65XE czy 800XL ma pełne 64 KB pamięci RAM, ale w obszarach `$C000`–`$CFFF` i `$D800`–`$FFFF` (oraz `$A000`–`$BFFF` przy włączonym BASIC-u) procesor widzi normalnie ROM. Pamięć RAM

pod spodem można odsłonić, przełączając bity rejestru PORTB (rozdział 28).

Pamięć ekranu jest na górze. System umieszcza obraz tuż pod szczytem dostępnej pamięci RAM. Przy wyłączonym BASIC-u szczyt to `$C000` i ekran tekstowy zaczyna się pod adresem `$BC40`; przy włączonym — szczyt to `$A000`, a ekran leży pod `$9C40`. Nigdy nie wpisuj tych adresów na stałe: aktualny adres ekranu odczytuje się ze zmiennej SAVMSC (`$58/$59`).

Gdzie umieścić program? Bezpieczny początek to `$2000` — powyżej każdego typowego DOS-u. Dokładną dolną granicę wolnej pamięci podaje zmienna MEMLO (`$02E7`), górną — MEMTOP (`$02E5`). Program o rozmiarze kilku–kilkunastu kilobajtów zaczynający się od `$2000` nie musi się nimi przejmować. Jeśli planujesz użycie pamięci dodatkowej, pamiętaj, że obszar `$4000–$7FFF` jest „oknem" banków (rozdział 28).

19.2 Strona zerowa: co wolno zająć

Strona zerowa jest najcenniejszym zasobem — tylko tam mogą leżeć wskaźniki. Jej podział w Atari:

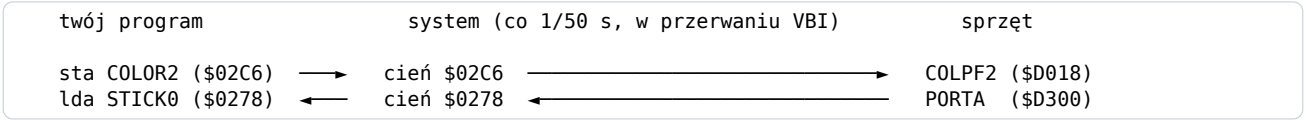
Zakres	Kto używa
`$00–$7F`	system operacyjny. **Nie ruszać** (poza udokumentowanymi zmiennymi).
`$80–$D1`	BASIC. Wolne, gdy BASIC nie jest używany.
`$CB–$D1`	7 bajtów wolnych **zawsze**, nawet przy pracującym BASIC-u.
`$D2–$D3`	zarezerwowane dla BASIC-a i kartridży.
`$D4–$FF`	pakiet zmiennoprzecinkowy. Wolne, gdy program go nie wywołuje.

Wniosek praktyczny: **program w asemblerze uruchamiany z DOS-u, który nie korzysta z BASIC-a ani z procedur zmiennoprzecinkowych, ma do dyspozycji `$80–$FF`, czyli 128 bajtów strony zerowej.** Tak zakładają wszystkie przykłady w tej książce. Procedury w języku maszynowym wywoływane z BASIC-a (instrukcją USR) muszą zadowolić się zakresem `$CB–$D1`.

Inne wolne obszary: **strona 6** (`$0600–$06FF`) — tradycyjne miejsce na krótkie procedury i bufory — oraz obszar `$0480–$057D`, wolny, jeśli nie używa się magnetofonu ani pakietu zmiennoprzecinkowego.

19.3 Rejestry sprzętowe i ich cienie

Większość rejestrów sprzętowych jest **tylko do zapisu** — nie da się odczytać, jaki kolor jest aktualnie ustawiony. Co więcej, wiele z nich trzeba odświeżać. System operacyjny rozwiązuje to następująco: dla najważniejszych rejestrów utrzymuje w pamięci RAM kopie zwane **cieniami** (ang. *shadow registers*). Pięćdziesiąt razy na sekundę, podczas przerwania wygaszania pionowego (VBI), system **przepisuje cienie do rejestrów sprzętowych**, a dla rejestrów wejściowych (joysticki) — odwrotnie.



Zapamiętaj

Zapisuj do cieni, nie do rejestrów sprzętowych — chyba że wiesz, dlaczego robisz inaczej. Wartość wpisana wprost do rejestru sprzętowego przetrwa najwyżej do najbliższego przerwania VBI, po czym system nadpisze ją zawartością cienia. Bezpośredni zapis do sprzętu stosuje się celowo wtedy, gdy zmiana ma obowiązywać tylko przez część ekranu (przerwania DLI, rozdział 24) albo gdy systemowe przerwanie VBI jest wyłączone.

19.3.1 Najważniejsze cienie

Cień	Adres	Dziesiętnie	Rejestr sprzętowy	Znaczenie
SDMCTL	`$022F`	559	DMACTL `$D400`	włączenie obrazu, szerokość pola, duszki
SDLSTL/H	`$0230`	560	DLISTL/H `$D402`	adres display list (2 bajty)
GPRIOR	`$026F`	623	PRIOR `$D01B`	priorytety duszków, tryby GTIA
PCOLR0–3	`$02C0`	704	COLPM0–3 `$D012`	kolory czterech graczy (duszków)
COLOR0	`$02C4`	708	COLPF0 `$D016`	kolor pola gry 0
COLOR1	`$02C5`	709	COLPF1 `$D017`	kolor pola 1 — w trybie tekstowym jasność liter
COLOR2	`$02C6`	710	COLPF2 `$D018`	kolor pola 2 — w trybie tekstowym tło
COLOR3	`$02C7`	711	COLPF3 `$D019`	kolor pola 3
COLOR4	`$02C8`	712	COLBK `$D01A`	kolor tła — w trybie tekstowym ramka
CHBAS	`$02F4`	756	CHBASE `$D409`	starszy bajt adresu zestawu znaków
CHACT	`$02F3`	755	CHACTL `$D401`	sposób wyświetlania znaków (negatyw, odwrócenie)
STICK0–1	`$0278`	632	PORTA `$D300`	położenie joysticków
STRIG0–1	`$0284`	644	TRIG0–1 `$D010`	przyciski joysticków

19.3.2 Ważne zmienne systemowe

Nazwa	Adres	Znaczenie
DOSVEC	`$0A`	adres powrotu do DOS-u (`jmp ($000A)`)
DOSINI	`$0C`	adres inicjalizacji DOS-u po RESET
RTCLOK	`$12–$14`	zegar: 3 bajty, `$14` rośnie co ramkę (uwaga: `$12` to bajt najstarszy!)
ATRACT	`$4D`	licznik wygaszacza ekranu; wpisanie 0 odsuwa wygaszenie
LMARGN, RMARGN	`$52`, `$53`	lewy i prawy margines ekranu tekstowego (domyślnie 2 i 39)
ROWCRS	`$54`	wiersz kursora
COLCRS	`$55`	kolumna kursora (2 bajty)
SAVMSC	`$58`	adres początku pamięci ekranu (2 bajty)
RAMTOP	`$6A`	szczyt pamięci RAM w stronach (`$C0` lub `$A0`)
VDSLST	`$0200`	wektor przerwania DLI
VVBLKI	`$0222`	wektor natychmiastowego przerwania VBI
VVBLKD	`$0224`	wektor opóźnionego przerwania VBI
CRSINH	`$02F0`	wartość różna od zera ukrywa kursor
CH	`$02FC`	kod ostatnio naciśniętego klawisza (`$FF` = brak)
RUNAD	`$02E0`	adres uruchomienia programu po załadowaniu
INITAD	`$02E2`	adres procedury wykonywanej w trakcie ładowania
MEMTOP	`$02E5`	górna granica wolnej pamięci
MEMLO	`$02E7`	dolna granica wolnej pamięci

Pełniejsza lista znajduje się w dodatku C.

19.4 Plik nagłówkowy z nazwami

Zamiast przepisywać adresy do każdego programu, warto mieć jeden plik z definicjami i dołączać go dyrektywą `icl`. Oto zestaw nazw używanych w dalszych rozdziałach. Zapisz go jako `atari.asm`:

Plik: atari.asm

```
; ------------------------------------------------------------------------
; atari.asm - nazwy adresów systemowych i sprzętowych Atari XL/XE
; Dołączanie:  icl 'atari.asm'   (na początku programu, przed dyrektywą org)
; ------------------------------------------------------------------------
```

```
; --- strona zerowa systemu ------------------------------------------------
DOSVEC  = $0A                   ; wektor powrotu do DOS-u
DOSINI  = $0C                   ; wektor inicjalizacji DOS-u
RTCLOK  = $12                   ; zegar systemowy (3 bajty; $14 = najmłodszy)
ATRACT  = $4D                   ; licznik wygaszacza ekranu
LMARGN  = $52                   ; lewy margines
RMARGN  = $53                   ; prawy margines
ROWCRS  = $54                   ; wiersz kursora
COLCRS  = $55                   ; kolumna kursora (2 bajty)
SAVMSC  = $58                   ; adres pamięci ekranu (2 bajty)
RAMTOP  = $6A                   ; szczyt pamięci RAM (numer strony)

; --- wektory przerwań -----------------------------------------------------
VDSLST  = $0200                 ; wektor DLI
VBREAK  = $0206                 ; wektor rozkazu BRK
VKEYBD  = $0208                 ; wektor przerwania klawiatury
VIMIRQ  = $0216                 ; główny wektor IRQ
CDTMV1  = $0218                 ; licznik systemowy 1 (2 bajty)
CDTMV2  = $021A                 ; licznik systemowy 2 (2 bajty)
VVBLKI  = $0222                 ; wektor VBI natychmiastowego
VVBLKD  = $0224                 ; wektor VBI opóźnionego
CDTMA1  = $0226                 ; adres procedury licznika 1
CDTMA2  = $0228                 ; adres procedury licznika 2

; --- rejestry-cienie ------------------------------------------------------
SDMCTL  = $022F                 ; cień DMACTL
SDLSTL  = $0230                 ; cień adresu display list (2 bajty)
GPRIOR  = $026F                 ; cień PRIOR
STICK0  = $0278                 ; joystick 0
STRIG0  = $0284                 ; przycisk joysticka 0
PCOLR0  = $02C0                 ; kolory graczy 0-3 (4 bajty)
COLOR0  = $02C4                 ; kolor pola 0
COLOR1  = $02C5                 ; kolor pola 1 (w trybie tekstowym: jasność liter)
COLOR2  = $02C6                 ; kolor pola 2 (w trybie tekstowym: tło)
COLOR3  = $02C7                 ; kolor pola 3
COLOR4  = $02C8                 ; kolor tła (w trybie tekstowym: ramka)
RUNAD   = $02E0                 ; adres uruchomienia programu
INITAD  = $02E2                 ; adres procedury inicjującej
MEMTOP  = $02E5                 ; górna granica wolnej pamięci
MEMLO   = $02E7                 ; dolna granica wolnej pamięci
CRSINH  = $02F0                 ; ukrycie kursora
CHACT   = $02F3                 ; cień CHACTL
CHBAS   = $02F4                 ; cień CHBASE
CH      = $02FC                 ; kod klawisza
BASICF  = $03F8                 ; znacznik wyłączenia BASIC-a

; --- blok sterujący urządzeń szeregowych (DCB) ----------------------------
DDEVIC  = $0300                 ; identyfikator urządzenia
DUNIT   = $0301                 ; numer urządzenia
DCOMND  = $0302                 ; rozkaz
DSTATS  = $0303                 ; kierunek transmisji / status
DBUFLO  = $0304                 ; adres bufora: młodszy bajt
DBUFHI  = $0305                 ; adres bufora: starszy bajt
DTIMLO  = $0306                 ; limit czasu
DBYTLO  = $0308                 ; liczba bajtów: młodszy bajt
DBYTHI  = $0309                 ; liczba bajtów: starszy bajt
DAUX1   = $030A                 ; parametr 1 (np. młodszy bajt numeru sektora)
DAUX2   = $030B                 ; parametr 2 (np. starszy bajt numeru sektora)

; --- bloki sterujące wejścia/wyjścia (IOCB) - adresy dla IOCB numer 0 -------
ICHID   = $0340                 ; identyfikator urządzenia
ICDNO   = $0341                 ; numer urządzenia
ICCOM   = $0342                 ; rozkaz
ICSTA   = $0343                 ; status
ICBAL   = $0344                 ; adres bufora: młodszy bajt
ICBAH   = $0345                 ; adres bufora: starszy bajt
ICBLL   = $0348                 ; długość bufora: młodszy bajt
ICBLH   = $0349                 ; długość bufora: starszy bajt
ICAX1   = $034A                 ; parametr 1
ICAX2   = $034B                 ; parametr 2

; --- GTIA -----------------------------------------------------------------
```

```
HPOSP0  = $D000                 ; pozycja pozioma gracza 0 (zapis)
SIZEP0  = $D008                 ; szerokość gracza 0 (zapis)
TRIG0   = $D010                 ; przycisk joysticka 0 (odczyt)
COLPM0  = $D012                 ; kolor gracza 0
COLPF0  = $D016                 ; kolor pola gry 0
COLPF1  = $D017                 ; kolor pola gry 1
COLPF2  = $D018                 ; kolor pola gry 2
COLPF3  = $D019                 ; kolor pola gry 3
COLBK   = $D01A                 ; kolor tła
PRIOR   = $D01B                 ; priorytety
GRACTL  = $D01D                 ; sterowanie grafiką PMG
HITCLR  = $D01E                 ; kasowanie rejestrów kolizji
CONSOL  = $D01F                 ; klawisze START, SELECT, OPTION

; --- POKEY ---------------------------------------------------------------
AUDF1   = $D200                 ; częstotliwość kanału 1
AUDC1   = $D201                 ; głośność i barwa kanału 1
AUDF2   = $D202                 ; częstotliwość kanału 2
AUDC2   = $D203                 ; głośność i barwa kanału 2
AUDF3   = $D204                 ; częstotliwość kanału 3
AUDC3   = $D205                 ; głośność i barwa kanału 3
AUDF4   = $D206                 ; częstotliwość kanału 4
AUDC4   = $D207                 ; głośność i barwa kanału 4
AUDCTL  = $D208                 ; sterowanie dźwiękiem
KBCODE  = $D209                 ; kod klawisza (odczyt)
RANDOM  = $D20A                 ; liczba losowa (odczyt)
IRQEN   = $D20E                 ; maska przerwań IRQ (zapis)
SKCTL   = $D20F                 ; sterowanie portem szeregowym (zapis)
SKSTAT  = $D20F                 ; status klawiatury i portu (odczyt)

; --- PIA -----------------------------------------------------------------
PORTA   = $D300                 ; joysticki
PORTB   = $D301                 ; sterowanie pamięcią

; --- ANTIC ---------------------------------------------------------------
DMACTL  = $D400                 ; sterowanie obrazem
CHACTL  = $D401                 ; sterowanie znakami
DLISTL  = $D402                 ; adres display list (2 bajty)
HSCROL  = $D404                 ; płynny przesuw poziomy
VSCROL  = $D405                 ; płynny przesuw pionowy
PMBASE  = $D407                 ; adres pamięci duszków
CHBASE  = $D409                 ; adres zestawu znaków
WSYNC   = $D40A                 ; zapis: zatrzymaj procesor do końca linii
VCOUNT  = $D40B                 ; odczyt: numer linii obrazu / 2
NMIEN   = $D40E                 ; zapis: maska przerwań NMI
NMIST   = $D40F                 ; odczyt: przyczyna NMI
NMIRES  = $D40F                 ; zapis: skasowanie NMIST

; --- procedury systemu operacyjnego --------------------------------------
DSKINV  = $E453                 ; obsługa stacji dysków (sektory)
CIOV    = $E456                 ; centralna procedura wejścia/wyjścia
SIOV    = $E459                 ; transmisja szeregowa
SETVBV  = $E45C                 ; ustawianie wektorów VBI i liczników
SYSVBV  = $E45F                 ; systemowa obsługa VBI (wyjście z VBI natychmiastowego)
XITVBV  = $E462                 ; zakończenie przerwania VBI (wyjście z VBI opóźnionego)
WARMSV  = $E474                 ; ciepły start (jak klawisz RESET)
COLDSV  = $E477                 ; zimny start (jak włączenie komputera)

EOL     = $9B                   ; znak końca linii
```

W kompletnych programach tej książki potrzebne adresy są dla wygody wypisane na początku każdego listingu — dzięki temu każdy program można zasemblować samodzielnie.

ROZDZIAŁ 20

Plik wykonywalny: segmenty, RUN i INIT

Plik, który MADS tworzy z twojego programu, ma prosty i elegancki format — ten sam, którego używa DOS Atari od 1979 roku. Zwyczajowo nadaje mu się rozszerzenie `.xex`, `.com` lub `.obx`. Warto go znać dokładnie, bo pozwala na ciekawe sztuczki: wykonywanie kodu w trakcie ładowania, ładowanie danych wprost do banków pamięci, ekrany tytułowe pokazywane, zanim wczyta się reszta programu.

20.1 Budowa pliku

Plik składa się z **segmentów** (bloków). Każdy segment to nagłówek i dane:

```
┌──────────┬──────────────┬──────────────┬──────────────────────┐
│ $FF $FF  │ adres początku│ adres końca  │ dane segmentu        │
│ (2 b.)   │ (2 b., LO HI) │ (2 b., LO HI)│ (koniec-początek+1 baj.)│
└──────────┴──────────────┴──────────────┴──────────────────────┘
```

- Dwa bajty `$FF $FF` to znacznik pliku binarnego. Są **obowiązkowe na początku pliku**, a przed kolejnymi segmentami — opcjonalne.
- Adres początku i adres końca mówią, gdzie w pamięci umieścić dane.
- Dalej następują same dane.

Program ładujący (DOS) czyta segment po segmencie i umieszcza dane pod wskazanymi adresami. Nie ma tu żadnej relokacji, sum kontrolnych ani nagłówka z nazwą — tylko „te bajty pod ten adres".

Oto pełna zawartość pliku `pierwszy.xex` z rozdziału 1 (40 bajtów):

```
FF FF 00 20 1B 20               nagłówek: segment od $2000 do $201B (28 bajtów)
A9 94 8D C6 02 A9 46 8D C8 02   lda #$94 / sta $02C6 / lda #$46 / sta $02C8
A9 FF 8D FC 02                  lda #$FF / sta $02FC
AD FC 02 C9 FF F0 F9            lda $02FC / cmp #$FF / beq czekaj
A9 FF 8D FC 02 60               lda #$FF / sta $02FC / rts
E0 02 E1 02                     nagłówek: segment od $02E0 do $02E1 (2 bajty)
00 20                           dane: adres $2000 - trafia do wektora RUNAD
```

Każda dyrektywa `org` w źródle zaczyna nowy segment. Drugi segment tego pliku ma tylko dwa bajty — i to on sprawia, że program się uruchamia.

20.2 Wektor RUNAD — uruchomienie po załadowaniu

Zanim DOS zacznie ładować plik, zapamiętuje w komórkach RUNAD (`$02E0/$02E1`) adres własnej procedury powrotu. Po załadowaniu **całego** pliku wykonuje skok pod adres zapisany w RUNAD. Jeżeli plik zawierał segment, który wpisał tam coś innego — program zostaje uruchomiony:

```
        org $02E0               ; segment dwubajtowy pod adresem RUNAD
        dta a(start)            ; adres uruchomienia
```

Plik bez takiego segmentu po prostu załaduje się do pamięci i DOS wróci do swojego menu. Bywa to przydatne: tak ładuje się dane lub procedury pomocnicze.

20.3 Wektor INITAD — wykonanie w trakcie ładowania

Drugi wektor, INITAD (`$02E2/$02E3`), działa inaczej: DOS sprawdza go **po każdym załadowanym segmencie**. Jeśli segment zmienił zawartość INITAD, DOS natychmiast wywołuje wskazaną procedurę (rozkazem `jsr`), a po jej powrocie (`rts`) kontynuuje ładowanie następnych segmentów.

```
kolejność zdarzeń podczas ładowania pliku:

1. załaduj segment A (kod procedury inicjującej)
2. załaduj segment B (2 bajty pod $02E2)  ->  DOS wykonuje: jsr (INITAD)
3. załaduj segment C (główna część programu)
4. załaduj segment D (2 bajty pod $02E0)
5. koniec pliku                           ->  DOS wykonuje: jmp (RUNAD)
```

Procedur INIT może być w pliku dowolnie wiele. Typowe zastosowania:

- wyświetlenie ekranu tytułowego na czas wczytywania reszty programu,
- wyłączenie BASIC-a, zanim załaduje się kod przeznaczony dla obszaru `$A000–$BFFF`,
- przełączenie banku pamięci, żeby następny segment załadował się wprost do pamięci dodatkowej (rozdział 28),
- sprawdzenie wymagań (ilość pamięci, system PAL/NTSC) i przerwanie ładowania, gdy nie są spełnione.

20.3.1 Przykład: komunikat podczas ładowania

Plik: init.asm

```
; ------------------------------------------------------------------------
; init.asm - demonstracja segmentu INIT: komunikat wypisywany w trakcie ładowania
; ------------------------------------------------------------------------
ICCOM   = $0342                 ; IOCB 0: rozkaz
ICBAL   = $0344                 ; IOCB 0: adres bufora (młodszy bajt)
ICBAH   = $0345                 ; IOCB 0: adres bufora (starszy bajt)
ICBLL   = $0348                 ; IOCB 0: długość bufora (młodszy bajt)
ICBLH   = $0349                 ; IOCB 0: długość bufora (starszy bajt)
CIOV    = $E456                 ; procedura wejścia/wyjścia systemu
CH      = $02FC                 ; kod klawisza

; === SEGMENT 1: procedura inicjująca ====================================
        org $0600               ; strona 6 - krótka procedura na czas ładowania

powitanie ldx #0                ; X = 0: IOCB numer 0 (edytor ekranowy)
        lda #9                  ; rozkaz 9: wypisz wiersz tekstu (do znaku EOL)
        sta ICCOM               ; zapisz A do ICCOM - rozkaz
        lda #<tekst1            ; adres tekstu
        sta ICBAL               ; zapisz A do ICBAL - adres bufora: młodszy bajt
        lda #>tekst1            ; A = starszy bajt adresu tekst1
        sta ICBAH               ; zapisz A do ICBAH - adres bufora: starszy bajt
        lda #40                 ; maksymalna długość wiersza
        sta ICBLL               ; zapisz A do ICBLL - długość bufora: młodszy bajt
        lda #0                  ; A = 0
        sta ICBLH               ; zapisz A do ICBLH - długość bufora: starszy bajt
        jsr CIOV                ; wywołaj system
        rts                     ; wróć do DOS-u - ładowanie potoczy się dalej

tekst1  dta c'Trwa wczytywanie...',$9B  ; tekst w kodach ATASCII

; === SEGMENT 2: wektor INITAD - DOS wykona "powitanie" natychmiast ==========
        org $02E2               ; wektor INITAD: procedura wykonywana podczas ładowania
        dta a(powitanie)        ; adres procedury: powitanie

; === SEGMENT 3: program główny ==========================================
        org $2000               ; adres asemblacji: $2000

start   ldx #0                  ; wypisz drugi komunikat - tak samo jak pierwszy
        lda #9                  ; A = 9
        sta ICCOM               ; zapisz A do ICCOM - rozkaz
        lda #<tekst2            ; A = młodszy bajt adresu tekst2
        sta ICBAL               ; zapisz A do ICBAL - adres bufora: młodszy bajt
        lda #>tekst2            ; A = starszy bajt adresu tekst2
        sta ICBAH               ; zapisz A do ICBAH - adres bufora: starszy bajt
        lda #40                 ; A = 40
        sta ICBLL               ; zapisz A do ICBLL - długość bufora: młodszy bajt
        lda #0                  ; A = 0
        sta ICBLH               ; zapisz A do ICBLH - długość bufora: starszy bajt
        jsr CIOV                ; wywołaj CIOV: centralna procedura wejścia/wyjścia
        lda #$FF                ; poczekaj na klawisz
```

```
        sta CH                  ; zapisz A do CH - kod klawisza
czekaj  lda CH                  ; A = CH - kod klawisza
        cmp #$FF                ; porównaj A z: $FF
        beq czekaj              ; skocz do czekaj, gdy Z = 1 (równe / zero)
        lda #$FF                ; skasuj klawisz
        sta CH                  ; zapisz A do CH - kod klawisza
        rts                     ; powrót do DOS-u

tekst2  dta c'Program gotowy. Nacisnij klawisz.',$9B  ; tekst w kodach ATASCII

; === SEGMENT 4: wektor RUNAD - uruchomienie po załadowaniu całości ==========
        org $02E0               ; wektor RUNAD: adres uruchomienia programu
        dta a(start)            ; adres uruchomienia: start
```

W tak krótkim pliku oba komunikaty pojawią się niemal jednocześnie; przy programie ładującym się kilkanaście sekund z dyskietki różnica jest wyraźna.

> **Uwaga**
>
> Procedura INIT działa w chwili, gdy DOS jest w trakcie czytania pliku. Wolno jej korzystać z ekranu (IOCB 0), ale **nie powinna otwierać ani zamykać plików dyskowych** i musi zakończyć się rozkazem `rts` z nienaruszonym stosem. Nie może też zamazać pamięci DOS-u ani obszaru, do którego załadują się następne segmenty.

20.4 Zapis skrócony w MADS

MADS ma dwie dyrektywy, które tworzą opisane wyżej dwubajtowe segmenty:

```
        run start               ; to samo co:  org $02E0  /  dta a(start)
        ini powitanie           ; to samo co:  org $02E2  /  dta a(powitanie)
```

Są wygodne i powszechnie stosowane. W tej książce konsekwentnie używamy zapisu pełnego, żeby mechanizm był widoczny.

20.5 Pliki bez nagłówków

Dyrektywa `opt h-` wyłącza generowanie nagłówków segmentów. MADS zapisuje wtedy **surowy obraz pamięci** — same bajty kodu, jeden za drugim. Tak przygotowuje się dane, które nie są ładowane przez DOS:

- sektory startowe dyskietki (rozdział 27),
- zawartość kartridża,
- bloki danych dołączane do innych programów dyrektywą `ins`.

```
        opt h-                  ; bez nagłówków: plik = surowe bajty
        org $0700               ; adres, pod którym kod BĘDZIE działał
start   lda #0                  ; pierwszy bajt pliku to kod tego rozkazu
```

W pliku bez nagłówków może być tylko jeden ciągły obszar adresów; dyrektywa `org` nie zapisuje nigdzie informacji o adresie — służy jedynie asemblerowi do obliczania etykiet.

20.6 Dobre obyczaje

1. **Zaczynaj od `$2000`** albo wyżej, jeżeli nie masz powodu, by robić inaczej. Poniżej może rezydować DOS.
2. **Nie zakładaj, gdzie jest ekran.** Zawsze czytaj SAVMSC.
3. **Kończ program rozkazem `rts`** (powrót do DOS-u) lub skokiem `jmp ($000A)` przez wektor DOSVEC. Programy, które nie zamierzają wracać — na przykład gry — kończą się zwykle pętlą nieskończoną albo zimnym startem `jmp $E477`.
4. **Przywróć to, co zmieniłeś**, jeśli wracasz do DOS-u: wektory przerwań, kolory, zestaw znaków, tryb graficzny, wyłączone przerwania.

5. **Sprawdzaj w listingu zakresy segmentów.** MADS wypisuje je w pierwszych kolumnach pliku `.lst`. Dwa segmenty nachodzące na siebie to błąd, którego asembler nie zgłosi — późniejszy segment po prostu nadpisze wcześniejszy.

ROZDZIAŁ 21

System operacyjny: wejście i wyjście przez CIO

System operacyjny Atari ma jeden, wspólny mechanizm obsługi wszystkich urządzeń: ekranu, klawiatury, stacji dysków, drukarki, magnetofonu. Nazywa się **CIO** (*Central Input/Output*). Niezależnie od urządzenia program robi zawsze to samo: wypełnia blok parametrów i wywołuje jedną procedurę pod adresem `$E456`.

Dzięki temu ten sam kod może wypisywać tekst na ekran, do pliku na dyskietce albo na drukarkę — zmienia się tylko nazwa urządzenia.

21.1 Bloki IOCB

System udostępnia osiem „kanałów" wejścia/wyjścia. Każdy opisuje 16-bajtowy blok **IOCB** (*Input/Output Control Block*). Bloki leżą jeden za drugim od adresu `$0340`:

Numer IOCB	Adres	Wartość X	Zwyczajowe przeznaczenie
0	`$0340`	`$00`	edytor ekranowy `E:` — otwarty przez system na stałe
1	`$0350`	`$10`	wolny
2	`$0360`	`$20`	wolny
3	`$0370`	`$30`	wolny
4	`$0380`	`$40`	wolny
5	`$0390`	`$50`	wolny
6	`$03A0`	`$60`	ekran graficzny `S:` (tak używa go BASIC)
7	`$03B0`	`$70`	wolny (BASIC: LPRINT, LOAD, SAVE)

Numer kanału przekazuje się procedurze CIO w **rejestrze X, pomnożony przez 16**. To sprytne: ta sama wartość służy od razu jako indeks do pól bloku — `sta ICCOM,x` zapisuje rozkaz do właściwego IOCB.

Pola bloku, które wypełnia program:

Pole	Adres dla IOCB 0	Znaczenie
ICCOM	`$0342`	rozkaz (co ma zostać wykonane)
ICSTA	`$0343`	status ostatniej operacji (wypełnia system)
ICBAL, ICBAH	`$0344`, `$0345`	adres bufora danych albo nazwy pliku
ICBLL, ICBLH	`$0348`, `$0349`	długość bufora; po operacji — liczba faktycznie przesłanych bajtów
ICAX1	`$034A`	parametr 1 (przy otwarciu: kierunek)
ICAX2	`$034B`	parametr 2 (zależny od urządzenia)

Rozkazy:

Kod	Nazwa	Działanie
3	OPEN	otwarcie kanału dla urządzenia lub pliku
5	GET RECORD	odczyt wiersza tekstu — do znaku EOL włącznie
7	GET CHARACTERS	odczyt zadanej liczby bajtów

Kod	Nazwa	Działanie
9	PUT RECORD	zapis wiersza tekstu — do znaku EOL włącznie
11	PUT CHARACTERS	zapis zadanej liczby bajtów
12	CLOSE	zamknięcie kanału
13	STATUS	odczyt stanu urządzenia

Nazwy urządzeń to jedna litera z dwukropkiem, ewentualnie z numerem: `E:` edytor ekranowy, `K:` klawiatura, `S:` ekran, `P:` drukarka, `C:` magnetofon, `D:` lub `D1:`...`D8:` stacje dysków.

21.1.1 Wynik operacji

Po powrocie z CIO:

- **rejestr Y** zawiera status: 1 oznacza powodzenie, wartości **od 128 wzwyż oznaczają błąd**,
- **znacznik N** jest ustawiony zgodnie z Y, więc po `jsr CIOV` wystarczy `bmi blad`,
- rejestr X zachowuje numer kanału, zawartość A zależy od operacji.

Status	Znaczenie
1	operacja udana
3	następny odczyt napotka koniec pliku (ostrzeżenie, nie błąd)
128	naciśnięto klawisz BREAK
129	kanał jest już otwarty
130	nieznane urządzenie
133	kanał nie jest otwarty
136	koniec pliku
137	wiersz dłuższy niż bufor (obcięty)
138	urządzenie nie odpowiada
144	błąd wykonania rozkazu przez urządzenie (np. dyskietka zabezpieczona)
162	dyskietka pełna
165	błędna nazwa pliku
167	plik zabezpieczony
170	nie znaleziono pliku

21.2 Wypisywanie tekstu

Kanał 0 jest zawsze otwarty dla edytora ekranowego, więc wypisywanie nie wymaga żadnych przygotowań. Rozkaz PUT RECORD wypisuje znaki aż do EOL (`$9B`) i przechodzi do nowego wiersza.

Plik: witaj.asm

```
; ------------------------------------------------------------------------
; witaj.asm - wypisanie tekstu na ekranie przez CIO
; ------------------------------------------------------------------------
ICCOM   = $0342                 ; pola bloku IOCB numer 0
ICBAL   = $0344                 ; adres bufora: młodszy bajt
ICBAH   = $0345                 ; adres bufora: starszy bajt
ICBLL   = $0348                 ; długość bufora: młodszy bajt
ICBLH   = $0349                 ; długość bufora: starszy bajt
CIOV    = $E456                 ; wejście do procedury CIO
EOL     = $9B                   ; znak końca linii

        org $2000               ; adres asemblacji: $2000

start   ldx #$00                ; X = numer kanału * 16; kanał 0 to edytor ekranowy
        lda #9                  ; rozkaz 9 = PUT RECORD: wypisz wiersz zakończony EOL
        sta ICCOM,x             ; zapisz A pod adres ICCOM+X - rozkaz
        lda #<tekst             ; adres tekstu do wypisania
```

```
        sta ICBAL,x             ; zapisz A pod adres ICBAL+X - adres bufora: młodszy bajt
        lda #>tekst             ; A = starszy bajt adresu tekst
        sta ICBAH,x             ; zapisz A pod adres ICBAH+X - adres bufora: starszy bajt
        lda #<255               ; maksymalna długość: wypisywanie skończy się na znaku EOL,
        sta ICBLL,x             ; a jeśli go zabraknie - najpóźniej po 255 znakach
        lda #>255               ; A = starszy bajt wartości 255
        sta ICBLH,x             ; zapisz A pod adres ICBLH+X - długość bufora: starszy bajt
        jsr CIOV                ; wykonaj operację
        rts                     ; powrót do DOS-u

tekst   dta c'Witaj w swiecie asemblera!',EOL  ; tekst w kodach ATASCII

        org $02E0               ; wektor RUNAD: adres uruchomienia programu
        dta a(start)            ; adres uruchomienia: start
```

Zapis `sta ICCOM,x` z X = 0 trafia w `$0342`. Gdybyśmy chcieli wypisać ten sam tekst do pliku otwartego w kanale 1, wystarczyłoby zmienić pierwszy rozkaz na `ldx #$10`.

> **Uwaga**
>
> Teksty w programach dla Atari piszemy **bez polskich znaków diakrytycznych**. Standardowy zestaw znaków ATASCII ich nie zawiera, a litery „ą" czy „ś" wpisane w edytorze na PC zostałyby zamienione na przypadkowe bajty. Polskie litery wymagają własnego zestawu znaków (rozdział 22).

21.2.1 Biblioteka: znak, napis, nowy wiersz

W praktyce warto mieć trzy małe procedury. Pierwsza wypisuje jeden znak, korzystając ze szczególnego przypadku: gdy długość bufora wynosi zero, PUT CHARACTERS wypisuje znak **z akumulatora**.

Plik: cio_ekran.asm

```
; ---------------------------------------------------------------------
; cio_ekran.asm - podstawowe procedury wypisywania na ekran przez CIO
; ---------------------------------------------------------------------
ICCOM   = $0342                 ; rozkaz
ICBAL   = $0344                 ; adres bufora: młodszy bajt
ICBAH   = $0345                 ; adres bufora: starszy bajt
ICBLL   = $0348                 ; długość bufora: młodszy bajt
ICBLH   = $0349                 ; długość bufora: starszy bajt
CIOV    = $E456                 ; centralna procedura wejścia/wyjścia
EOL     = $9B                   ; znak końca linii

; putchar - wypisuje znak z akumulatora.  Niszczy: A, X, Y.
putchar ldx #$00                ; kanał 0: edytor ekranowy
        pha                     ; zachowaj znak (A będzie potrzebny do ustawienia parametrów)
        lda #11                 ; rozkaz 11 = PUT CHARACTERS
        sta ICCOM,x             ; zapisz A pod adres ICCOM+X - rozkaz
        lda #0                  ; długość bufora = 0: znak zostanie pobrany z rejestru A
        sta ICBLL,x             ; zapisz A pod adres ICBLL+X - długość bufora: młodszy bajt
        sta ICBLH,x             ; zapisz A pod adres ICBLH+X - długość bufora: starszy bajt
        pla                     ; odtwórz znak
        jmp CIOV                ; wywołaj CIO i wróć bezpośrednio do wywołującego

; print - wypisuje wiersz tekstu zakończony znakiem EOL.
; Wejście: A = młodszy bajt adresu tekstu, Y = starszy bajt.  Niszczy: A, X, Y.
print   ldx #$00                ; kanał 0
        sta ICBAL,x             ; adres tekstu: młodszy bajt z A
        tya                     ; A = Y
        sta ICBAH,x             ; starszy bajt z Y
        lda #9                  ; rozkaz 9 = PUT RECORD
        sta ICCOM,x             ; zapisz A pod adres ICCOM+X - rozkaz
        lda #255                ; maksymalna długość 255 znaków
        sta ICBLL,x             ; zapisz A pod adres ICBLL+X - długość bufora: młodszy bajt
        lda #0                  ; A = 0
        sta ICBLH,x             ; zapisz A pod adres ICBLH+X - długość bufora: starszy bajt
        jmp CIOV                ; skocz do CIOV - centralna procedura wejścia/wyjścia

; nowy_wiersz - przenosi kursor na początek następnego wiersza.  Niszczy: A, X, Y.
nowy_wiersz lda #EOL            ; wypisanie znaku EOL to przejście do nowej linii
        jmp putchar             ; skocz do putchar
```

Wywołanie: `lda #<tekst` / `ldy #>tekst` / `jsr print`.

21.2.2 Znaki sterujące

Edytor ekranowy interpretuje niektóre kody jako polecenia. Wystarczy je „wypisać”:

Kod	Dziesiętnie	Działanie
`$9B`	155	koniec linii — przejście na początek następnego wiersza
`$7D`	125	wyczyszczenie ekranu, kursor w lewym górnym rogu
`$1C–$1F`	28–31	kursor w górę, w dół, w lewo, w prawo
`$7E`	126	cofnięcie z kasowaniem znaku (Backspace)
`$7F`	127	tabulacja
`$FD`	253	sygnał dźwiękowy
`$1B`	27	ESC: następny znak zostanie wyświetlony, a nie wykonany

Aby ustawić kursor w dowolnym miejscu, przed wypisaniem zapisz jego współrzędne do zmiennych systemowych: `ROWCRS` (`$54`, wiersz 0–23) i `COLCRS` (`$55`, kolumna 0–39):

```
        lda #10                 ; wiersz 10
        sta $54                 ; ROWCRS
        lda #5                  ; kolumna 5
        sta $55                 ; COLCRS (młodszy bajt)
        lda #0                  ; A = 0
        sta $56                 ; COLCRS (starszy bajt - zawsze 0 w trybie tekstowym)
        lda #<tekst             ; wypisz tekst od tej pozycji
        ldy #>tekst             ; Y = starszy bajt adresu tekst
        jsr print               ; wywołaj podprogram print
```

21.3 Wczytywanie tekstu z klawiatury

Rozkaz GET RECORD na kanale 0 pozwala użytkownikowi wpisać wiersz tekstu z pełną edycją (kursory, kasowanie); kończy go klawisz RETURN. Do bufora trafia wpisany tekst zakończony znakiem EOL, a w polach ICBLL/ICBLH system zostawia jego długość (wraz z EOL).

```
; Wczytuje wiersz tekstu do "bufor" (najwyżej 40 znaków łącznie z EOL)
        ldx #$00                ; kanał 0: edytor ekranowy
        lda #5                  ; rozkaz 5 = GET RECORD: wczytaj wiersz
        sta ICCOM,x             ; zapisz A pod adres ICCOM+X - rozkaz
        lda #<bufor             ; adres bufora na tekst
        sta ICBAL,x             ; zapisz A pod adres ICBAL+X - adres bufora: młodszy bajt
        lda #>bufor             ; A = starszy bajt adresu bufor
        sta ICBAH,x             ; zapisz A pod adres ICBAH+X - adres bufora: starszy bajt
        lda #40                 ; rozmiar bufora: system nie zapisze więcej
        sta ICBLL,x             ; zapisz A pod adres ICBLL+X - długość bufora: młodszy bajt
        lda #0                  ; A = 0
        sta ICBLH,x             ; zapisz A pod adres ICBLH+X - długość bufora: starszy bajt
        jsr CIOV                ; czekaj na wpisanie tekstu i klawisz RETURN
        bmi blad                ; Y >= 128: błąd (np. 128 = naciśnięto BREAK)
        lda ICBLL               ; A = liczba wczytanych znaków, wliczając EOL
```

21.4 Pliki dyskowe

Praca z plikiem to trzy kroki: **otwarcie**, **odczyt lub zapis**, **zamknięcie**. Przy otwarciu podajemy nazwę (zakończoną EOL) oraz kierunek w polu ICAX1:

ICAX1	Tryb otwarcia
4	odczyt
8	zapis (istniejący plik zostanie zastąpiony)
9	dopisywanie na końcu
12	odczyt i zapis (aktualizacja)
6	odczyt katalogu dyskietki

Pułapka

Zawsze zamykaj plik otwarty do zapisu. Dopiero zamknięcie zapisuje na dyskietkę ostatni, niepełny sektor danych i aktualizuje katalog. Plik niezamknięty jest uszkodzony albo pusty.

21.4.1 Program: wyświetlenie pliku tekstowego

Poniższy program łączy wszystko, co do tej pory poznaliśmy: pyta o nazwę pliku, otwiera go, czyta porcjami po 128 bajtów i wypisuje na ekran, a w razie błędu pokazuje jego numer.

Plik: pokaz.asm

```
; ------------------------------------------------------------------------
; pokaz.asm - wyświetla zawartość pliku tekstowego
; Po uruchomieniu wpisz nazwę z urządzeniem, np.  D:LIST.TXT
; ------------------------------------------------------------------------
ICCOM   = $0342                 ; pola IOCB; dla kanału N dodajemy N*16 (przez rejestr X)
ICBAL   = $0344                 ; adres bufora: młodszy bajt
ICBAH   = $0345                 ; adres bufora: starszy bajt
ICBLL   = $0348                 ; długość bufora: młodszy bajt
ICBLH   = $0349                 ; długość bufora: starszy bajt
ICAX1   = $034A                 ; parametr 1
ICAX2   = $034B                 ; parametr 2
CIOV    = $E456                 ; centralna procedura wejścia/wyjścia
CH      = $02FC                 ; kod klawisza
EOL     = $9B                   ; znak końca linii

KANAL   = $10                   ; kanał 1 (numer * 16) - dla pliku
ROZM    = 128                   ; rozmiar porcji odczytu w bajtach

status  = $80                   ; zapamiętany status operacji odczytu
setki   = $81                   ; zmienne pomocnicze procedury wypisującej liczbę
dzies   = $82                   ; (cyfra dziesiątek)

        org $2000               ; adres asemblacji: $2000

start   lda #<pytanie           ; wypisz pytanie o nazwę pliku
        ldy #>pytanie           ; Y = starszy bajt adresu pytanie
        jsr print               ; wywołaj podprogram print

; --- wczytaj nazwę pliku z klawiatury -----------------------------------
        ldx #$00                ; kanał 0: edytor
        lda #5                  ; GET RECORD
        sta ICCOM,x             ; zapisz A pod adres ICCOM+X - rozkaz
        lda #<nazwa             ; A = młodszy bajt adresu nazwa
        sta ICBAL,x             ; zapisz A pod adres ICBAL+X - adres bufora: młodszy bajt
        lda #>nazwa             ; A = starszy bajt adresu nazwa
        sta ICBAH,x             ; zapisz A pod adres ICBAH+X - adres bufora: starszy bajt
        lda #40                 ; najwyżej 40 znaków
        sta ICBLL,x             ; zapisz A pod adres ICBLL+X - długość bufora: młodszy bajt
        lda #0                  ; A = 0
        sta ICBLH,x             ; zapisz A pod adres ICBLH+X - długość bufora: starszy bajt
        jsr CIOV                ; wywołaj CIOV: centralna procedura wejścia/wyjścia
        bpl otworz              ; bez błędu: otwieraj plik
        jmp blad                ; skocz do blad

; --- otwórz plik do odczytu ---------------------------------------------
otworz  ldx #KANAL              ; kanał 1
        lda #12                 ; najpierw CLOSE - na wypadek, gdyby kanał był otwarty
        sta ICCOM,x             ; zapisz A pod adres ICCOM+X - rozkaz
        jsr CIOV                ; (ewentualny błąd zamykania ignorujemy)
```

```
        ldx #KANAL              ; X = KANAL
        lda #3                  ; rozkaz 3 = OPEN
        sta ICCOM,x             ; zapisz A pod adres ICCOM+X - rozkaz
        lda #<nazwa             ; adres nazwy pliku (zakończonej EOL - tak wpisał ją edytor)
        sta ICBAL,x             ; zapisz A pod adres ICBAL+X - adres bufora: młodszy bajt
        lda #>nazwa             ; A = starszy bajt adresu nazwa
        sta ICBAH,x             ; zapisz A pod adres ICBAH+X - adres bufora: starszy bajt
        lda #4                  ; ICAX1 = 4: otwarcie do odczytu
        sta ICAX1,x             ; zapisz A pod adres ICAX1+X - parametr 1
        lda #0                  ; ICAX2 = 0
        sta ICAX2,x             ; zapisz A pod adres ICAX2+X - parametr 2
        jsr CIOV                ; wywołaj CIOV: centralna procedura wejścia/wyjścia
        bmi blad                ; nie udało się otworzyć (np. 170 = brak pliku)

; --- czytaj plik porcjami i wypisuj na ekran ------------------------------
czytaj  ldx #KANAL              ; X = KANAL
        lda #7                  ; rozkaz 7 = GET CHARACTERS: czytaj zadaną liczbę bajtów
        sta ICCOM,x             ; zapisz A pod adres ICCOM+X - rozkaz
        lda #<bufor             ; A = młodszy bajt adresu bufor
        sta ICBAL,x             ; zapisz A pod adres ICBAL+X - adres bufora: młodszy bajt
        lda #>bufor             ; A = starszy bajt adresu bufor
        sta ICBAH,x             ; zapisz A pod adres ICBAH+X - adres bufora: starszy bajt
        lda #<ROZM              ; chcemy przeczytać ROZM bajtów
        sta ICBLL,x             ; zapisz A pod adres ICBLL+X - długość bufora: młodszy bajt
        lda #>ROZM              ; A = starszy bajt wartości ROZM
        sta ICBLH,x             ; zapisz A pod adres ICBLH+X - długość bufora: starszy bajt
        jsr CIOV                ; wywołaj CIOV: centralna procedura wejścia/wyjścia
        sty status              ; zapamiętaj status (może to być koniec pliku)

        lda ICBLL+KANAL         ; ile bajtów faktycznie przeczytano?
        beq po_druku            ; zero: nie ma czego wypisywać
        ldx #$00                ; kanał 0: ekran
        sta ICBLL,x             ; długość = liczba przeczytanych bajtów
        lda #0                  ; A = 0
        sta ICBLH,x             ; zapisz A pod adres ICBLH+X - długość bufora: starszy bajt
        lda #11                 ; rozkaz 11 = PUT CHARACTERS
        sta ICCOM,x             ; zapisz A pod adres ICCOM+X - rozkaz
        lda #<bufor             ; A = młodszy bajt adresu bufor
        sta ICBAL,x             ; zapisz A pod adres ICBAL+X - adres bufora: młodszy bajt
        lda #>bufor             ; A = starszy bajt adresu bufor
        sta ICBAH,x             ; zapisz A pod adres ICBAH+X - adres bufora: starszy bajt
        jsr CIOV                ; wypisz porcję na ekran

po_druku lda status             ; sprawdź status odczytu
        bpl czytaj              ; poniżej 128: wszystko w porządku, czytaj dalej
        cmp #136                ; 136 = koniec pliku - to normalne zakończenie
        beq zamknij             ; skocz do zamknij, gdy Z = 1 (równe / zero)
        tay                     ; inny błąd: przekaż jego numer w Y
        jmp blad                ; skocz do blad

; --- zamknij plik i zakończ -----------------------------------------------
zamknij ldx #KANAL              ; X = KANAL
        lda #12                 ; rozkaz 12 = CLOSE
        sta ICCOM,x             ; zapisz A pod adres ICCOM+X - rozkaz
        jsr CIOV                ; wywołaj CIOV: centralna procedura wejścia/wyjścia
koniec  lda #EOL                ; przejdź do nowego wiersza (plik mógł nie kończyć się znakiem EOL)
        jsr putchar             ; wywołaj podprogram putchar
        lda #<nacisnij          ; komunikat końcowy
        ldy #>nacisnij          ; Y = starszy bajt adresu nacisnij
        jsr print               ; wywołaj podprogram print
        lda #$FF                ; czekaj na dowolny klawisz
        sta CH                  ; zapisz A do CH - kod klawisza
czekaj  lda CH                  ; A = CH - kod klawisza
        cmp #$FF                ; porównaj A z: $FF
        beq czekaj              ; skocz do czekaj, gdy Z = 1 (równe / zero)
        lda #$FF                ; A = $FF
        sta CH                  ; zapisz A do CH - kod klawisza
        rts                     ; powrót do DOS-u

; --- obsługa błędu: Y = numer błędu ---------------------------------------
blad    tya                     ; A = numer błędu (128..255)
        pha                     ; zachowaj go na czas zamykania pliku i wypisania tekstu
```

```
        ldx #KANAL              ; zamknij plik (nie zaszkodzi, jeśli nie był otwarty)
        lda #12                 ; A = 12
        sta ICCOM,x             ; zapisz A pod adres ICCOM+X - rozkaz
        jsr CIOV                ; wywołaj CIOV: centralna procedura wejścia/wyjścia
        lda #<tblad             ; wypisz słowo "Blad"
        ldy #>tblad             ; Y = starszy bajt adresu tblad
        jsr print               ; wywołaj podprogram print
        pla                     ; odtwórz numer błędu
        jsr liczba              ; wypisz go dziesiętnie
        lda #EOL                ; A = EOL
        jsr putchar             ; wywołaj podprogram putchar
        jmp koniec              ; skocz do koniec

; --- liczba: wypisuje A jako trzy cyfry dziesiętne ------------------------
liczba  ldx #'0'                ; X = znak cyfry setek
l1      cmp #100                ; odejmuj setki, dopóki się da
        bcc l2                  ; skocz do l2, gdy C = 0
        sbc #100                ; A = A - 100 - pożyczka (gdy C = 0)
        inx                     ; X = X + 1
        bne l1                  ; (skok zawsze)
l2      stx setki               ; zapisz X do setki
        ldx #'0'                ; X = znak cyfry dziesiątek
l3      cmp #10                 ; odejmuj dziesiątki, dopóki się da
        bcc l4                  ; skocz do l4, gdy C = 0
        sbc #10                 ; A = A - 10 - pożyczka (gdy C = 0)
        inx                     ; X = X + 1
        bne l3                  ; (skok zawsze)
l4      stx dzies               ; zapisz X do dzies
        ora #'0'                ; A = znak cyfry jedności
        pha                     ; zachowaj jedności
        lda setki               ; wypisz kolejno: setki,
        jsr putchar             ; wywołaj podprogram putchar
        lda dzies               ; dziesiątki,
        jsr putchar             ; wywołaj podprogram putchar
        pla                     ; i jedności
        jmp putchar             ; skocz do putchar

; --- putchar: wypisuje znak z A -------------------------------------------
putchar ldx #$00                ; X = $00
        pha                     ; odłóż A na stos
        lda #11                 ; PUT CHARACTERS z długością 0 = znak z akumulatora
        sta ICCOM,x             ; zapisz A pod adres ICCOM+X - rozkaz
        lda #0                  ; A = 0
        sta ICBLL,x             ; zapisz A pod adres ICBLL+X - długość bufora: młodszy bajt
        sta ICBLH,x             ; zapisz A pod adres ICBLH+X - długość bufora: starszy bajt
        pla                     ; zdejmij A ze stosu
        jmp CIOV                ; skocz do CIOV - centralna procedura wejścia/wyjścia

; --- print: wypisuje wiersz spod adresu A (młodszy) / Y (starszy) ----------
print   ldx #$00                ; X = $00
        sta ICBAL,x             ; zapisz A pod adres ICBAL+X - adres bufora: młodszy bajt
        tya                     ; A = Y
        sta ICBAH,x             ; zapisz A pod adres ICBAH+X - adres bufora: starszy bajt
        lda #9                  ; PUT RECORD
        sta ICCOM,x             ; zapisz A pod adres ICCOM+X - rozkaz
        lda #255                ; A = 255
        sta ICBLL,x             ; zapisz A pod adres ICBLL+X - długość bufora: młodszy bajt
        lda #0                  ; A = 0
        sta ICBLH,x             ; zapisz A pod adres ICBLH+X - długość bufora: starszy bajt
        jmp CIOV                ; skocz do CIOV - centralna procedura wejścia/wyjścia

; --- dane -----------------------------------------------------------------
pytanie dta c'Nazwa pliku (np. D:LIST.TXT):',EOL  ; tekst w kodach ATASCII
tblad   dta c'Blad numer:',EOL  ; tekst w kodach ATASCII
nacisnij dta c'Nacisnij dowolny klawisz',EOL  ; tekst w kodach ATASCII
nazwa   .ds 40                  ; bufor na nazwę pliku
bufor   .ds ROZM                ; bufor na dane z pliku

        org $02E0               ; wektor RUNAD: adres uruchomienia programu
        dta a(start)            ; adres uruchomienia: start
```

Kilka rzeczy wartych zauważenia w tym programie:

- **`sty status`** zaraz po `jsr CIOV` — status trzeba zapamiętać, bo następne wywołanie CIO (wypisanie porcji na ekran) go zamaże.
- **Koniec pliku nie jest błędem.** Ostatnia porcja jest zwykle krótsza niż 128 bajtów: CIO zwraca wtedy status 136 **oraz** poprawną liczbę przeczytanych bajtów w ICBLL. Dlatego najpierw wypisujemy dane, a dopiero potem badamy status.
- **`lda ICBLL+KANAL`** — odczyt pola IOCB kanału 1 bez użycia rejestru X: asembler doda 16 do adresu.
- **Zamknięcie przed otwarciem** chroni przed błędem 129, gdyby poprzedni program zostawił kanał otwarty.

21.4.2 Zapis bloku pamięci do pliku

Zapis różni się tylko wartością ICAX1 przy otwarciu (8) i rozkazem (11). CIO przyjmuje dowolną długość — całą tablicę, ekran czy bufor można zapisać jednym wywołaniem:

Plik: zapisz_blok.asm

```
; zapisz_blok - zapisuje obszar pamięci do pliku
; Wejście:  zb_adr = adres danych (2 bajty), zb_dl = długość (2 bajty)
;           nazwa pliku w "zb_nazwa" (zakończona EOL)
; Wyjście:  Y = status; N = 1 oznacza błąd
ICCOM   = $0342                 ; rozkaz
ICBAL   = $0344                 ; adres bufora: młodszy bajt
ICBAH   = $0345                 ; adres bufora: starszy bajt
ICBLL   = $0348                 ; długość bufora: młodszy bajt
ICBLH   = $0349                 ; długość bufora: starszy bajt
ICAX1   = $034A                 ; parametr 1
ICAX2   = $034B                 ; parametr 2
CIOV    = $E456                 ; centralna procedura wejścia/wyjścia
zb_adr  = $80                   ; adres zapisywanych danych
zb_dl   = $82                   ; długość danych

zapisz_blok ldx #$10            ; kanał 1
        lda #3                  ; OPEN
        sta ICCOM,x             ; zapisz A pod adres ICCOM+X - rozkaz
        lda #<zb_nazwa          ; A = młodszy bajt adresu zb_nazwa
        sta ICBAL,x             ; zapisz A pod adres ICBAL+X - adres bufora: młodszy bajt
        lda #>zb_nazwa          ; A = starszy bajt adresu zb_nazwa
        sta ICBAH,x             ; zapisz A pod adres ICBAH+X - adres bufora: starszy bajt
        lda #8                  ; ICAX1 = 8: otwarcie do zapisu
        sta ICAX1,x             ; zapisz A pod adres ICAX1+X - parametr 1
        lda #0                  ; A = 0
        sta ICAX2,x             ; zapisz A pod adres ICAX2+X - parametr 2
        jsr CIOV                ; wywołaj CIOV: centralna procedura wejścia/wyjścia
        bmi zb_zamk             ; błąd otwarcia: zamknij kanał i wróć z błędem

        ldx #$10                ; X = $10
        lda #11                 ; PUT CHARACTERS: zapisz blok bajtów
        sta ICCOM,x             ; zapisz A pod adres ICCOM+X - rozkaz
        lda zb_adr              ; adres danych
        sta ICBAL,x             ; zapisz A pod adres ICBAL+X - adres bufora: młodszy bajt
        lda zb_adr+1            ; A = bajt spod adresu zb_adr+1
        sta ICBAH,x             ; zapisz A pod adres ICBAH+X - adres bufora: starszy bajt
        lda zb_dl               ; długość danych
        sta ICBLL,x             ; zapisz A pod adres ICBLL+X - długość bufora: młodszy bajt
        lda zb_dl+1             ; A = bajt spod adresu zb_dl+1
        sta ICBLH,x             ; zapisz A pod adres ICBLH+X - długość bufora: starszy bajt
        jsr CIOV                ; zapisz wszystko jednym wywołaniem

zb_zamk tya                     ; zachowaj status zapisu (lub otwarcia)...
        pha                     ; odłóż A na stos
        ldx #$10                ; X = $10
        lda #12                 ; CLOSE - zawsze, także po błędzie
        sta ICCOM,x             ; zapisz A pod adres ICCOM+X - rozkaz
        jsr CIOV                ; wywołaj CIOV: centralna procedura wejścia/wyjścia
        pla                     ; ...i odtwórz go jako wynik procedury
        tay                     ; TAY ustawia też znacznik N zgodnie ze statusem
        rts                     ; powrót z podprogramu

zb_nazwa dta c'D:DANE.BIN',$9B  ; tekst w kodach ATASCII
```

21.5 CIO czy bezpośredni dostęp do sprzętu?

CIO jest wygodne i uniwersalne, ale powolne: wypisanie jednego znaku to kilkaset rozkazów procesora. Dlatego programy dzielą zadania:

- **przez CIO**: pliki dyskowe (zawsze), wczytywanie tekstu od użytkownika, komunikaty, inicjalizacja trybów graficznych,
- **bezpośrednio**: szybkie rysowanie na ekranie (zapis do pamięci obrazu — rozdział 22), odczyt klawiszy i joysticka w grach (rozdział 23), dźwięk (rozdział 25).

Ćwiczenia

1. Napisz program, który pyta o imię i odpowiada „Witaj, *imię*!”.
2. Zmień `pokaz.asm` tak, by po każdych 20 wierszach czekał na klawisz (wskazówka: zliczaj znaki EOL w buforze).
3. Napisz program kopiujący plik: otwórz źródło w kanale 1 (ICAX1 = 4), cel w kanale 2 (ICAX1 = 8) i przepisuj porcje danych, aż do końca pliku.

ROZDZIAŁ 22

Ekran: pamięć obrazu, kolory, display list

Obraz w Atari tworzą wspólnie dwa układy. **ANTIC** co 1/50 sekundy wykonuje własny mały program — *display list* — który mówi mu, jakie rodzaje linii wyświetlić i skąd brać dane. **GTIA** zamienia te dane na kolory. Procesor 6502 w ogóle nie uczestniczy w rysowaniu: jego rolą jest tylko wpisanie danych do pamięci.

Dla programisty oznacza to wielką swobodę. Ekran to zwykły obszar pamięci RAM, a display list to tablica bajtów, którą można napisać samodzielnie.

22.1 Ekran tekstowy od środka

Po włączeniu komputera system ustawia tryb tekstowy (znany z BASIC-a jako GRAPHICS 0): 24 wiersze po 40 znaków. Pamięć obrazu ma 960 bajtów — jeden bajt na znak, wiersz po wierszu. Adres jej początku zawiera zmienna **SAVMSC** (`$58/$59`).

```
SAVMSC ──►  bajt 0     bajt 1    ...   bajt 39      wiersz 0
            bajt 40    bajt 41   ...   bajt 79      wiersz 1
            ...
            bajt 920   bajt 921  ...   bajt 959     wiersz 23

adres znaku w kolumnie x i wierszu y  =  SAVMSC + y * 40 + x
```

Wpisanie bajtu do tej pamięci **natychmiast** zmienia obraz. To o rzędy wielkości szybsze niż wypisywanie przez CIO.

22.1.1 Kody ekranowe

Jest jeden haczyk: w pamięci obrazu znaki mają inne numery niż w ATASCII. To **kody ekranowe** (zwane też wewnętrznymi), odpowiadające kolejności znaków w zestawie znaków:

Kody ATASCII	Znaki	Kody ekranowe	Przeliczenie
`$00–$1F`	znaki graficzne	`$40–$5F`	+ `$40`
`$20–$5F`	spacja, cyfry, wielkie litery	`$00–$3F`	− `$20`
`$60–$7F`	małe litery	`$60–$7F`	bez zmian

W obu kodowaniach **bit 7 oznacza negatyw** (znak w odwróconych kolorach) i przy przeliczaniu pozostaje bez zmian. Spacja ma kod ekranowy 0 — dlatego wyzerowana pamięć obrazu to czysty ekran.

MADS zna oba kodowania. Teksty przeznaczone wprost do pamięci obrazu zapisujemy z literą `d` zamiast `c`, a pojedyncze znaki w cudzysłowie zamiast apostrofu:

```
        lda #"A"                ; A = $21: kod EKRANOWY litery A
        lda #'A'                ; A = $41: kod ATASCII litery A
        rts                     ; powrót z podprogramu
napis1  dta d'Atari'            ; tekst w kodach ekranowych
napis2  dta d'Atari'*           ; gwiazdka za napisem: ten sam tekst w negatywie (bit 7 ustawiony)
```

Gdy tekst pochodzi z zewnątrz (z klawiatury, z pliku), trzeba go przeliczyć w programie:

Plik: asc2int.asm

```
; asc2int - zamiana kodu ATASCII na kod ekranowy
; Wejście:  A = kod ATASCII.  Wyjście: A = kod ekranowy (bit negatywu zachowany).
; Niszczy:  Y, komórkę a2i_tmp.
a2i_tmp = $80                   ; komórka pomocnicza

asc2int tay                     ; zachowaj oryginał w Y (potrzebny będzie bit 7)
```

```
        and #$7F                ; pracuj na kodzie bez bitu negatywu
        cmp #$20                ; kody $00..$1F ?
        bcs a2i_1               ; nie
        ora #$40                ; tak: znaki graficzne -> dodaj $40
        bne a2i_3               ; (skok zawsze - wynik jest niezerowy)
a2i_1   cmp #$60                ; kody $60..$7F ?
        bcs a2i_3               ; tak: małe litery bez zmian
        sec                     ; pozostały kody $20..$5F:
        sbc #$20                ; odejmij $20
a2i_3   sta a2i_tmp             ; zapamiętaj przeliczony kod
        tya                     ; odtwórz oryginał...
        and #$80                ; ...i wydziel z niego bit negatywu
        ora a2i_tmp             ; dołącz go do przeliczonego kodu
        rts                     ; powrót z podprogramu
```

22.1.2 Program: pisanie bezpośrednio po ekranie

Program czyści ekran, rysuje pasek tytułowy w negatywie i wypisuje tekst w wybranym miejscu — wszystko zapisem do pamięci obrazu. Adres wiersza pobiera z tablicy, żeby uniknąć mnożenia przez 40.

Plik: ekran.asm

```
; ------------------------------------------------------------------------
; ekran.asm - bezpośredni zapis do pamięci ekranu tekstowego
; ------------------------------------------------------------------------
SAVMSC  = $58                   ; adres początku pamięci ekranu (2 bajty)
COLOR1  = $02C5                 ; jasność liter
COLOR2  = $02C6                 ; kolor tła
COLOR4  = $02C8                 ; kolor ramki
CRSINH  = $02F0                 ; <> 0: kursor niewidoczny
CH      = $02FC                 ; kod klawisza

wsk     = $80                   ; wskaźnik na miejsce w pamięci ekranu (2 bajty)
zr      = $82                   ; wskaźnik na wypisywany tekst (2 bajty)

        org $2000               ; adres asemblacji: $2000

start   lda #1                  ; ukryj kursor edytora
        sta CRSINH              ; zapisz A do CRSINH - ukrycie kursora
        lda #$00                ; czarne tło i czarna ramka
        sta COLOR2              ; zapisz A do COLOR2 - kolor pola 2 (w trybie tekstowym: tło)
        sta COLOR4              ; zapisz A do COLOR4 - kolor tła (w trybie tekstowym: ramka)
        lda #$0E                ; białe litery (najwyższa jasność)
        sta COLOR1              ; zapisz A do COLOR1

; --- wyczyść cały ekran: 960 bajtów = 3 pełne strony (768) + 192 bajty ------
        lda SAVMSC              ; wsk := początek pamięci ekranu
        sta wsk                 ; zapisz A do wsk
        lda SAVMSC+1            ; A = bajt spod adresu SAVMSC+1
        sta wsk+1               ; zapisz A pod adres wsk+1
        lda #0                  ; A = 0: kod ekranowy spacji
        tay                     ; Y = 0: indeks na stronie
        ldx #3                  ; X = 3 pełne strony
cz1     sta (wsk),y             ; wyzeruj bajt
        iny                     ; Y = Y + 1
        bne cz1                 ; 256 bajtów
        inc wsk+1               ; następna strona
        dex                     ; X = X - 1
        bne cz1                 ; skocz do cz1, gdy Z = 0 (różne / nie zero)
cz2     sta (wsk),y             ; reszta: bajty 768..959 (Y = 0..191)
        iny                     ; Y = Y + 1
        cpy #192                ; porównaj Y z: 192
        bne cz2                 ; skocz do cz2, gdy Z = 0 (różne / nie zero)

; --- pasek tytułowy: cały wiersz 0 w negatywie ------------------------------
        ldy #0                  ; wiersz 0
        jsr ustaw_wiersz        ; wsk := adres początku wiersza
        ldy #39                 ; 40 znaków, od ostatniego
pas     lda #$80                ; kod ekranowy spacji w negatywie
        sta (wsk),y             ; zapisz A pod adres (wsk) + Y
        dey                     ; Y = Y - 1
```

```
        bpl pas                 ; skocz do pas, gdy N = 0 (bit 7 skasowany)
        ldy #[40-DL_TYT]/2      ; kolumna początkowa: tytuł wyśrodkowany
        ldx #0                  ; X = indeks znaku tytułu
tyt     lda tytul,x             ; znak tytułu (kod ekranowy)
        ora #$80                ; ustaw bit 7: negatyw
        sta (wsk),y             ; postaw na ekranie
        iny                     ; Y = Y + 1
        inx                     ; X = X + 1
        cpx #DL_TYT             ; cały tytuł?
        bne tyt                 ; skocz do tyt, gdy Z = 0 (różne / nie zero)

; --- tekst w wierszu 10, od kolumny 8 ---------------------------------------
        ldy #10                 ; wiersz 10
        jsr ustaw_wiersz        ; wywołaj podprogram ustaw_wiersz
        ldy #8                  ; kolumna 8
        ldx #0                  ; X = 0
tk1     lda tekst,x             ; znak tekstu
        sta (wsk),y             ; zapisz A pod adres (wsk) + Y
        iny                     ; Y = Y + 1
        inx                     ; X = X + 1
        cpx #DL_TEK             ; porównaj X z: DL_TEK
        bne tk1                 ; skocz do tk1, gdy Z = 0 (różne / nie zero)

; --- czekaj na klawisz i zakończ --------------------------------------------
        lda #$FF                ; A = $FF
        sta CH                  ; zapisz A do CH - kod klawisza
czekaj  lda CH                  ; A = CH - kod klawisza
        cmp #$FF                ; porównaj A z: $FF
        beq czekaj              ; skocz do czekaj, gdy Z = 1 (równe / zero)
        lda #$FF                ; A = $FF
        sta CH                  ; zapisz A do CH - kod klawisza
        lda #0                  ; przywróć widoczność kursora
        sta CRSINH              ; zapisz A do CRSINH - ukrycie kursora
        rts                     ; powrót z podprogramu

; --- ustaw_wiersz: wsk := SAVMSC + Y * 40  (Y = numer wiersza 0..23) -------
ustaw_wiersz lda SAVMSC         ; młodszy bajt adresu ekranu
        clc                     ; C = 0 (przed dodawaniem)
        adc wiersz_lo,y         ; plus młodszy bajt przesunięcia wiersza
        sta wsk                 ; zapisz A do wsk
        lda SAVMSC+1            ; starszy bajt adresu ekranu
        adc wiersz_hi,y         ; plus starszy bajt przesunięcia i przeniesienie
        sta wsk+1               ; zapisz A pod adres wsk+1
        rts                     ; powrót z podprogramu

; --- tablica przesunięć początków wierszy: numer wiersza * 40 ---------------
wiersz_lo dta l(0,40,80,120,160,200,240,280,320,360,400,440)  ; młodsze bajty
        dta l(480,520,560,600,640,680,720,760,800,840,880,920)  ; młodsze bajty
wiersz_hi dta h(0,40,80,120,160,200,240,280,320,360,400,440)  ; starsze bajty
        dta h(480,520,560,600,640,680,720,760,800,840,880,920)  ; starsze bajty

tytul   dta d'KURS ASEMBLERA 6502'  ; litera d: kody ekranowe
DL_TYT  = *-tytul               ; długość tytułu
tekst   dta d'Tekst wpisany wprost do pamieci'  ; tekst w kodach ekranowych
DL_TEK  = *-tekst               ; długość tekstu

        org $02E0               ; wektor RUNAD: adres uruchomienia programu
        dta a(start)            ; adres uruchomienia: start
```

22.2 Kolory

Atari ma paletę 256 barw: 16 **odcieni** po 16 **jasności** (w większości trybów rozróżnianych jest 8 jasności — bit 0 jest pomijany). Bajt koloru składa się z dwóch półbajtów:

```
    bajt koloru  =  odcień * 16 + jasność          np. $94 = odcień 9 (niebieski), jasność 4
```

Odcień	Barwa (PAL)	Odcień	Barwa (PAL)
0	szarości (czerń–biel)	8	niebieski
1	złoty / brązowy	9	jasnoniebieski
2	pomarańczowy	10	turkusowy
3	czerwonopomarańczowy	11	zielononiebieski
4	różowy / czerwony	12	zielony
5	purpurowy	13	żółtozielony
6	fioletowy	14	oliwkowy
7	fioletowoniebieski	15	jasnopomarańczowy

Dokładne barwy zależą od systemu telewizyjnego i regulacji komputera.

Kolorów w danej chwili jest na ekranie niewiele — tyle, ile rejestrów kolorów: pięć dla pola gry i cztery dla duszków. Każdy tryb graficzny inaczej je wykorzystuje. W trybie tekstowym:

Cień	Adres	Rola w trybie tekstowym (GRAPHICS 0)
COLOR1	`$02C5` (709)	**jasność** liter (odcień jest brany z koloru tła)
COLOR2	`$02C6` (710)	kolor tła
COLOR4	`$02C8` (712)	kolor ramki

```
; białe litery na czarnym tle, czarna ramka
        lda #$00                ; A = $00
        sta $02C6               ; COLOR2: tło czarne
        sta $02C8               ; COLOR4: ramka czarna
        lda #$0E                ; A = $0E
        sta $02C5               ; COLOR1: litery o najwyższej jasności
```

22.3 Tryby graficzne

System operacyjny potrafi sam przygotować display list i pamięć obrazu dla kilkunastu trybów. Robi się to, otwierając przez CIO urządzenie `S:` — dokładnie to wykonuje instrukcja GRAPHICS w BASIC-u. Numer trybu podajemy w ICAX2, a w ICAX1 wartość 12 (odczyt i zapis), powiększoną o 16, jeśli chcemy okno tekstowe u dołu ekranu.

Tryb BASIC	Tryb ANTIC	Rodzaj	Rozdzielczość	Kolory	Bajtów w linii	Pamięć obrazu
0	2	tekst	40 × 24	1 + jasność	40	960
1	6	tekst	20 × 24	5	20	480
2	7	tekst	20 × 12	5	20	240
3	8	grafika	40 × 24	4	10	240
5	A	grafika	80 × 48	4	20	960
7	D	grafika	160 × 96	4	40	3840
8	F	grafika	320 × 192	1 + jasność	40	7680
15	E	grafika	160 × 192	4	40	7680
9, 10, 11	F	grafika GTIA	80 × 192	16 jasności / 9 kolorów / 16 odcieni	40	7680

Po otwarciu trybu zmienna SAVMSC wskazuje nową pamięć obrazu. Punkty stawiamy, zapisując bity wprost do tej pamięci. W trybie 8 każdy bit to jeden punkt; najstarszy bit bajtu jest punktem najbardziej z lewej:

```
adres bajtu  =  SAVMSC + y * 40 + x / 8             (x = 0..319, y = 0..191)
bit w bajcie =  maska $80 przesunięta w prawo o (x mod 8) pozycji
```

To znakomite ćwiczenie łączące arytmetykę 16-bitową, przesunięcia i tablicę masek:

Plik: plot8.asm

```
; plot8 - stawia punkt w trybie graficznym 8 (320 x 192)
; Wejście:  px = współrzędna x (2 bajty, 0..319), py = współrzędna y (1 bajt, 0..191)
; Niszczy:  A, X, Y oraz zmienne pl_wsk, pl_tmp.
SAVMSC  = $58                   ; adres pamięci obrazu
px      = $80                   ; współrzędna x (2 bajty)
py      = $82                   ; współrzędna y
pl_wsk  = $83                   ; wskaźnik na bajt obrazu (2 bajty)
pl_tmp  = $85                   ; zmienna pomocnicza (2 bajty)

plot8   lda py                  ; --- pl_wsk := y * 40 (wynik 16-bitowy) ---
        sta pl_wsk              ; zapisz A do pl_wsk
        lda #0                  ; A = 0
        sta pl_wsk+1            ; zapisz A pod adres pl_wsk+1
        asl pl_wsk              ; y * 2
        rol pl_wsk+1            ; obróć bajt spod adresu pl_wsk+1 w lewo przez C
        asl pl_wsk              ; y * 4
        rol pl_wsk+1            ; obróć bajt spod adresu pl_wsk+1 w lewo przez C
        asl pl_wsk              ; y * 8
        rol pl_wsk+1            ; obróć bajt spod adresu pl_wsk+1 w lewo przez C
        lda pl_wsk              ; pl_tmp := y * 8
        sta pl_tmp              ; zapisz A do pl_tmp
        lda pl_wsk+1            ; A = bajt spod adresu pl_wsk+1
        sta pl_tmp+1            ; zapisz A pod adres pl_tmp+1
        asl pl_wsk              ; y * 16
        rol pl_wsk+1            ; obróć bajt spod adresu pl_wsk+1 w lewo przez C
        asl pl_wsk              ; y * 32
        rol pl_wsk+1            ; obróć bajt spod adresu pl_wsk+1 w lewo przez C
        clc                     ; pl_wsk := y * 32 + y * 8 = y * 40
        lda pl_wsk              ; A = pl_wsk
        adc pl_tmp              ; A = A + pl_tmp + C
        sta pl_wsk              ; zapisz A do pl_wsk
        lda pl_wsk+1            ; A = bajt spod adresu pl_wsk+1
        adc pl_tmp+1            ; A = A + bajt spod adresu pl_tmp+1 + C
        sta pl_wsk+1            ; zapisz A pod adres pl_wsk+1
        clc                     ; --- pl_wsk := pl_wsk + SAVMSC (adres początku linii) ---
        lda pl_wsk              ; A = pl_wsk
        adc SAVMSC              ; A = A + SAVMSC + C
        sta pl_wsk              ; zapisz A do pl_wsk
        lda pl_wsk+1            ; A = bajt spod adresu pl_wsk+1
        adc SAVMSC+1            ; A = A + bajt spod adresu SAVMSC+1 + C
        sta pl_wsk+1            ; zapisz A pod adres pl_wsk+1

        lda px+1                ; --- Y := x / 8 (numer bajtu w linii, 0..39) ---
        lsr @                   ; najstarszy (dziewiąty) bit współrzędnej x -> C
        lda px                  ; młodszy bajt x
        ror @                   ; A = x / 2 (dziewiąty bit wszedł z lewej)
        lsr @                   ; A = x / 4
        lsr @                   ; A = x / 8
        tay                     ; Y = przesunięcie bajtu w linii

        lda px                  ; --- X := x mod 8 (numer bitu w bajcie, od lewej) ---
        and #7                  ; A = A AND 7
        tax                     ; X = A
        lda (pl_wsk),y          ; pobierz bajt obrazu
        ora pl_maski,x          ; zapal w nim właściwy bit
        sta (pl_wsk),y          ; zapisz z powrotem
        rts                     ; powrót z podprogramu

pl_maski dta $80,$40,$20,$10,$08,$04,$02,$01  ; maski punktów od lewej do prawej
```

Aby punkt **zgasić**, zamiast `ora pl_maski,x` użyj maski odwróconej i rozkazu `and`; aby go **odwrócić** — rozkazu `eor`.

22.3.1 Program: tryb graficzny 8

Plik: gr8.asm

```
; ------------------------------------------------------------------------
; gr8.asm - otwiera tryb graficzny 8 i rysuje dwie przekątne oraz ramkę
; ------------------------------------------------------------------------
ICCOM   = $0342                 ; rozkaz
ICBAL   = $0344                 ; adres bufora: młodszy bajt
ICBAH   = $0345                 ; adres bufora: starszy bajt
ICAX1   = $034A                 ; parametr 1
ICAX2   = $034B                 ; parametr 2
CIOV    = $E456                 ; centralna procedura wejścia/wyjścia
SAVMSC  = $58                   ; adres pamięci ekranu (2 bajty)
COLOR1  = $02C5                 ; kolor pola 1 (w trybie tekstowym: jasność liter)
COLOR2  = $02C6                 ; kolor pola 2 (w trybie tekstowym: tło)
COLOR4  = $02C8                 ; kolor tła (w trybie tekstowym: ramka)
CH      = $02FC                 ; kod klawisza

px      = $80                   ; współrzędna x punktu (2 bajty)
py      = $82                   ; współrzędna y punktu
pl_wsk  = $83                   ; zmienne procedury plot8
pl_tmp  = $85                   ; zmienna pomocnicza procedury plot8 (2 bajty)
licz    = $87                   ; licznik pętli (2 bajty)

        org $2000               ; adres asemblacji: $2000

start   jsr grafika8            ; włącz tryb graficzny 8
        bpl st1                 ; N = 0: udało się
        rts                     ; błąd (np. brak pamięci): wróć do DOS-u
st1     lda #$00                ; czarne tło i ramka
        sta COLOR2              ; zapisz A do COLOR2 - kolor pola 2 (w trybie tekstowym: tło)
        sta COLOR4              ; zapisz A do COLOR4 - kolor tła (w trybie tekstowym: ramka)
        lda #$0E                ; jasne punkty
        sta COLOR1              ; zapisz A do COLOR1

; --- przekątna z lewego górnego rogu: punkty (i, i) dla i = 0..191 ---------
        lda #0                  ; A = 0
        sta licz                ; zapisz A do licz
p1      lda licz                ; x := i
        sta px                  ; zapisz A do px
        lda #0                  ; A = 0
        sta px+1                ; zapisz A pod adres px+1
        lda licz                ; y := i
        sta py                  ; zapisz A do py
        jsr plot8               ; wywołaj podprogram plot8
        inc licz                ; licz = licz + 1
        lda licz                ; A = licz
        cmp #192                ; i = 192 ?
        bne p1                  ; skocz do p1, gdy Z = 0 (różne / nie zero)

; --- druga przekątna: punkty (319 - i, i) ----------------------------------
        lda #0                  ; A = 0
        sta licz                ; zapisz A do licz
p2      sec                     ; x := 319 - i (odejmowanie 16-bitowe)
        lda #<319               ; A = młodszy bajt wartości 319
        sbc licz                ; A = A - licz - pożyczka (gdy C = 0)
        sta px                  ; zapisz A do px
        lda #>319               ; A = starszy bajt wartości 319
        sbc #0                  ; A = A - 0 - pożyczka (gdy C = 0)
        sta px+1                ; zapisz A pod adres px+1
        lda licz                ; y := i
        sta py                  ; zapisz A do py
        jsr plot8               ; wywołaj podprogram plot8
        inc licz                ; licz = licz + 1
        lda licz                ; A = licz
        cmp #192                ; porównaj A z: 192
        bne p2                  ; skocz do p2, gdy Z = 0 (różne / nie zero)

; --- linie poziome na górze (y = 0) i na dole (y = 191): x = 0..319 --------
        lda #0                  ; licznik 16-bitowy := 0
        sta licz                ; zapisz A do licz
        sta licz+1              ; zapisz A pod adres licz+1
```

```
p3      lda licz                ; x := licznik
        sta px                  ; zapisz A do px
        lda licz+1              ; A = bajt spod adresu licz+1
        sta px+1                ; zapisz A pod adres px+1
        lda #0                  ; punkt na górnej krawędzi
        sta py                  ; zapisz A do py
        jsr plot8               ; wywołaj podprogram plot8
        lda #191                ; punkt na dolnej krawędzi (px się nie zmieniło)
        sta py                  ; zapisz A do py
        jsr plot8               ; wywołaj podprogram plot8
        inc licz                ; licznik := licznik + 1
        bne p4                  ; skocz do p4, gdy Z = 0 (różne / nie zero)
        inc licz+1              ; zwiększ o 1 bajt spod adresu licz+1
p4      lda licz                ; czy licznik = 320 ?
        cmp #<320               ; porównaj A z: młodszy bajt wartości 320
        bne p3                  ; skocz do p3, gdy Z = 0 (różne / nie zero)
        lda licz+1              ; A = bajt spod adresu licz+1
        cmp #>320               ; porównaj A z: starszy bajt wartości 320
        bne p3                  ; skocz do p3, gdy Z = 0 (różne / nie zero)

; --- czekaj na klawisz, wróć do trybu tekstowego ----------------------------
        lda #$FF                ; A = $FF
        sta CH                  ; zapisz A do CH - kod klawisza
czekaj  lda CH                  ; A = CH - kod klawisza
        cmp #$FF                ; porównaj A z: $FF
        beq czekaj              ; skocz do czekaj, gdy Z = 1 (równe / zero)
        lda #$FF                ; A = $FF
        sta CH                  ; zapisz A do CH - kod klawisza
        jsr tekstowy            ; przywróć tryb tekstowy
        rts                     ; powrót z podprogramu

; --- grafika8: odpowiednik instrukcji GRAPHICS 8+16 (bez okna tekstowego) --
; Wyjście: Y = status CIO, N = 1 przy błędzie
grafika8 ldx #$60               ; kanał 6 - zwyczajowo dla ekranu graficznego
        lda #12                 ; CLOSE: zamknij, jeśli był otwarty
        sta ICCOM,x             ; zapisz A pod adres ICCOM+X - rozkaz
        jsr CIOV                ; wywołaj CIOV: centralna procedura wejścia/wyjścia
        ldx #$60                ; X = $60
        lda #3                  ; OPEN
        sta ICCOM,x             ; zapisz A pod adres ICCOM+X - rozkaz
        lda #<nazwa_s           ; nazwa urządzenia: "S:"
        sta ICBAL,x             ; zapisz A pod adres ICBAL+X - adres bufora: młodszy bajt
        lda #>nazwa_s           ; A = starszy bajt adresu nazwa_s
        sta ICBAH,x             ; zapisz A pod adres ICBAH+X - adres bufora: starszy bajt
        lda #12                 ; ICAX1 = 12: odczyt i zapis, bez okna tekstowego
        sta ICAX1,x             ; zapisz A pod adres ICAX1+X - parametr 1
        lda #8                  ; ICAX2 = 8: numer trybu graficznego
        sta ICAX2,x             ; zapisz A pod adres ICAX2+X - parametr 2
        jmp CIOV                ; otwórz; system przygotuje display list i wyczyści obraz

; --- tekstowy: powrót do trybu tekstowego (ponowne otwarcie edytora E:) -----
tekstowy ldx #$60               ; zamknij kanał 6
        lda #12                 ; A = 12
        sta ICCOM,x             ; zapisz A pod adres ICCOM+X - rozkaz
        jsr CIOV                ; wywołaj CIOV: centralna procedura wejścia/wyjścia
        ldx #$00                ; zamknij kanał 0 (edytor)
        lda #12                 ; A = 12
        sta ICCOM,x             ; zapisz A pod adres ICCOM+X - rozkaz
        jsr CIOV                ; wywołaj CIOV: centralna procedura wejścia/wyjścia
        ldx #$00                ; i otwórz go ponownie - to odtwarza ekran tekstowy
        lda #3                  ; A = 3
        sta ICCOM,x             ; zapisz A pod adres ICCOM+X - rozkaz
        lda #<nazwa_e           ; A = młodszy bajt adresu nazwa_e
        sta ICBAL,x             ; zapisz A pod adres ICBAL+X - adres bufora: młodszy bajt
        lda #>nazwa_e           ; A = starszy bajt adresu nazwa_e
        sta ICBAH,x             ; zapisz A pod adres ICBAH+X - adres bufora: starszy bajt
        lda #12                 ; A = 12
        sta ICAX1,x             ; zapisz A pod adres ICAX1+X - parametr 1
        lda #0                  ; A = 0
        sta ICAX2,x             ; zapisz A pod adres ICAX2+X - parametr 2
        jmp CIOV                ; skocz do CIOV - centralna procedura wejścia/wyjścia

; --- plot8: stawia punkt (px, py) - opis w tekście rozdziału ----------------
```

```
plot8   lda py                  ; A = py
        sta pl_wsk              ; zapisz A do pl_wsk
        lda #0                  ; A = 0
        sta pl_wsk+1            ; zapisz A pod adres pl_wsk+1
        asl pl_wsk              ; y * 2
        rol pl_wsk+1            ; obróć bajt spod adresu pl_wsk+1 w lewo przez C
        asl pl_wsk              ; y * 4
        rol pl_wsk+1            ; obróć bajt spod adresu pl_wsk+1 w lewo przez C
        asl pl_wsk              ; y * 8
        rol pl_wsk+1            ; obróć bajt spod adresu pl_wsk+1 w lewo przez C
        lda pl_wsk              ; A = pl_wsk
        sta pl_tmp              ; zapisz A do pl_tmp
        lda pl_wsk+1            ; A = bajt spod adresu pl_wsk+1
        sta pl_tmp+1            ; zapisz A pod adres pl_tmp+1
        asl pl_wsk              ; y * 16
        rol pl_wsk+1            ; obróć bajt spod adresu pl_wsk+1 w lewo przez C
        asl pl_wsk              ; y * 32
        rol pl_wsk+1            ; obróć bajt spod adresu pl_wsk+1 w lewo przez C
        clc                     ; y * 32 + y * 8 = y * 40
        lda pl_wsk              ; A = pl_wsk
        adc pl_tmp              ; A = A + pl_tmp + C
        sta pl_wsk              ; zapisz A do pl_wsk
        lda pl_wsk+1            ; A = bajt spod adresu pl_wsk+1
        adc pl_tmp+1            ; A = A + bajt spod adresu pl_tmp+1 + C
        sta pl_wsk+1            ; zapisz A pod adres pl_wsk+1
        clc                     ; + adres pamięci obrazu
        lda pl_wsk              ; A = pl_wsk
        adc SAVMSC              ; A = A + SAVMSC + C
        sta pl_wsk              ; zapisz A do pl_wsk
        lda pl_wsk+1            ; A = bajt spod adresu pl_wsk+1
        adc SAVMSC+1            ; A = A + bajt spod adresu SAVMSC+1 + C
        sta pl_wsk+1            ; zapisz A pod adres pl_wsk+1
        lda px+1                ; Y := x / 8
        lsr @                   ; przesuń A o 1 bit w prawo (bit 0 -> C)
        lda px                  ; A = px
        ror @                   ; obróć A w prawo przez C
        lsr @                   ; przesuń A o 1 bit w prawo (bit 0 -> C)
        lsr @                   ; przesuń A o 1 bit w prawo (bit 0 -> C)
        tay                     ; Y = A
        lda px                  ; X := x mod 8
        and #7                  ; A = A AND 7
        tax                     ; X = A
        lda (pl_wsk),y          ; zapal punkt
        ora pl_maski,x          ; A = A OR bajt spod adresu pl_maski+X
        sta (pl_wsk),y          ; zapisz A pod adres (pl_wsk) + Y
        rts                     ; powrót z podprogramu

pl_maski dta $80,$40,$20,$10,$08,$04,$02,$01  ; dane: bajty
nazwa_s dta c'S:',$9B           ; tekst w kodach ATASCII
nazwa_e dta c'E:',$9B           ; tekst w kodach ATASCII

        org $02E0               ; wektor RUNAD: adres uruchomienia programu
        dta a(start)            ; adres uruchomienia: start
```

Stawianie pojedynczych punktów jest proste, ale wolne — każde wywołanie liczy adres od nowa. Szybkie procedury rysowania linii przesuwają wskaźnik i maskę przyrostowo (o jeden punkt), zamiast przeliczać je za każdym razem.

22.4 Display list

Display list to program dla układu ANTIC — ciąg bajtów-rozkazów, z których każdy opisuje jedną linię trybu albo steruje samym układem ANTIC. Adres aktualnej display list znajduje się w cieniu SDLSTL (`$0230`/`$0231`).

22.4.1 Rozkazy display list

Bajt	Znaczenie
`$00`, `$10`, ... `$70`	1, 2, ... 8 pustych linii obrazu (w kolorze tła)
`$02`–`$0F`	jedna linia w trybie ANTIC o tym numerze
`$41` + 2 bajty adresu	skok na początek display list i oczekiwanie na następną ramkę (JVB)
`$01` + 2 bajty adresu	skok do dalszej części display list (JMP)

Do rozkazu linii trybu można dodać bity modyfikujące:

Bit	Wartość	Znaczenie
6	`$40`	**LMS**: za rozkazem następują 2 bajty adresu pamięci obrazu dla tej linii
7	`$80`	**DLI**: po tej linii wywołaj przerwanie (rozdział 24)
5	`$20`	płynny przesuw pionowy
4	`$10`	płynny przesuw poziomy

Tryby ANTIC:

Tryb	Rodzaj	Znaków / punktów w linii	Linii obrazu	Bajtów	Kolory
2	tekst	40 znaków	8	40	1 + jasność
3	tekst (wysokie znaki)	40 znaków	10	40	1 + jasność
4	tekst wielokolorowy	40 znaków	8	40	5
5	tekst wielokolorowy	40 znaków	16	40	5
6	tekst duży	20 znaków	8	20	5
7	tekst duży, wysoki	20 znaków	16	20	5
8	grafika	40 punktów	8	10	4
9	grafika	80 punktów	4	10	2
A	grafika	80 punktów	4	20	4
B	grafika	160 punktów	2	20	2
C	grafika	160 punktów	1	20	2
D	grafika	160 punktów	2	40	4
E	grafika	160 punktów	1	40	4
F	grafika	320 punktów	1	40	1 + jasność

Standardowa display list trybu tekstowego wygląda tak:

```
$70 $70 $70         3 x 8 = 24 puste linie (górny margines, niewidoczny na telewizorze)
$42 lo hi           linia trybu 2 z LMS: stąd ANTIC zna adres pamięci obrazu
$02 $02 ... $02     23 kolejne linie trybu 2 (dane pobierane kolejno z pamięci)
$41 lo hi           czekaj na koniec ramki i skocz na początek tej display list
```

Zasady, których trzeba przestrzegać:

- display list zaczyna się zwykle od 24 pustych linii, a cały obraz nie powinien przekraczać 240 linii,
- pierwszy rozkaz trybu musi mieć LMS (inaczej ANTIC nie wie, skąd brać dane),
- **display list nie może przekraczać granicy 1 KB** (adresu podzielnego przez `$400`) — w razie potrzeby użyj rozkazu skoku `$01`,
- **pamięć obrazu nie może przekraczać granicy 4 KB** (adresu podzielnego przez `$1000`) w obrębie jednego ciągu linii — linia leżąca za granicą musi dostać własny LMS.

22.4.2 Program: własna display list

Budujemy ekran złożony z różnych trybów: wielki tytuł, podtytuł średniej wielkości i trzy wiersze zwykłego tekstu. W trybach 6 i 7 dwa najstarsze bity kodu znaku wybierają **kolor** (z rejestrów COLOR0–COLOR3), więc ten sam tekst można pokolorować samym sposobem zapisu: wielkie litery, małe litery i znaki w negatywie dają różne kolory.

Plik: dlist.asm

```
; ------------------------------------------------------------------------
; dlist.asm - własna display list łącząca trzy tryby tekstowe
; ------------------------------------------------------------------------
SDLSTL  = $0230                 ; cień adresu display list (2 bajty)
COLOR0  = $02C4                 ; kolory pola gry
COLOR1  = $02C5                 ; kolor pola 1 (w trybie tekstowym: jasność liter)
COLOR2  = $02C6                 ; kolor pola 2 (w trybie tekstowym: tło)
COLOR3  = $02C7                 ; kolor pola 3
COLOR4  = $02C8                 ; kolor tła
RTCLOK  = $14                   ; zegar systemowy
CH      = $02FC                 ; kod klawisza

stara_dl = $80                  ; zapamiętany adres systemowej display list (2 bajty)

        org $2000               ; adres asemblacji: $2000

start   lda SDLSTL              ; zapamiętaj adres systemowej display list,
        sta stara_dl            ; żeby przywrócić ją przy wyjściu
        lda SDLSTL+1            ; A = bajt spod adresu SDLSTL+1
        sta stara_dl+1          ; zapisz A pod adres stara_dl+1

        lda #<dlist             ; wpisz adres własnej display list do cienia;
        sta SDLSTL              ; system przekaże go układowi ANTIC
        lda #>dlist             ; w najbliższym przerwaniu VBI
        sta SDLSTL+1            ; zapisz A pod adres SDLSTL+1

        lda #$1A                ; COLOR0: złoty  - wielkie litery w trybach 6 i 7
        sta COLOR0              ; zapisz A do COLOR0 - kolor pola 0
        lda #$0E                ; COLOR1: biały  - małe litery w trybach 6 i 7,
        sta COLOR1              ; a w trybie 2: jasność liter
        lda #$94                ; COLOR2: niebieski - tło wierszy trybu 2
        sta COLOR2              ; zapisz A do COLOR2 - kolor pola 2 (w trybie tekstowym: tło)
        lda #$46                ; COLOR3: czerwony
        sta COLOR3              ; zapisz A do COLOR3 - kolor pola 3
        lda #$00                ; COLOR4: czarne tło całego ekranu
        sta COLOR4              ; zapisz A do COLOR4 - kolor tła (w trybie tekstowym: ramka)

        lda #$FF                ; czekaj na klawisz
        sta CH                  ; zapisz A do CH - kod klawisza
czekaj  lda CH                  ; A = CH - kod klawisza
        cmp #$FF                ; porównaj A z: $FF
        beq czekaj              ; skocz do czekaj, gdy Z = 1 (równe / zero)
        lda #$FF                ; A = $FF
        sta CH                  ; zapisz A do CH - kod klawisza

        lda stara_dl            ; przywróć systemową display list
        sta SDLSTL              ; zapisz A do SDLSTL - cień adresu display list (2 bajty)
        lda stara_dl+1          ; A = bajt spod adresu stara_dl+1
        sta SDLSTL+1            ; zapisz A pod adres SDLSTL+1
        lda RTCLOK              ; poczekaj jedną ramkę, aż system przekaże
cz2     cmp RTCLOK              ; stary adres układowi ANTIC - dopiero wtedy
        beq cz2                 ; nasza display list przestanie być używana
        rts                     ; powrót z podprogramu

; --- display list -------------------------------------------------------
dlist   dta $70,$70,$70         ; 24 puste linie: górny margines
        dta $70,$70             ; jeszcze 16 pustych linii odstępu
        dta $47,a(ekran)        ; tryb 7 (wielkie znaki) + LMS: dane od adresu "ekran"
        dta $70                 ; 8 pustych linii
        dta $06                 ; tryb 6 (duże znaki): następne 20 bajtów danych
        dta $70,$70             ; 16 pustych linii
        dta $02                 ; tryb 2 (zwykły tekst): następne 40 bajtów
        dta $02                 ; tryb 2: kolejne 40 bajtów
```

```
        dta $02                 ; tryb 2: kolejne 40 bajtów
        dta $41,a(dlist)        ; koniec: czekaj na następną ramkę, zacznij od "dlist"

; --- pamięć obrazu: dane w kolejności wyświetlania ---------------------------
ekran   dta d'   ATARI  XL/XE     '  ; tryb 7: dokładnie 20 znaków (kolor COLOR0)
        dta d'  asembler  6502    '  ; tryb 6: 20 znaków; małe litery = kolor COLOR1
        dta d'  To jest zwykly tekst w trybie 2.      '  ; tryb 2: 40 znaków
        dta d'  Kazda linia moze miec inny tryb.      '  ; tryb 2: 40 znaków
        dta d'  Nacisnij dowolny klawisz...           '  ; tryb 2: 40 znaków

        org $02E0               ; wektor RUNAD: adres uruchomienia programu
        dta a(start)            ; adres uruchomienia: start
```

Zwróć uwagę, że LMS jest tu tylko jeden: ANTIC po wyświetleniu linii sam przesuwa swój wewnętrzny wskaźnik pamięci o tyle bajtów, ile linia zużyła. Dane kolejnych linii muszą więc leżeć w pamięci jedna za drugą — i mieć dokładnie właściwą długość (20 lub 40 bajtów).

22.5 Własny zestaw znaków

Kształty znaków to także zwykłe dane. Zestaw znaków zajmuje 1024 bajty: 128 znaków po 8 bajtów, z których każdy opisuje jedną linię znaku (bit ustawiony = punkt zapalony). Znak o kodzie ekranowym k zajmuje bajty od $k \cdot 8$ do $k \cdot 8 + 7$.

```
   litera A (kod ekranowy $21)       bajty
   . . . . . . . .                   %00000000  $00
   . . . # # . . .                   %00011000  $18
   . . # # # # . .                   %00111100  $3C
   . # # . . # # .                   %01100110  $66
   . # # . . # # .                   %01100110  $66
   . # # # # # # .                   %01111110  $7E
   . # # . . # # .                   %01100110  $66
   . . . . . . . .                   %00000000  $00
```

Standardowy zestaw leży w pamięci ROM pod adresem `$E000`. ANTIC szuka zestawu pod adresem, którego **starszy bajt** podaje rejestr CHBASE (cień: CHBAS, `$02F4`); młodszy bajt jest zawsze zerem. Aby zmienić znaki:

1. skopiuj zestaw z ROM-u do pamięci RAM — pod adres podzielny przez 1024 (`$400`),
2. zmień kształty wybranych znaków,
3. wpisz starszy bajt adresu do CHBAS.

Plik: znaki.asm

```
; ---------------------------------------------------------------------------
; znaki.asm - własny zestaw znaków: zamienia znak "serce" na ludzika
; ---------------------------------------------------------------------------
CHBAS   = $02F4                 ; cień CHBASE: starszy bajt adresu zestawu znaków
SAVMSC  = $58                   ; adres pamięci ekranu
CH      = $02FC                 ; kod klawisza

ZNAKI   = $3000                 ; adres nowego zestawu: MUSI być podzielny przez $400
ROMZN   = $E000                 ; adres zestawu w pamięci ROM

zr      = $80                   ; wskaźnik źródła
cel     = $82                   ; wskaźnik celu

        org $2000               ; adres asemblacji: $2000

start   lda #<ROMZN             ; zr := adres zestawu w ROM-ie
        sta zr                  ; zapisz A do zr
        lda #>ROMZN             ; A = starszy bajt wartości ROMZN
        sta zr+1                ; zapisz A pod adres zr+1
        lda #<ZNAKI             ; cel := adres nowego zestawu
        sta cel                 ; zapisz A do cel
        lda #>ZNAKI             ; A = starszy bajt wartości ZNAKI
        sta cel+1               ; zapisz A pod adres cel+1
        ldx #4                  ; X = 4 strony = 1024 bajty
        ldy #0                  ; Y = 0
kop     lda (zr),y              ; skopiuj bajt z ROM-u do RAM-u
```

```
        sta (cel),y             ; zapisz A pod adres (cel) + Y
        iny                     ; Y = Y + 1
        bne kop                 ; cała strona
        inc zr+1                ; następna strona
        inc cel+1               ; zwiększ o 1 bajt spod adresu cel+1
        dex                     ; X = X - 1
        bne kop                 ; skocz do kop, gdy Z = 0 (różne / nie zero)

        ldx #7                  ; --- podmień kształt znaku o kodzie ekranowym $40 ("serce") ---
zm      lda ludzik,x            ; bajt nowego kształtu
        sta ZNAKI+$40*8,x       ; znak $40 zaczyna się $40*8 bajtów od początku zestawu
        dex                     ; X = X - 1
        bpl zm                  ; skocz do zm, gdy N = 0 (bit 7 skasowany)

        lda #>ZNAKI             ; włącz nowy zestaw: starszy bajt adresu do cienia CHBAS
        sta CHBAS               ; zapisz A do CHBAS - cień CHBASE

        ldy #0                  ; postaw 10 ludzików na początku ekranu
        lda #$40                ; kod ekranowy podmienionego znaku
lud     sta (SAVMSC),y          ; SAVMSC leży na stronie zerowej - można go użyć jako wskaźnika
        iny                     ; Y = Y + 1
        cpy #10                 ; porównaj Y z: 10
        bne lud                 ; skocz do lud, gdy Z = 0 (różne / nie zero)

        lda #$FF                ; czekaj na klawisz
        sta CH                  ; zapisz A do CH - kod klawisza
czekaj  lda CH                  ; A = CH - kod klawisza
        cmp #$FF                ; porównaj A z: $FF
        beq czekaj              ; skocz do czekaj, gdy Z = 1 (równe / zero)
        lda #$FF                ; A = $FF
        sta CH                  ; zapisz A do CH - kod klawisza
        lda #>ROMZN             ; przywróć standardowy zestaw znaków
        sta CHBAS               ; zapisz A do CHBAS - cień CHBASE
        rts                     ; powrót z podprogramu

ludzik  dta %00011000           ; kształt ludzika: 8 linii po 8 punktów
        dta %00011000           ; dane: wzór bitów
        dta %01111110           ; dane: wzór bitów
        dta %00011000           ; dane: wzór bitów
        dta %00011000           ; dane: wzór bitów
        dta %00100100           ; dane: wzór bitów
        dta %01000010           ; dane: wzór bitów
        dta %00000000           ; dane: wzór bitów

        org $02E0               ; wektor RUNAD: adres uruchomienia programu
        dta a(start)            ; adres uruchomienia: start
```

Własny zestaw znaków to najprostsza droga do polskich liter, a także do szybkiej grafiki w grach: plansza zbudowana ze znaków zajmuje 40 razy mniej pamięci niż ta sama plansza w trybie bitowym, a przesunięcie „klocka” to zmiana jednego bajtu.

Ćwiczenia

1. Napisz procedurę `pisz_xy`, która wypisuje tekst ATASCII (zakończony EOL) bezpośrednio do pamięci ekranu od kolumny X i wiersza Y. Wykorzystaj `asc2int` i tablicę `wiersz_lo/wiersz_hi`.
2. Zmodyfikuj `dlist.asm` tak, by pierwsza linia trybu 2 miała własny LMS wskazujący inny obszar pamięci.
3. Zaprojektuj literę „ą” na siatce 8×8 i podmień nią wybrany znak.

ROZDZIAŁ 23

Klawiatura, joystick i klawisze konsoli

Program może odbierać polecenia użytkownika na kilka sposobów — od wygodnego, ale powolnego CIO, po bezpośredni odczyt rejestrów sprzętowych. Wybór zależy od tego, czy piszemy narzędzie (liczy się wygoda), czy grę (liczy się szybkość reakcji).

23.1 Klawiatura

23.1.1 Zmienna CH — ostatnio naciśnięty klawisz

Obsługą klawiatury zajmuje się przerwanie IRQ układu POKEY. Po każdym naciśnięciu klawisza system zapisuje jego kod do komórki **CH** (`$02FC`). Wartość `$FF` oznacza „nic nie naciśnięto". Odczyt nie kasuje zawartości — program sam musi wpisać tam `$FF` po obsłużeniu klawisza.

```
CH      = $02FC                 ; kod klawisza

        lda CH                  ; A = kod klawisza
        cmp #$FF                ; czy cokolwiek naciśnięto?
        beq nic                 ; nie
        ldx #$FF                ; tak: skasuj kod, żeby nie obsłużyć go drugi raz
        stx CH                  ; zapisz X do CH - kod klawisza
                                ; ... tutaj A zawiera kod klawisza ...
nic
```

Kod w CH **nie jest kodem ATASCII**. To numer klawisza wynikający z jego położenia w matrycy klawiatury: bity 0–5 to numer klawisza, **bit 6 oznacza wciśnięty SHIFT, bit 7 — wciśnięty CONTROL**.

Klawisz	Kod	Klawisz	Kod	Klawisz	Kod	Klawisz	Kod
A	`$3F`	J	`$01`	S	`$3E`	1	`$1F`
B	`$15`	K	`$05`	T	`$2D`	2	`$1E`
C	`$12`	L	`$00`	U	`$0B`	3	`$1A`
D	`$3A`	M	`$25`	V	`$10`	4	`$18`
E	`$2A`	N	`$23`	W	`$2E`	5	`$1D`
F	`$38`	O	`$08`	X	`$16`	6	`$1B`
G	`$3D`	P	`$0A`	Y	`$2B`	7	`$33`
H	`$39`	Q	`$2F`	Z	`$17`	8	`$35`
I	`$0D`	R	`$28`	0	`$32`	9	`$30`

Klawisz	Kod	Klawisz	Kod
RETURN	`$0C`	ESC	`$1C`
spacja	`$21`	TAB	`$2C`
BACKSPACE	`$34`	- (z CONTROL: kursor w górę)	`$0E`
= (z CONTROL: kursor w dół)	`$0F`	+ (z CONTROL: kursor w lewo)	`$06`
* (z CONTROL: kursor w prawo)	`$07`		

Przykład: CONTROL + A daje kod `$3F` + `$80` = `$BF`. Aby rozpoznać klawisz niezależnie od SHIFT i CONTROL, zamaskuj dwa najstarsze bity:

```
        lda CH                  ; A = kod klawisza
        and #%00111111          ; usuń bity SHIFT i CONTROL
        cmp #$1C                ; czy to klawisz ESC?
        beq wyjscie             ; skocz do wyjscie, gdy Z = 1 (równe / zero)
```

Badanie CH jest idealne do poleceń typu „naciśnij klawisz" i sterowania klawiszami w grach: jest szybkie i nie zatrzymuje programu.

23.1.2 Znak ATASCII bez CIO — procedura systemowa klawiatury

Gdy potrzebny jest kod ATASCII (z uwzględnieniem SHIFT, CONTROL, CAPS), najprościej poprosić o niego procedurę obsługi klawiatury w systemie. Można otworzyć urządzenie `K:` przez CIO, ale jest krótsza droga. System przechowuje dla każdego urządzenia tablicę adresów jego procedur; tablica klawiatury leży pod `$E420`, a jej trzecia pozycja (`$E424`/`$E425`) to adres procedury „pobierz znak" — **pomniejszony o 1**, czyli gotowy do użycia w sztuczce z `rts` z rozdziału 10:

Plik: getkey.asm

```
; getkey - czeka na naciśnięcie klawisza i zwraca jego kod ATASCII
; Wyjście:  A = kod ATASCII.  Niszczy: X, Y.
KEYBDV  = $E420                 ; tablica wektorów procedury obsługi klawiatury w ROM-ie

getkey  lda KEYBDV+5            ; starszy bajt (adresu procedury "pobierz znak" - 1)
        pha                     ; odłóż na stos
        lda KEYBDV+4            ; młodszy bajt
        pha                     ; odłóż na stos
        rts                     ; "powrót" pod ten adres = skok do procedury systemowej;
                                ; jej własny RTS wróci do tego, kto wywołał getkey
```

Procedura **czeka** na klawisz. Jeżeli program ma w tym czasie robić coś innego, najpierw sprawdź, czy klawisz w ogóle naciśnięto:

```
        lda CH                  ; czy jest klawisz do odebrania?
        cmp #$FF                ; porównaj A z: $FF
        beq brak                ; nie: nie wywołuj getkey, bo zatrzymałby program
        jsr getkey              ; tak: pobierz go jako kod ATASCII (to także skasuje CH)
brak
```

23.1.3 Czy klawisz jest nadal wciśnięty?

CH informuje o zdarzeniu naciśnięcia. Do sterowania w grach potrzebna bywa informacja o stanie: czy klawisz jest trzymany *w tej chwili*. Podaje ją rejestr SKSTAT (`$D20F`) układu POKEY:

- bit 2 = **0**, gdy jakikolwiek klawisz jest wciśnięty; jego numer zawiera rejestr KBCODE (`$D209`),
- bit 3 = **0**, gdy wciśnięty jest SHIFT.

```
SKSTAT  = $D20F                 ; status klawiatury i portu (odczyt)
KBCODE  = $D209                 ; kod klawisza (odczyt)

        lda SKSTAT              ; odczytaj stan klawiatury
        and #%00000100          ; bit 2: 0 = klawisz wciśnięty
        bne puszczony           ; bit = 1: żaden klawisz nie jest trzymany
        lda KBCODE              ; A = kod trzymanego klawisza (taki sam układ jak w CH)
        and #%00111111          ; A = A AND %00111111
puszczony
```

23.1.4 Klawisz BREAK

Klawisz BREAK nie ma zwykłego kodu. System ustawia po jego naciśnięciu zmienną BRKKEY (`$11`) na zero, a operacje CIO kończą się błędem 128. Program, który sam obsługuje BREAK, sprawdza tę zmienną i wpisuje tam z powrotem wartość niezerową.

23.2 Klawisze konsoli: START, SELECT, OPTION

Trzy klawisze funkcyjne są podłączone do układu GTIA i odczytuje się je z rejestru **CONSOL** (`$D01F`). Każdemu odpowiada jeden bit, a **zero oznacza klawisz wciśnięty**:

Bit	Maska	Klawisz
0	`%001`	START
1	`%010`	SELECT
2	`%100`	OPTION

Gdy żaden nie jest wciśnięty, odczyt daje 7.

```
CONSOL  = $D01F                 ; klawisze START, SELECT, OPTION

        lda CONSOL              ; odczytaj klawisze konsoli
        lsr @                   ; bit 0 (START) -> C
        bcc start_wcis          ; C = 0: START wciśnięty
        lsr @                   ; bit 1 (SELECT) -> C
        bcc select_wcis         ; C = 0: SELECT wciśnięty
        lsr @                   ; bit 2 (OPTION) -> C
        bcc option_wcis         ; C = 0: OPTION wciśnięty
```

Klawisz HELP w modelach XL/XE zgłasza się w zmiennej HELPFG (`$02DC`): wartość 17 oznacza naciśnięcie; program powinien ją potem wyzerować.

23.2.1 Eliminacja wielokrotnych odczytów

Program sprawdzający klawisz w pętli wykona się tysiące razy, zanim użytkownik zdąży cofnąć palec. Jeśli naciśnięcie ma wywołać jedną akcję (np. przełączenie opcji w menu), trzeba poczekać na zwolnienie klawisza:

```
        lda CONSOL              ; czy START jest wciśnięty?
        and #1                  ; A = A AND 1
        bne nie_start           ; nie
        jsr przelacz            ; tak: wykonaj akcję...
zwolnij lda CONSOL              ; ...i zaczekaj,
        and #1                  ; aż klawisz zostanie puszczony
        beq zwolnij             ; skocz do zwolnij, gdy Z = 1 (równe / zero)
nie_start
```

23.3 Joystick

Joystick to pięć styków: cztery kierunki i przycisk. System w każdym przerwaniu VBI odczytuje je i zapisuje w cieniach:

Zmienna	Adres	Zawartość
STICK0	`$0278`	kierunki joysticka w porcie 1
STICK1	`$0279`	kierunki joysticka w porcie 2
STRIG0	`$0284`	przycisk joysticka 1: **0 = wciśnięty**, 1 = puszczony
STRIG1	`$0285`	przycisk joysticka 2

W zmiennej STICK cztery najmłodsze bity odpowiadają kierunkom, a **zero oznacza wychylenie**:

```
   bit 0 (maska 1): góra         bit 2 (maska 4): lewo
   bit 1 (maska 2): dół          bit 3 (maska 8): prawo

                14
           10    |     6              15 = joystick w położeniu środkowym
              \  |  /
         11 ——— 15 ——— 7
```

```
            /  |  \
          9    |    5
              13
```

Zamiast porównywać z dziewięcioma wartościami, najlepiej badać bity — skosy obsługują się wtedy same, bo ruch w pionie i w poziomie sprawdzamy niezależnie:

```
STICK0  = $0278                 ; joystick 0

        lda STICK0              ; A = stan joysticka
        lsr @                   ; bit 0 (góra) -> C
        bcs nie_gora            ; C = 1: nie wychylony w górę
        dec poz_y               ; C = 0: ruch w górę
nie_gora lsr @                  ; bit 1 (dół) -> C
        bcs nie_dol             ; skocz do nie_dol, gdy C = 1
        inc poz_y               ; ruch w dół
nie_dol lsr @                   ; bit 2 (lewo) -> C
        bcs nie_lewo            ; skocz do nie_lewo, gdy C = 1
        dec poz_x               ; ruch w lewo
nie_lewo lsr @                  ; bit 3 (prawo) -> C
        bcs nie_prawo           ; skocz do nie_prawo, gdy C = 1
        inc poz_x               ; ruch w prawo
nie_prawo
```

Rozkazy `inc` i `dec` nie zmieniają akumulatora ani znacznika C, więc kolejne `lsr` działają na tej samej, stopniowo przesuwanej wartości.

Bezpośrednio ze sprzętu joysticki odczytuje się z rejestru PORTA (`$D300`: młodszy półbajt to port 1, starszy — port 2), a przyciski z TRIG0 i TRIG1 (`$D010`, `$D011`). Jest to potrzebne tylko wtedy, gdy systemowe przerwanie VBI jest wyłączone.

23.4 Program: ruchomy znak sterowany joystickiem

Program łączy bezpośredni zapis do pamięci ekranu, odczyt joysticka i synchronizację z ramką obrazu. Gwiazdka porusza się po ekranie, nie wychodząc poza jego krawędzie; przycisk joysticka albo klawisz START kończy program.

Plik: joy.asm

```
; ------------------------------------------------------------------------
; joy.asm - znak poruszany joystickiem po ekranie tekstowym
; ------------------------------------------------------------------------
SAVMSC  = $58                   ; adres pamięci ekranu
RTCLOK  = $14                   ; zegar systemowy (rośnie co ramkę)
ATRACT  = $4D                   ; licznik wygaszacza ekranu
STICK0  = $0278                 ; kierunki joysticka
STRIG0  = $0284                 ; przycisk joysticka (0 = wciśnięty)
CRSINH  = $02F0                 ; ukrycie kursora
CONSOL  = $D01F                 ; klawisze konsoli

wsk     = $80                   ; wskaźnik na wiersz ekranu (2 bajty)
poz_x   = $82                   ; kolumna znaku (0..39)
poz_y   = $83                   ; wiersz znaku (0..23)

        org $2000               ; adres asemblacji: $2000

start   lda #1                  ; ukryj kursor
        sta CRSINH              ; zapisz A do CRSINH - ukrycie kursora
        lda #20                 ; pozycja początkowa: środek ekranu
        sta poz_x               ; zapisz A do poz_x
        lda #12                 ; A = 12
        sta poz_y               ; zapisz A do poz_y

petla   lda #"*"                ; narysuj gwiazdkę (kod ekranowy) w bieżącym miejscu
        jsr postaw              ; wywołaj podprogram postaw

        ldx #3                  ; odczekaj 3 ramki - to ustala prędkość ruchu
czekaj  lda RTCLOK              ; A = RTCLOK - zegar systemowy (3 bajty; $14 = najmłodszy)
cz1     cmp RTCLOK              ; czekaj na zmianę zegara, czyli na następną ramkę
```

```
        beq cz1                 ; skocz do cz1, gdy Z = 1 (równe / zero)
        dex                     ; X = X - 1
        bne czekaj              ; skocz do czekaj, gdy Z = 0 (różne / nie zero)

        lda STRIG0              ; przycisk joysticka wciśnięty?
        beq koniec              ; tak (0): zakończ
        lda CONSOL              ; klawisz START wciśnięty?
        and #1                  ; A = A AND 1
        beq koniec              ; tak (bit 0 = 0): zakończ

        lda STICK0              ; czy joystick jest wychylony?
        cmp #15                 ; porównaj A z: 15
        beq petla               ; nie (15 = środek): nic się nie zmienia

        lda #0                  ; joystick wychylony: odsuń wygaszacz ekranu
        sta ATRACT              ; zapisz A do ATRACT - licznik wygaszacza ekranu
        lda #" "                ; zmaż znak w starym miejscu (spacja)
        jsr postaw              ; wywołaj podprogram postaw

        lda STICK0              ; A = stan joysticka; badamy bity po kolei
        lsr @                   ; bit 0: góra
        bcs r1                  ; skocz do r1, gdy C = 1
        ldx poz_y               ; czy jesteśmy już w wierszu 0 ?
        beq r1                  ; tak: nie wychodź poza ekran
        dec poz_y               ; nie: wiersz w górę
r1      lsr @                   ; bit 1: dół
        bcs r2                  ; skocz do r2, gdy C = 1
        ldx poz_y               ; X = poz_y
        cpx #23                 ; czy jesteśmy w ostatnim wierszu?
        beq r2                  ; skocz do r2, gdy Z = 1 (równe / zero)
        inc poz_y               ; nie: wiersz w dół
r2      lsr @                   ; bit 2: lewo
        bcs r3                  ; skocz do r3, gdy C = 1
        ldx poz_x               ; czy jesteśmy w kolumnie 0 ?
        beq r3                  ; skocz do r3, gdy Z = 1 (równe / zero)
        dec poz_x               ; nie: kolumna w lewo
r3      lsr @                   ; bit 3: prawo
        bcs r4                  ; skocz do r4, gdy C = 1
        ldx poz_x               ; X = poz_x
        cpx #39                 ; czy jesteśmy w ostatniej kolumnie?
        beq r4                  ; skocz do r4, gdy Z = 1 (równe / zero)
        inc poz_x               ; nie: kolumna w prawo
r4      jmp petla               ; narysuj znak w nowym miejscu

koniec  lda #0                  ; przywróć kursor
        sta CRSINH              ; zapisz A do CRSINH - ukrycie kursora
        rts                     ; powrót z podprogramu

; --- postaw: wpisuje znak z A do pamięci ekranu w kolumnie poz_x, wierszu poz_y -----
postaw  pha                     ; zachowaj znak
        ldy poz_y               ; Y = numer wiersza
        lda SAVMSC              ; wsk := SAVMSC + przesunięcie wiersza
        clc                     ; C = 0 (przed dodawaniem)
        adc wiersz_lo,y         ; A = A + bajt spod adresu wiersz_lo+Y + C
        sta wsk                 ; zapisz A do wsk
        lda SAVMSC+1            ; A = bajt spod adresu SAVMSC+1
        adc wiersz_hi,y         ; A = A + bajt spod adresu wiersz_hi+Y + C
        sta wsk+1               ; zapisz A pod adres wsk+1
        ldy poz_x               ; Y = kolumna
        pla                     ; odtwórz znak
        sta (wsk),y             ; wpisz go do pamięci ekranu
        rts                     ; powrót z podprogramu

wiersz_lo dta l(0,40,80,120,160,200,240,280,320,360,400,440)  ; młodsze bajty
        dta l(480,520,560,600,640,680,720,760,800,840,880,920)  ; młodsze bajty
wiersz_hi dta h(0,40,80,120,160,200,240,280,320,360,400,440)  ; starsze bajty
        dta h(480,520,560,600,640,680,720,760,800,840,880,920)  ; starsze bajty

        org $02E0               ; wektor RUNAD: adres uruchomienia programu
        dta a(start)            ; adres uruchomienia: start
```

Zwróć uwagę na wykorzystanie rejestru X do sprawdzania krawędzi: akumulator przechowuje stopniowo przesuwany stan joysticka i nie wolno go zniszczyć, a rozkazy `ldx` i `cpx` nie zmieniają akumulatora.

Ćwiczenia

1. Zmień program tak, by gwiazdka zostawiała za sobą ślad z kropek.
2. Dodaj sterowanie klawiszami kursora (kody `$0E`, `$0F`, `$06`, `$07` w zmiennej CH).
3. Napisz procedurę `tak_nie`, która czeka na klawisz T lub N i zwraca C = 0 dla „tak”, C = 1 dla „nie”, ignorując inne klawisze.

ROZDZIAŁ 24

Przerwania: VBI, DLI, IRQ

Przerwania to mechanizm, dzięki któremu Atari robi kilka rzeczy „naraz”: gdy twój program liczy, system w tle obsługuje klawiaturę, odświeża kolory, tyka zegarem. Możesz dołączyć do tego mechanizmu własne procedury. To klucz do muzyki grającej w tle, płynnej animacji i efektów graficznych niemożliwych do uzyskania inaczej.

24.1 Rodzaje przerwań w Atari

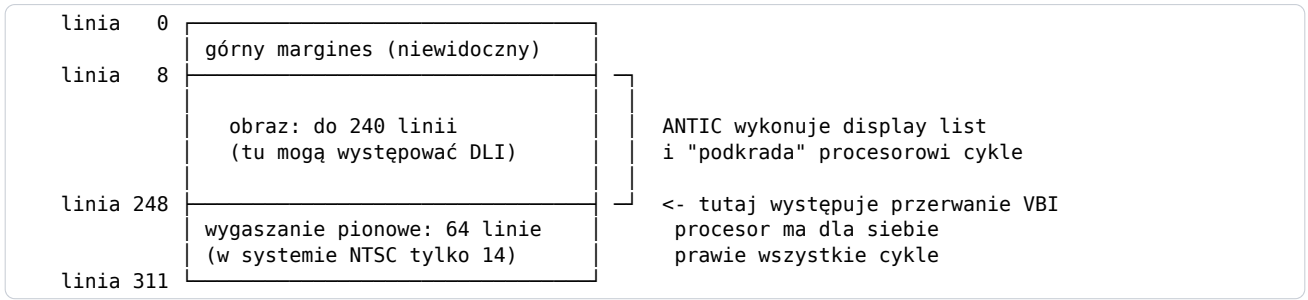
Przerwanie	Rodzaj	Kiedy występuje	Typowe zastosowanie
VBI (*Vertical Blank Interrupt*)	NMI	raz na ramkę, gdy plamka wraca na górę ekranu (PAL: 50 razy na sekundę)	muzyka, ruch obiektów, odmierzanie czasu
DLI (*Display List Interrupt*)	NMI	w liniach ekranu wskazanych w display list	zmiana kolorów, zestawu znaków, pozycji duszków w połowie ekranu
IRQ	IRQ	klawiatura, liczniki POKEY, transmisja szeregowa, klawisz BREAK	precyzyjne odmierzanie czasu, odtwarzanie sampli, transmisja

Przerwania NMI włącza się i wyłącza w rejestrze **NMIEN** (`$D40E`): bit 6 (`$40`) to VBI, bit 7 (`$80`) to DLI. Normalnie rejestr zawiera `$40` — działa tylko VBI. Rozkaz procesora `sei` na przerwania NMI nie wpływa.

Przerwania IRQ włącza się pojedynczymi bitami rejestru IRQEN (`$D20E`) układu POKEY, a blokuje wszystkie naraz rozkazem `sei`.

24.2 Ramka obrazu i czas

Żeby rozumieć przerwania, trzeba mieć w głowie obraz jednej ramki. W systemie PAL ramka to **312 linii**, z których każda trwa **114 cykli** procesora:



Cała ramka to około 35 500 cykli, ale w czasie wyświetlania obrazu ANTIC zatrzymuje procesor, żeby pobrać dane z pamięci — w trybie tekstowym zabiera mniej więcej jedną trzecią czasu. Wygaszanie pionowe to okres, w którym procesor pracuje bez przeszkód, a zmiany na ekranie są niewidoczne. Dlatego to najlepszy moment na modyfikowanie obrazu.

24.3 Przerwanie VBI

Systemowa obsługa VBI składa się z dwóch etapów, a każdy ma własny wektor, pod który można się „wpiąć”:

```
przerwanie NMI
     │
     ▼
zachowaj rejestry A, X, Y na stosie
     │
     ▼
jmp (VVBLKI)  ──►  [twoja procedura natychmiastowa]  ──►  jmp SYSVBV
     │
     ▼                                              ETAP 1 (zawsze):
SYSVBV: zwiększ zegar RTCLOK, obsłuż licznik 1,     krótki, krytyczny czasowo
        wygaszacz ekranu
     │
     ├── jeśli przerwano sekcję krytyczną (CRITIC <> 0 lub znacznik I = 1) ──► koniec
     ▼
przepisz cienie do rejestrów sprzętowych,           ETAP 2 (gdy wolno):
odczytaj joysticki, obsłuż liczniki 2-5             dłuższy
     │
     ▼
jmp (VVBLKD)  ──►  [twoja procedura opóźniona]  ──►  jmp XITVBV
     │
     ▼
XITVBV: odtwórz rejestry Y, X, A;  rti
```

	VBI natychmiastowe	VBI opóźnione
Wektor	VVBLKI (`$0222`)	VVBLKD (`$0224`)
Wywoływane	zawsze, na początku	po zadaniach systemu; **pomijane** podczas operacji dyskowych
Zakończenie procedury	`jmp $E45F` (SYSVBV)	`jmp $E462` (XITVBV)
Dozwolony czas	bardzo krótki	długi — nawet kilkanaście tysięcy cykli
Zastosowanie	muzyka, która nie może się zacinać	wszystko inne

Rejestrów nie trzeba zachowywać — system zrobił to wcześniej. W procedurze VBI można swobodnie używać A, X i Y.

24.3.1 Instalowanie procedury: SETVBV

Wektor ma dwa bajty. Gdyby przerwanie nadeszło między zapisaniem pierwszego a drugiego, procesor skoczyłby pod adres „w połowie zmieniony". Dlatego wektory VBI ustawia się przez procedurę systemową **SETVBV** (`$E45C`), która robi to bezpiecznie:

Rejestr	Zawartość
A	6 = wektor natychmiastowy, 7 = wektor opóźniony
X	**starszy** bajt adresu procedury
Y	**młodszy** bajt adresu procedury

24.3.2 Program: zegar i pulsująca ramka w tle

Procedura VBI co ramkę zmienia kolor ramki i liczy sekundy, podczas gdy „program główny" zajmuje się czymś zupełnie innym — tu po prostu czeka na klawisz.

Plik: vbi.asm

```
; ------------------------------------------------------------------------
; vbi.asm - własna procedura w opóźnionym przerwaniu VBI
; ------------------------------------------------------------------------
SETVBV  = $E45C                 ; systemowa procedura ustawiania wektorów VBI
XITVBV  = $E462                 ; systemowe zakończenie przerwania VBI
COLOR4  = $02C8                 ; cień koloru ramki
CH      = $02FC                 ; kod klawisza
SAVMSC  = $58                   ; adres pamięci ekranu

ramki   = $80                   ; licznik ramek w bieżącej sekundzie (0..49)
```

```
sekundy = $81                   ; licznik sekund (0..9, potem od nowa)

        org $2000               ; adres asemblacji: $2000

start   lda #0                  ; wyzeruj liczniki, ZANIM przerwanie zacznie ich używać
        sta ramki               ; zapisz A do ramki
        sta sekundy             ; zapisz A do sekundy

        lda #7                  ; A = 7: ustawiamy wektor VBI opóźnionego
        ldx #>moje_vbi          ; X = starszy bajt adresu procedury
        ldy #<moje_vbi          ; Y = młodszy bajt adresu procedury
        jsr SETVBV              ; od tej chwili "moje_vbi" wykonuje się 50 razy na sekundę

        lda #$FF                ; --- program główny: czekaj na klawisz ---
        sta CH                  ; zapisz A do CH - kod klawisza
czekaj  lda CH                  ; A = CH - kod klawisza
        cmp #$FF                ; porównaj A z: $FF
        beq czekaj              ; skocz do czekaj, gdy Z = 1 (równe / zero)
        lda #$FF                ; A = $FF
        sta CH                  ; zapisz A do CH - kod klawisza

        lda #7                  ; --- sprzątanie: przywróć systemowy wektor VBI ---
        ldx #>XITVBV            ; X = starszy bajt wartości XITVBV
        ldy #<XITVBV            ; Y = młodszy bajt wartości XITVBV
        jsr SETVBV              ; wywołaj SETVBV: ustawianie wektorów VBI i liczników
        lda #0                  ; przywróć czarną ramkę
        sta COLOR4              ; zapisz A do COLOR4 - kolor tła (w trybie tekstowym: ramka)
        rts                     ; powrót z podprogramu

; --- procedura wykonywana w każdym przerwaniu VBI ----------------------------
moje_vbi inc COLOR4             ; zmień kolor ramki (zapis do cienia; system przepisze go
                                ; do rejestru sprzętowego w następnym przerwaniu)
        inc ramki               ; policz ramkę
        lda ramki               ; A = ramki
        cmp #50                 ; czy minęło 50 ramek, czyli sekunda (system PAL)?
        bne mv_k                ; jeszcze nie
        lda #0                  ; tak: zacznij liczyć ramki od nowa
        sta ramki               ; zapisz A do ramki
        inc sekundy             ; sekundy + 1
        lda sekundy             ; A = sekundy
        cmp #10                 ; po 9 następuje 0
        bne mv_1                ; skocz do mv_1, gdy Z = 0 (różne / nie zero)
        lda #0                  ; A = 0
        sta sekundy             ; zapisz A do sekundy
mv_1    ora #"0"                ; zamień cyfrę na kod ekranowy
        ldy #39                 ; ostatnia kolumna pierwszego wiersza
        sta (SAVMSC),y          ; wpisz cyfrę wprost do pamięci ekranu
mv_k    jmp XITVBV              ; OBOWIĄZKOWE zakończenie procedury VBI opóźnionego

        org $02E0               ; wektor RUNAD: adres uruchomienia programu
        dta a(start)            ; adres uruchomienia: start
```

Pułapka

Zasady bezpieczeństwa dla procedur VBI:

1. Kończ je **skokiem** `jmp XITVBV` (lub `jmp SYSVBV` dla natychmiastowej), nigdy `rts` ani `rti`.
2. **Nie wywołuj CIO** ani innych długich procedur systemowych — przerwanie mogło nadejść w trakcie takiej samej operacji programu głównego.
3. **Używaj własnych komórek strony zerowej**, innych niż program główny. Jeśli przerwanie zmieni wskaźnik, którego właśnie używa program główny, skutki będą nieprzewidywalne.
4. **Przed wyjściem z programu odinstaluj procedurę** — inaczej po załadowaniu następnego programu wektor wskaże przypadkowy kod.
5. Zmienne wielobajtowe dzielone z programem głównym odczytuj ostrożnie: przerwanie może je zmienić między odczytem młodszego i starszego bajtu.

24.3.3 Liczniki systemowe

System utrzymuje pięć 16-bitowych liczników zmniejszanych w każdym VBI. Najprostsze w użyciu są liczniki 3, 4 i 5: gdy licznik dojdzie do zera, system zeruje odpowiadającą mu flagę.

Licznik	Adres	Gdy dojdzie do zera
1	CDTMV1 `$0218`	skok przez wektor CDTMA1 (`$0226`) — w etapie 1
2	CDTMV2 `$021A`	skok przez wektor CDTMA2 (`$0228`) — w etapie 2
3	CDTMV3 `$021C`	zerowana flaga CDTMF3 (`$022A`)
4	CDTMV4 `$021E`	zerowana flaga CDTMF4 (`$022C`)
5	CDTMV5 `$0220`	zerowana flaga CDTMF5 (`$022E`)

Licznik ustawia się także przez SETVBV: A = numer licznika (1–5), X = starszy bajt wartości, Y = młodszy.

```
; Pauza 2 sekundy (100 ramek) z użyciem licznika systemowego nr 3
        lda #1                  ; ustaw flagę licznika 3 na wartość niezerową
        sta $022A               ; CDTMF3
        lda #3                  ; A = 3: numer licznika
        ldx #>100               ; X = starszy bajt liczby ramek
        ldy #<100               ; Y = młodszy bajt liczby ramek
        jsr $E45C               ; SETVBV: uruchom odliczanie
czek    lda $022A               ; czekaj, aż system wyzeruje flagę
        bne czek                ; skocz do czek, gdy Z = 0 (różne / nie zero)
```

24.4 Przerwanie DLI

DLI to specjalność Atari. Pozwala wykonać krótki fragment kodu **w ściśle określonej linii ekranu** i zmienić w tym miejscu parametry obrazu. Skutek: więcej kolorów naraz, niż wynikałoby z liczby rejestrów, kilka zestawów znaków na jednym ekranie, duszki użyte wielokrotnie.

Przygotowanie wymaga trzech kroków:

1. **Oznacz linię w display list** — ustaw bit 7 w rozkazie tej linii (dodaj `$80`). Przerwanie wystąpi w ostatniej linii obrazu należącej do tak oznaczonego wiersza.
2. **Wpisz adres procedury** do wektora VDSLST (`$0200/$0201`).
3. **Włącz DLI**: wpisz `$C0` do NMIEN (bit 7 = DLI, bit 6 = VBI).

Procedura DLI musi spełniać surowe wymagania, bo **system niczego za nią nie robi**:

- sama zachowuje każdy używany rejestr (`pha`; a jeśli używa X lub Y — także je),
- zapisuje do rejestrów **sprzętowych**, nie do cieni (cienie zadziałałyby dopiero w następnej ramce i na całym ekranie),
- kończy się rozkazem **`rti`**,
- jest krótka: do następnej linii zostało kilkadziesiąt cykli.

24.4.1 Rejestr WSYNC

Zmiana koloru w środku linii daje brzydki „schodek”. Rozwiązaniem jest rejestr **WSYNC** (`$D40A`): zapis dowolnej wartości zatrzymuje procesor do końca bieżącej linii obrazu. Następny rozkaz wykona się, gdy plamka jest poza widoczną częścią ekranu — zmiana koloru obejmie całą następną linię.

```
dli     pha                     ; zachowaj akumulator - jedyny rejestr, którego używamy
        lda #$34                ; nowy kolor (czerwony)
        sta $D40A               ; WSYNC: poczekaj do końca bieżącej linii
        sta $D018               ; COLPF2: zmień kolor tła - obowiązuje od następnej linii
        pla                     ; odtwórz akumulator
        rti                     ; wróć do przerwanego programu
```

Na górze ekranu system (w przerwaniu VBI) przepisze cień COLOR2 do rejestru COLPF2 i pierwotny kolor wróci. Efekt: dwie części ekranu w różnych kolorach.

24.4.2 Program: tęczowy pas na ekranie tekstowym

Program wstawia DLI w jednym z wierszy standardowego ekranu tekstowego. Procedura przerwania zmienia kolor tła **w każdej z 16 kolejnych linii obrazu**, tworząc poziomy pas barw — efekt niemożliwy do uzyskania bez przerwań.

Plik: dli.asm

```
; ------------------------------------------------------------------------
; dli.asm - przerwanie DLI: pas kolorów w poprzek ekranu tekstowego
; ------------------------------------------------------------------------
SDLSTL  = $0230                 ; adres display list (cień)
VDSLST  = $0200                 ; wektor przerwania DLI
COLOR2  = $02C6                 ; cień koloru tła trybu tekstowego
CH      = $02FC                 ; kod klawisza
RTCLOK  = $14                   ; zegar systemowy
COLPF2  = $D018                 ; sprzętowy rejestr koloru tła trybu tekstowego
WSYNC   = $D40A                 ; synchronizacja z końcem linii
NMIEN   = $D40E                 ; maska przerwań NMI

WIERSZ  = 10                    ; numer wiersza tekstu, po którym zaczyna się pas

dl      = $80                   ; wskaźnik na display list (2 bajty)

        org $2000               ; adres asemblacji: $2000

start   lda SDLSTL              ; dl := adres systemowej display list
        sta dl                  ; zapisz A do dl
        lda SDLSTL+1            ; A = bajt spod adresu SDLSTL+1
        sta dl+1                ; zapisz A pod adres dl+1

; Układ systemowej display list trybu tekstowego:
;   bajty 0-2: trzy rozkazy pustych linii ($70)
;   bajty 3-5: rozkaz wiersza 0 z LMS ($42) i dwa bajty adresu
;   bajt 6:    rozkaz wiersza 1,  bajt 7: wiersza 2, ...  bajt 5+N: wiersza N
        ldy #5+WIERSZ           ; Y = pozycja rozkazu wybranego wiersza
        lda (dl),y              ; pobierz rozkaz ($02)
        ora #$80                ; ustaw bit 7: po tym wierszu wywołaj DLI
        sta (dl),y              ; zapisz z powrotem ($82)

        lda #<dli               ; wpisz adres procedury do wektora DLI;
        sta VDSLST              ; DLI jest jeszcze wyłączone, więc można to zrobić
        lda #>dli               ; zwykłymi rozkazami
        sta VDSLST+1            ; zapisz A pod adres VDSLST+1

        lda RTCLOK              ; poczekaj na początek ramki, żeby nie włączyć DLI
cz0     cmp RTCLOK              ; w środku rysowania ekranu
        beq cz0                 ; skocz do cz0, gdy Z = 1 (równe / zero)
        lda #$C0                ; włącz DLI (bit 7), pozostaw VBI (bit 6)
        sta NMIEN               ; zapisz A do NMIEN - zapis: maska przerwań NMI

        lda #$FF                ; --- program główny: czekaj na klawisz ---
        sta CH                  ; zapisz A do CH - kod klawisza
czekaj  lda CH                  ; A = CH - kod klawisza
        cmp #$FF                ; porównaj A z: $FF
        beq czekaj              ; skocz do czekaj, gdy Z = 1 (równe / zero)
        lda #$FF                ; A = $FF
        sta CH                  ; zapisz A do CH - kod klawisza

        lda #$40                ; --- sprzątanie: wyłącz DLI, zostaw VBI ---
        sta NMIEN               ; zapisz A do NMIEN - zapis: maska przerwań NMI
        ldy #5+WIERSZ           ; usuń bit DLI z display list
        lda (dl),y              ; A = bajt spod adresu (dl) + Y
        and #$7F                ; A = A AND $7F
        sta (dl),y              ; zapisz A pod adres (dl) + Y
        rts                     ; powrót z podprogramu

; --- procedura przerwania DLI ------------------------------------------------
dli     pha                     ; zachowaj A
        txa                     ; A = X
        pha                     ; zachowaj X
        ldx #0                  ; X = numer linii pasa
dli_1   lda kolory,x            ; kolor dla kolejnej linii obrazu
```

```
        sta WSYNC               ; poczekaj do końca bieżącej linii
        sta COLPF2              ; ustaw kolor - obowiązuje w następnej linii
        inx                     ; X = X + 1
        cpx #ILE_KOL            ; wszystkie linie pasa?
        bne dli_1               ; skocz do dli_1, gdy Z = 0 (różne / nie zero)
        lda COLOR2              ; na koniec przywróć zwykły kolor tła (odczytany z cienia),
        sta WSYNC               ; aby reszta ekranu wyglądała normalnie
        sta COLPF2              ; zapisz A do COLPF2 - kolor pola gry 2
        pla                     ; odtwórz X
        tax                     ; X = A
        pla                     ; odtwórz A
        rti                     ; koniec przerwania

kolory  dta $10,$22,$34,$46,$58,$6A,$7C,$8E  ; jasności rosnące, odcienie 1..8
        dta $9E,$AC,$BA,$C8,$D6,$E4,$F2,$10  ; jasności malejące, odcienie 9..15
ILE_KOL = *-kolory              ; liczba kolorów w tablicy (obliczana przez asembler)

        org $02E0               ; wektor RUNAD: adres uruchomienia programu
        dta a(start)            ; adres uruchomienia: start
```

Pas ma 16 linii obrazu, czyli wysokość dwóch wierszy tekstu. Tekst w tych wierszach pozostaje czytelny — zmienia się tylko kolor tła pod nim.

24.4.3 Kilka przerwań DLI na jednym ekranie

Wektor VDSLST jest jeden, a linii z bitem DLI może być wiele. Każde przerwanie musi więc „wiedzieć", które jest z kolei. Trzy popularne sposoby:

- **łańcuch**: każda procedura DLI na końcu wpisuje do VDSLST adres następnej, a procedura VBI przywraca adres pierwszej,
- **licznik**: jedna procedura zwiększa zmienną i używa jej jako indeksu do tablicy kolorów; VBI zeruje licznik,
- **rejestr VCOUNT** (`$D40B`): zawiera numer bieżącej linii obrazu podzielony przez 2 — procedura sama sprawdza, gdzie jest.

24.5 Przerwania IRQ

Przerwania maskowalne w Atari pochodzą głównie z układu POKEY. System rozpoznaje źródło i skacze przez odpowiedni wektor w pamięci RAM:

Wektor	Adres	Źródło	Bit w IRQEN
VIMIRQ	`$0216`	każde IRQ — główny wektor, przed rozpoznaniem źródła	—
VKEYBD	`$0208`	naciśnięcie klawisza	6 (`$40`)
VBREAK	`$0206`	rozkaz `brk`	—
VTIMR1	`$0210`	licznik 1 układu POKEY	0 (`$01`)
VTIMR2	`$0212`	licznik 2 układu POKEY	1 (`$02`)
VTIMR4	`$0214`	licznik 4 układu POKEY	2 (`$04`)
VSERIN	`$020A`	odebrano bajt z portu szeregowego	5 (`$20`)

Rejestr IRQEN jest tylko do zapisu, więc system trzyma jego kopię w zmiennej POKMSK (`$10`). Zmieniając maskę przerwań, zawsze modyfikuj kopię i zapisuj ją do obu miejsc.

Procedury wywoływane przez te wektory system poprzedza odłożeniem akumulatora na stos, więc kończą się parą **`pla` / `rti`**. Rejestry X i Y trzeba zachować samodzielnie.

Liczniki układu POKEY — te same, które generują dźwięk — mogą wywoływać przerwania z częstotliwością od kilkudziesięciu herców do kilkudziesięciu kiloherców. Na tym opiera się odtwarzanie dźwięku cyfrowego i szybka transmisja danych. To już jednak zagadnienia zaawansowane; wymagają też wyłączenia lub ostrożnego traktowania kanałów dźwiękowych używanych jako liczniki.

24.6 Sekcje krytyczne

Czasem fragment programu nie może zostać przerwany — bo wymaga dokładnego czasu albo zmienia konfigurację pamięci, przy której procedury przerwań przestałyby działać. Pełne wyłączenie przerwań wygląda tak:

```
        sei                     ; zablokuj przerwania IRQ
        lda #0                  ; A = 0
        sta $D40E               ; NMIEN = 0: wyłącz przerwania NMI (VBI i DLI)
                                ; ... sekcja krytyczna ...
        lda #$40                ; A = $40
        sta $D40E               ; NMIEN = $40: włącz z powrotem VBI
        cli                     ; odblokuj IRQ
```

W czasie takiej sekcji stoi zegar systemowy, nie działa klawiatura i nie są odświeżane kolory. Powinna trwać jak najkrócej. Łagodniejszym sposobem jest ustawienie zmiennej **CRITIC** (`$42`) na wartość niezerową: system wykonuje wtedy tylko pierwszy, krótki etap VBI (zegar tyka), pomijając resztę.

Ćwiczenia

1. Zmień `vbi.asm` tak, by wyświetlał dwucyfrowy licznik sekund (00–59).
2. Dodaj do `dli.asm` drugi pas kolorów kilka wierszy niżej, stosując metodę łańcucha: pierwsza procedura DLI przestawia wektor na drugą, a druga — z powrotem na pierwszą.
3. Dlaczego w procedurze DLI nie wolno zapisywać kolorów do cieni (np. `$02C6`)?

ROZDZIAŁ 25

Dźwięk i liczby losowe: układ POKEY

Układ POKEY ma cztery niezależne kanały dźwiękowe. Każdy kanał to dzielnik częstotliwości z regulacją głośności i wyborem „zniekształcenia", czyli rodzaju brzmienia: od czystego tonu po różne odmiany szumu. Ten sam układ dostarcza też liczb losowych.

25.1 Rejestry dźwięku

Rejestr	Adres	Znaczenie
AUDF1 – AUDF4	`$D200`, `$D202`, `$D204`, `$D206`	częstotliwość kanału (dzielnik 0–255)
AUDC1 – AUDC4	`$D201`, `$D203`, `$D205`, `$D207`	głośność i barwa kanału
AUDCTL	`$D208`	ustawienia wspólne dla wszystkich kanałów
SKCTL	`$D20F`	sterowanie portem szeregowym — wpływa także na dźwięk

25.1.1 AUDC — głośność i barwa

```
bit:     7   6   5   4   3   2   1   0
       ┌───┬───┬───┬───┬───┬───┬───┬───┐
       │ zniekształc.  │ V │   głośność    │
       └───┴───┴───┴───┴───┴───┴───┴───┘
        barwa dźwięku  │     0 = cisza, 15 = najgłośniej
                       └──── 1 = tylko poziom głośności (do odtwarzania sampli)
```

Starszy półbajt AUDC	Brzmienie	Zastosowanie
`$A_`	czysty ton	melodie
`$C_`	ton brzęczący, basowy	bas, silniki
`$8_`	biały szum	wybuchy, wiatr, perkusja
`$0_`, `$2_`, `$4_`	różne odmiany szumu i warkotu	efekty specjalne

Przykład: `$A8` to czysty ton o głośności 8; `$86` to szum o głośności 6. Suma głośności wszystkich czterech kanałów nie powinna przekraczać około 32 — inaczej dźwięk ulega przesterowaniu.

25.1.2 AUDF — częstotliwość

Wysokość dźwięku zależy od dzielnika: **im większa wartość AUDF, tym niższy dźwięk**. Dla czystego tonu przy standardowych ustawieniach:

```
częstotliwość [Hz]  ≈  31 700 / (AUDF + 1)
```

Wartości dla jednej oktawy (takie same, jak w instrukcji SOUND w BASIC-u):

Dźwięk	C	D	E	F	G	A	H	C (wyższe)
AUDF	121	108	96	91	81	72	64	60

25.1.3 AUDCTL — ustawienia wspólne

Bit	Wartość	Działanie po ustawieniu
0	`$01`	częstotliwość bazowa 15 kHz zamiast 64 kHz (dźwięki cztery razy niższe)
1	`$02`	filtr górnoprzepustowy kanału 2 sterowany kanałem 4
2	`$04`	filtr górnoprzepustowy kanału 1 sterowany kanałem 3
3	`$08`	połączenie kanałów 3 i 4 w jeden o 16-bitowym dzielniku
4	`$10`	połączenie kanałów 1 i 2 w jeden o 16-bitowym dzielniku
5	`$20`	kanał 3 taktowany zegarem 1,77 MHz
6	`$40`	kanał 1 taktowany zegarem 1,77 MHz
7	`$80`	krótszy generator szumu (inne brzmienie szumów)

Dzielnik ośmiobitowy daje niewiele ponad trzy oktawy o przyzwoitej dokładności stroju. Połączenie dwóch kanałów w jeden 16-bitowy (bity 3 i 4) daje pełną skalę z doskonałym strojem — kosztem liczby głosów.

25.1.4 Inicjalizacja

Po operacjach dyskowych POKEY pozostaje w konfiguracji transmisji szeregowej. Przed użyciem dźwięku trzeba go przywrócić do stanu podstawowego — inaczej kanały zabrzmią fałszywie albo wcale:

```
        lda #0                  ; A = 0
        sta $D208               ; AUDCTL = 0: ustawienia standardowe
        lda #3                  ; A = 3
        sta $D20F               ; SKCTL = 3: normalny tryb pracy układu POKEY
```

25.2 Program: gama

Plik: gama.asm

```
; ------------------------------------------------------------------------
; gama.asm - odgrywa gamę C-dur na kanale 1
; ------------------------------------------------------------------------
AUDF1   = $D200                 ; częstotliwość kanału 1
AUDC1   = $D201                 ; głośność i barwa kanału 1
AUDCTL  = $D208                 ; sterowanie dźwiękiem
SKCTL   = $D20F                 ; sterowanie portem szeregowym
RTCLOK  = $14                   ; zegar systemowy

        org $2000               ; adres asemblacji: $2000

start   lda #0                  ; inicjalizacja układu POKEY
        sta AUDCTL              ; zapisz A do AUDCTL - sterowanie dźwiękiem
        lda #3                  ; A = 3
        sta SKCTL               ; zapisz A do SKCTL - sterowanie portem szeregowym (zapis)

        ldx #0                  ; X = numer nuty
graj    lda nuty,x              ; A = wartość dzielnika dla kolejnej nuty
        beq koniec              ; zero w tablicy oznacza koniec melodii
        sta AUDF1               ; ustaw wysokość dźwięku
        lda #$A8                ; czysty ton ($A_), głośność 8
        sta AUDC1               ; włącz dźwięk
        ldy #15                 ; nuta trwa 15 ramek (0,3 sekundy)
        jsr pauza               ; wywołaj podprogram pauza
        lda #0                  ; wycisz kanał (głośność 0)
        sta AUDC1               ; zapisz A do AUDC1 - głośność i barwa kanału 1
        ldy #3                  ; krótka przerwa między nutami: 3 ramki
        jsr pauza               ; wywołaj podprogram pauza
        inx                     ; następna nuta
        bne graj                ; (skok zawsze - tablica ma mniej niż 256 pozycji)

koniec  lda #0                  ; na wszelki wypadek wycisz kanał
```

```
        sta AUDC1               ; zapisz A do AUDC1 - głośność i barwa kanału 1
        rts                     ; powrót z podprogramu

; --- pauza: czeka Y ramek. Niszczy A, Y. Zachowuje X. ------------------------
pauza   lda RTCLOK              ; A = bieżący stan zegara
pau1    cmp RTCLOK              ; czekaj na jego zmianę (jedna ramka)
        beq pau1                ; skocz do pau1, gdy Z = 1 (równe / zero)
        dey                     ; jedna ramka mniej
        bne pauza               ; skocz do pauza, gdy Z = 0 (różne / nie zero)
        rts                     ; powrót z podprogramu

nuty    dta 121,108,96,91,81,72,64,60  ; C D E F G A H C
        dta 0                   ; znacznik końca

        org $02E0               ; wektor RUNAD: adres uruchomienia programu
        dta a(start)            ; adres uruchomienia: start
```

25.3 Efekty dźwiękowe

Ciekawe dźwięki powstają, gdy parametry **zmieniają się w czasie**. Dwa klasyczne przykłady:

Plik: efekty.asm

```
; Efekty dźwiękowe. Obie procedury niszczą A, X, Y.
AUDF1   = $D200                 ; częstotliwość kanału 1
AUDC1   = $D201                 ; głośność i barwa kanału 1
RTCLOK  = $14                   ; zegar systemowy (3 bajty; $14 = najmłodszy)

; wybuch - szum o malejącej głośności (15 -> 0), każdy poziom trwa 2 ramki
wybuch  lda #40                 ; niska "częstotliwość" szumu: głuchy łoskot
        sta AUDF1               ; zapisz A do AUDF1 - częstotliwość kanału 1
        ldx #15                 ; X = głośność, od najwyższej
wyb_1   txa                     ; A = głośność (0..15)
        ora #$80                ; dołącz barwę: biały szum
        sta AUDC1               ; zapisz A do AUDC1 - głośność i barwa kanału 1
        ldy #2                  ; odczekaj 2 ramki
wyb_2   lda RTCLOK              ; A = RTCLOK - zegar systemowy (3 bajty; $14 = najmłodszy)
wyb_3   cmp RTCLOK              ; porównaj A z: RTCLOK - zegar systemowy (3 bajty; $14 = najmłodszy)
        beq wyb_3               ; skocz do wyb_3, gdy Z = 1 (równe / zero)
        dey                     ; Y = Y - 1
        bne wyb_2               ; skocz do wyb_2, gdy Z = 0 (różne / nie zero)
        dex                     ; ciszej
        bpl wyb_1               ; powtarzaj aż do głośności 0 włącznie
        rts                     ; powrót z podprogramu

; laser - czysty ton o szybko opadającej wysokości
laser   ldx #10                 ; X = dzielnik: zaczynamy od wysokiego dźwięku
        lda #$A8                ; czysty ton, głośność 8
        sta AUDC1               ; zapisz A do AUDC1 - głośność i barwa kanału 1
las_1   stx AUDF1               ; ustaw wysokość
        lda RTCLOK              ; odczekaj 1 ramkę
las_2   cmp RTCLOK              ; porównaj A z: RTCLOK - zegar systemowy (3 bajty; $14 = najmłodszy)
        beq las_2               ; skocz do las_2, gdy Z = 1 (równe / zero)
        inx                     ; większy dzielnik = niższy dźwięk
        inx                     ; X = X + 1
        inx                     ; X = X + 1
        inx                     ; X = X + 1
        cpx #90                 ; opadaj aż do tej wartości
        bcc las_1               ; skocz do las_1, gdy C = 0
        lda #0                  ; wycisz
        sta AUDC1               ; zapisz A do AUDC1 - głośność i barwa kanału 1
        rts                     ; powrót z podprogramu
```

Te procedury blokują program na czas trwania efektu. W grze dźwięk obsługuje się w przerwaniu VBI: procedura co ramkę ustawia kolejną wartość głośności lub częstotliwości z tablicy, a program główny tylko „zamawia" efekt, wpisując jego numer do zmiennej. Tak samo działają odtwarzacze muzyki.

25.4 Liczby losowe

Rejestr **RANDOM** (`$D20A`) układu POKEY przy każdym odczycie zwraca inny, praktycznie losowy bajt. Układ zmienia jego zawartość z częstotliwością zegara procesora, więc kolejne odczyty nie są ze sobą związane w widoczny sposób.

```
        lda $D20A               ; A = liczba losowa 0..255
```

25.4.1 Zakres będący potęgą dwójki

Najprostszy przypadek — wystarczy maska:

```
        lda $D20A               ; liczba losowa 0..255
        and #%00000111          ; zostaw 3 bity: liczba losowa 0..7
```

25.4.2 Dowolny zakres — metoda odrzucania

Aby wylosować liczbę z zakresu 0...N−1, gdy N nie jest potęgą dwójki, maskujemy do najbliższej większej potęgi i odrzucamy wartości za duże:

Plik: losuj40.asm

```
; losuj40 - liczba losowa z zakresu 0..39 (np. kolumna ekranu)
; Wyjście: A = liczba losowa.
RANDOM  = $D20A                 ; liczba losowa (odczyt)

losuj40 lda RANDOM              ; liczba losowa 0..255
        and #%00111111          ; zostaw 6 bitów: 0..63
        cmp #40                 ; czy mieści się w zakresie?
        bcs losuj40             ; nie (40..63): losuj jeszcze raz
        rts                     ; tak: A = 0..39
```

Każda wartość jest dokładnie tak samo prawdopodobna. Średnio potrzeba 1,6 losowania.

25.4.3 Dowolny zakres — metoda mnożenia

Gdy nie wolno dopuścić do zmiennego czasu wykonania, używamy mnożenia: iloczyn liczby losowej 0...255 i wielkości zakresu N, podzielony przez 256, daje liczbę 0...N−1. A dzielenie przez 256 to po prostu **starszy bajt** iloczynu:

Plik: losuj_n.asm

```
; losuj_n - liczba losowa z zakresu 0..N-1
; Wejście:  A = N (1..255).  Wyjście: A = liczba losowa 0..N-1.
; Niszczy:  X oraz komórki ln_a i ln_b.
RANDOM  = $D20A                 ; liczba losowa (odczyt)
ln_a    = $80                   ; mnożna: wielkość zakresu N
ln_b    = $81                   ; mnożnik: liczba losowa; potem młodszy bajt iloczynu

losuj_n sta ln_a                ; mnożna := N
        lda RANDOM              ; A = RANDOM - liczba losowa (odczyt)
        sta ln_b                ; mnożnik := liczba losowa 0..255
        lda #0                  ; --- mnożenie 8 x 8 (procedura mul8f z rozdziału 14) ---
        ldx #8                  ; X = 8
        lsr ln_b                ; przesuń ln_b o 1 bit w prawo (bit 0 -> C)
ln_1    bcc ln_2                ; skocz do ln_2, gdy C = 0
        clc                     ; C = 0 (przed dodawaniem)
        adc ln_a                ; A = A + ln_a + C
ln_2    ror @                   ; obróć A w prawo przez C
        ror ln_b                ; obróć ln_b w prawo przez C
        dex                     ; X = X - 1
        bne ln_1                ; skocz do ln_1, gdy Z = 0 (różne / nie zero)
        rts                     ; A = starszy bajt iloczynu = (N * losowa) / 256
```

25.5 Dźwięk klawiatury i głośnik wewnętrzny

Dla porządku: krótkie „kliknięcie” przy naciskaniu klawiszy generuje nie POKEY, lecz układ GTIA — przez bit 3 rejestru CONSOL (`$D01F`). W modelach XL/XE dźwięk ten trafia do telewizora razem z resztą. Zmienna NOCLIK (`$02DB`) o wartości różnej od zera wyłącza kliknięcia klawiatury.

Ćwiczenia

1. Zmień program `gama.asm` tak, by grał gamę w górę, a potem w dół.
2. Dodaj drugi głos: na kanale 2 graj ten sam dźwięk o oktawę niżej (dzielnik w przybliżeniu dwa razy większy).
3. Napisz procedurę symulującą rzut kostką: liczba losowa 1…6.

ROZDZIAŁ 26

Duszki: grafika graczy i pocisków (PMG)

Atari ma sprzętowe „duszki” — obiekty graficzne nakładane na obraz niezależnie od jego zawartości. Oficjalnie nazywają się **graczami** i **pociskami** (*Player/Missile Graphics*, PMG). Czterech graczy po 8 punktów szerokości i cztery pociski po 2 punkty: to niewiele, ale mają ogromne zalety — własne kolory, natychmiastowy ruch w poziomie i sprzętowe wykrywanie kolizji.

26.1 Jak działa duszek

Gracz to **pionowy pasek** o szerokości 8 punktów, sięgający przez całą wysokość ekranu. Jego kształt opisuje ciąg bajtów w pamięci: jeden bajt na jedną linię obrazu (lub na dwie linie, w niższej rozdzielczości). Ustawiony bit to zapalony punkt.

```
pamięć gracza                ekran
...
bajt 60:  %00000000
bajt 61:  %00011000            ##
bajt 62:  %00111100           ####
bajt 63:  %01111110          ######
bajt 64:  %11111111         ########
bajt 65:  %00100100           #  #
bajt 66:  %00000000
...
└── numer bajtu = pozycja pionowa       pozycja pozioma: rejestr HPOSP
```

Wynikają z tego dwie zupełnie różne metody ruchu:

- **w poziomie** — wystarczy wpisać nową wartość do rejestru pozycji. Jeden rozkaz, natychmiastowy efekt.
- **w pionie** — trzeba **przesunąć bajty kształtu w pamięci**: zmazać je w starym miejscu i wpisać w nowym.

26.2 Pamięć duszków

Dane wszystkich duszków leżą w jednym bloku pamięci, którego początek (starszy bajt adresu) podaje rejestr **PMBASE** (`$D407`). Układ bloku zależy od rozdzielczości pionowej:

	Rozdzielczość dwuliniowa	Rozdzielczość jednoliniowa
Jeden bajt kształtu to	2 linie obrazu	1 linia obrazu
Rozmiar bloku	1 KB	2 KB
Adres bloku podzielny przez	`$400` (1024)	`$800` (2048)
Pociski (wszystkie cztery)	PMBASE + `$180`	PMBASE + `$300`
Gracz 0	PMBASE + `$200`	PMBASE + `$400`
Gracz 1	PMBASE + `$280`	PMBASE + `$500`
Gracz 2	PMBASE + `$300`	PMBASE + `$600`
Gracz 3	PMBASE + `$380`	PMBASE + `$700`
Bajtów na gracza	128	256
Widoczny zakres pozycji pionowych	około 16–111	około 32–223

26.3 Rejestry

Rejestr	Adres	Cień	Znaczenie
HPOSP0–3	`$D000–$D003`	—	pozycja pozioma graczy (zapis)
HPOSM0–3	`$D004–$D007`	—	pozycja pozioma pocisków (zapis)
SIZEP0–3	`$D008–$D00B`	—	szerokość graczy: 0 = zwykła, 1 = podwójna, 3 = poczwórna
COLPM0–3	`$D012–$D015`	PCOLR0–3 `$02C0`	kolory graczy (pocisk ma kolor swojego gracza)
PRIOR	`$D01B`	GPRIOR `$026F`	priorytety: co zasłania co
GRACTL	`$D01D`	—	3 = wyświetlaj graczy i pociski
PMBASE	`$D407`	—	starszy bajt adresu pamięci duszków
DMACTL	`$D400`	SDMCTL `$022F`	włączenie pobierania danych duszków

Pozycja pozioma: lewa krawędź standardowego ekranu to wartość 48, prawa — 208. Gracz o zwykłej szerokości jest w całości widoczny dla wartości 48–200. Wartość 0 chowa go poza ekran.

SDMCTL — do standardowej wartości `$22` (obraz normalnej szerokości) dodajemy bity:

Bit	Wartość	Znaczenie
2	`$04`	pobieraj dane pocisków
3	`$08`	pobieraj dane graczy
4	`$10`	rozdzielczość jednoliniowa (bez tego bitu: dwuliniowa)

Stąd dwie typowe wartości: **`$2E`** — duszki w rozdzielczości dwuliniowej, **`$3E`** — w jednoliniowej.

GPRIOR — najczęściej używane wartości: 1 = gracze przed polem gry (zasłaniają grafikę), 4 = pole gry przed graczami (gracze chowają się „za" grafiką). Bit 4 (`$10`) nadaje wszystkim pociskom wspólny kolor z rejestru COLOR3, co pozwala złożyć z nich piątego gracza.

26.4 Kolizje

GTIA rejestruje każde nałożenie się zapalonego punktu duszka na inny obiekt. Rejestry kolizji odczytuje się spod tych samych adresów, pod które zapisuje się pozycje:

Rejestr (odczyt)	Adres	Zawartość
M0PF–M3PF	`$D000–$D003`	kolizje pocisków z polem gry (bity 0–3: kolory pola 0–3)
P0PF–P3PF	`$D004–$D007`	kolizje graczy z polem gry
M0PL–M3PL	`$D008–$D00B`	kolizje pocisków z graczami (bity 0–3: gracze 0–3)
P0PL–P3PL	`$D00C–$D00F`	kolizje graczy między sobą

Bity kolizji **same się nie kasują**. Zapis dowolnej wartości do rejestru HITCLR (`$D01E`) zeruje wszystkie naraz. Typowy schemat w grze: raz na ramkę odczytaj kolizje, zareaguj, skasuj.

```
        lda $D00C               ; P0PL: kolizje gracza 0 z innymi graczami
        and #%00000010          ; bit 1: zderzenie z graczem 1?
        beq brak_kol            ; nie
        jsr trafiony            ; tak
brak_kol sta $D01E              ; HITCLR: skasuj wszystkie kolizje (wartość nie ma znaczenia)
```

26.5 Program: duszek sterowany joystickiem

Plik: duszek.asm

```
; ------------------------------------------------------------------------
; duszek.asm - gracz 0 poruszany joystickiem (rozdzielczość dwuliniowa)
; ------------------------------------------------------------------------
SDMCTL  = $022F                 ; cień DMACTL
GPRIOR  = $026F                 ; cień PRIOR
PCOLR0  = $02C0                 ; cień koloru gracza 0
STICK0  = $0278                 ; joystick
STRIG0  = $0284                 ; przycisk joysticka
RTCLOK  = $14                   ; zegar systemowy
ATRACT  = $4D                   ; licznik wygaszacza ekranu
HPOSP0  = $D000                 ; pozycja pozioma gracza 0
GRACTL  = $D01D                 ; sterowanie grafiką PMG
PMBASE  = $D407                 ; adres pamięci duszków

PMG     = $3000                 ; blok pamięci duszków: 1 KB od adresu podzielnego przez $400
GRACZ0  = PMG+$200              ; dane gracza 0 w rozdzielczości dwuliniowej (128 bajtów)
WYS     = 8                     ; wysokość kształtu w bajtach

wsk     = $80                   ; wskaźnik pomocniczy (2 bajty)
px      = $82                   ; pozycja pozioma duszka
py      = $83                   ; pozycja pionowa duszka (numer pierwszego bajtu kształtu)

        org $2000               ; adres asemblacji: $2000

start   lda #<PMG               ; --- wyczyść cały blok pamięci duszków (4 strony) ---
        sta wsk                 ; zapisz A do wsk
        lda #>PMG               ; A = starszy bajt wartości PMG
        sta wsk+1               ; zapisz A pod adres wsk+1
        lda #0                  ; A = 0: żaden punkt niezapalony
        tay                     ; Y = A
        ldx #4                  ; 4 strony = 1024 bajty
czysc   sta (wsk),y             ; zapisz A pod adres (wsk) + Y
        iny                     ; Y = Y + 1
        bne czysc               ; skocz do czysc, gdy Z = 0 (różne / nie zero)
        inc wsk+1               ; zwiększ o 1 bajt spod adresu wsk+1
        dex                     ; X = X - 1
        bne czysc               ; skocz do czysc, gdy Z = 0 (różne / nie zero)

        lda #>PMG               ; --- włącz grafikę PMG ---
        sta PMBASE              ; adres bloku pamięci duszków
        lda #$2E                ; obraz normalny + gracze + pociski, rozdzielczość dwuliniowa
        sta SDMCTL              ; zapisz A do SDMCTL - cień DMACTL
        lda #3                  ; wyświetlaj graczy i pociski
        sta GRACTL              ; zapisz A do GRACTL - sterowanie grafiką PMG
        lda #$1E                ; kolor gracza 0: jasnożółty
        sta PCOLR0              ; zapisz A do PCOLR0 - kolory graczy 0-3 (4 bajty)
        lda #1                  ; gracze przed polem gry
        sta GPRIOR              ; zapisz A do GPRIOR - cień PRIOR

        lda #124                ; pozycja początkowa: środek ekranu
        sta px                  ; zapisz A do px
        lda #60                 ; A = 60
        sta py                  ; zapisz A do py

petla   jsr rysuj               ; wpisz kształt do pamięci gracza na wysokości py
        lda px                  ; ustaw pozycję poziomą
        sta HPOSP0              ; zapisz A do HPOSP0 - pozycja pozioma gracza 0 (zapis)

        lda RTCLOK              ; poczekaj na następną ramkę
cz1     cmp RTCLOK              ; porównaj A z: RTCLOK - zegar systemowy (3 bajty; $14 = najmłodszy)
        beq cz1                 ; skocz do cz1, gdy Z = 1 (równe / zero)

        lda STRIG0              ; przycisk wciśnięty?
        beq koniec              ; tak: zakończ

        lda STICK0              ; joystick wychylony?
        cmp #15                 ; porównaj A z: 15
        beq petla               ; nie: nic nie zmieniaj
```

```
        lda #0                  ; tak: odsuń wygaszacz ekranu
        sta ATRACT              ; zapisz A do ATRACT - licznik wygaszacza ekranu
        jsr zmaz                ; usuń kształt ze starego miejsca (potrzebne przy ruchu w pionie)

        lda STICK0              ; A = stan joysticka
        lsr @                   ; bit 0: góra
        bcs r1                  ; skocz do r1, gdy C = 1
        ldx py                  ; X = py
        cpx #17                 ; górna granica widocznego obszaru
        bcc r1                  ; py < 17: nie wyżej
        dec py                  ; py = py - 1
r1      lsr @                   ; bit 1: dół
        bcs r2                  ; skocz do r2, gdy C = 1
        ldx py                  ; X = py
        cpx #104                ; dolna granica (111 minus wysokość kształtu)
        bcs r2                  ; py >= 104: nie niżej
        inc py                  ; py = py + 1
r2      lsr @                   ; bit 2: lewo
        bcs r3                  ; skocz do r3, gdy C = 1
        ldx px                  ; X = px
        cpx #49                 ; lewa krawędź ekranu
        bcc r3                  ; skocz do r3, gdy C = 0
        dec px                  ; px = px - 1
r3      lsr @                   ; bit 3: prawo
        bcs r4                  ; skocz do r4, gdy C = 1
        ldx px                  ; X = px
        cpx #200                ; prawa krawędź ekranu minus szerokość gracza
        bcs r4                  ; skocz do r4, gdy C = 1
        inc px                  ; px = px + 1
r4      jmp petla               ; skocz do petla

koniec  lda #0                  ; --- wyłącz grafikę PMG ---
        sta GRACTL              ; zapisz A do GRACTL - sterowanie grafiką PMG
        sta HPOSP0              ; schowaj gracza poza ekran
        lda #$22                ; standardowa wartość SDMCTL
        sta SDMCTL              ; zapisz A do SDMCTL - cień DMACTL
        rts                     ; powrót z podprogramu

; --- rysuj: kopiuje kształt do pamięci gracza 0 od pozycji py ---------------
rysuj   ldx py                  ; X = indeks w pamięci gracza
        ldy #0                  ; Y = indeks w tablicy kształtu
rys1    lda ksztalt,y           ; bajt kształtu
        sta GRACZ0,x            ; do pamięci gracza
        inx                     ; X = X + 1
        iny                     ; Y = Y + 1
        cpy #WYS                ; cały kształt?
        bne rys1                ; skocz do rys1, gdy Z = 0 (różne / nie zero)
        rts                     ; powrót z podprogramu

; --- zmaz: zeruje WYS bajtów pamięci gracza 0 od pozycji py -----------------
zmaz    ldx py                  ; X = py
        ldy #WYS                ; Y = licznik bajtów
        lda #0                  ; A = 0
zm1     sta GRACZ0,x            ; zapisz A pod adres GRACZ0+X
        inx                     ; X = X + 1
        dey                     ; Y = Y - 1
        bne zm1                 ; skocz do zm1, gdy Z = 0 (różne / nie zero)
        rts                     ; powrót z podprogramu

ksztalt dta %00111100           ; kształt duszka: 8 bajtów = 16 linii obrazu
        dta %01111110           ; dane: wzór bitów
        dta %11011011           ; dane: wzór bitów
        dta %11111111           ; dane: wzór bitów
        dta %10111101           ; dane: wzór bitów
        dta %11000011           ; dane: wzór bitów
        dta %01111110           ; dane: wzór bitów
        dta %00111100           ; dane: wzór bitów

        org $02E0               ; wektor RUNAD: adres uruchomienia programu
        dta a(start)            ; adres uruchomienia: start
```

Kilka uwag do programu:

- Adres `PMG = $3000` leży poza samym programem (który zajmuje okolice `$2000`). Blok pamięci duszków nie jest częścią pliku — program sam go czyści przy starcie.
- Ruch w poziomie o jedną jednostkę to przesunięcie o jeden punkt trybu o 160 punktach w linii; ruch w pionie o 1 to dwie linie obrazu.
- Gdybyśmy zapisywali pozycję do HPOSP0 w dowolnym momencie, duszek mógłby się „rozerwać" — górna część w starym miejscu, dolna w nowym. Dlatego zmiany wprowadzamy raz na ramkę; w poważniejszej grze robi się to w przerwaniu VBI.

26.6 Co dalej z duszkami

- **Więcej niż cztery obiekty**: ten sam gracz może pojawić się kilka razy na różnych wysokościach ekranu, jeśli przerwanie DLI zmieni mu pozycję poziomą i kolor między jednym a drugim wystąpieniem.
- **Duszki wielokolorowe**: dwaj gracze nałożeni na siebie; po ustawieniu bitu 5 (`$20`) w GPRIOR obszar wspólny otrzymuje trzeci kolor.
- **Animacja**: kilka kształtów w tablicy, wpisywanych kolejno w to samo miejsce.
- **Duszki programowe**: gdy sprzętowych jest za mało, obiekty rysuje się zwykłymi operacjami na pamięci obrazu — operacjami `and` (wycięcie tła maską) i `ora` (wstawienie kształtu), poznanymi w rozdziale 13.

ROZDZIAŁ 27

Stacja dysków od podszewki: SIO i sektory

W rozdziale 21 czytaliśmy pliki przez CIO — o to, gdzie na dyskietce leżą dane, martwił się DOS. Teraz schodzimy piętro niżej: do pojedynczych **sektorów**. Na tym poziomie pisze się programy kopiujące dyskietki, edytory sektorów, własne systemy plików, programy startowe (*boot loadery*) i gry, które ładują dane bez udziału DOS-u.

27.1 Dyskietka i sektory

Dyskietka jest podzielona na ponumerowane **sektory** — bloki danych o stałym rozmiarze. Numeracja zaczyna się **od 1**.

Format	Sektorów	Rozmiar sektora	Pojemność	Stacje
pojedyncza gęstość (SD)	720	128 bajtów	90 KB	810, 1050, XF551
rozszerzona gęstość (ED)	1040	128 bajtów	130 KB	1050, XF551
podwójna gęstość (DD)	720	256 bajtów	180 KB	XF551 i stacje innych firm
podwójna gęstość, dwie strony	1440	256 bajtów	360 KB	XF551

W formatach o sektorach 256-bajtowych **pierwsze trzy sektory mają zawsze 128 bajtów** — żeby system operacyjny mógł z nich wystartować bez wiedzy o gęstości dyskietki.

27.2 Warstwy oprogramowania

```
program użytkowy
     |  "otwórz plik D:GRA.DAT, czytaj 100 bajtów"
     ▼
CIO  ──►  DOS (procedura obsługi urządzenia D:)        pliki, katalog, mapa zajętości
     |  "czytaj sektor 372 ze stacji 1"
     ▼
SIO  ($E459)                                           ramki danych, sumy kontrolne
     |  bity na linii szeregowej, 19200 bitów/s
     ▼
stacja dysków (ma własny procesor i sama obsługuje mechanikę)
```

SIO (*Serial Input/Output*) to procedura systemu operacyjnego, która wysyła rozkaz do urządzenia podłączonego do szyny szeregowej i odbiera lub wysyła blok danych. Stacja dysków Atari jest urządzeniem „inteligentnym”: dostaje polecenie *podaj sektor 372* i odsyła 128 bajtów.

Ważna dla posiadaczy nowoczesnych urządzeń informacja: interfejsy typu SIDE czy SIO2SD oraz emulatory **przechwytują właśnie wywołania SIO**. Program korzystający z wektora `$E459` będzie więc działał z każdym urządzeniem udającym stację dysków — także z przyspieszoną transmisją.

27.3 Blok DCB

Parametry operacji podaje się w bloku **DCB** (*Device Control Block*) pod adresem `$0300`:

Pole	Adres	Znaczenie
DDEVIC	$0300	identyfikator urządzenia: $31 dla stacji dysków
DUNIT	$0301	numer urządzenia: 1–8 (D1: ... D8:)
DCOMND	$0302	kod rozkazu
DSTATS	$0303	przed wywołaniem: kierunek danych; po wywołaniu: status
DBUFLO, DBUFHI	$0304, $0305	adres bufora danych
DTIMLO	$0306	limit czasu oczekiwania (w przybliżeniu w sekundach)
DBYTLO, DBYTHI	$0308, $0309	liczba bajtów danych
DAUX1, DAUX2	$030A, $030B	parametr rozkazu: zwykle numer sektora (młodszy, starszy bajt)

Kierunek danych w DSTATS:

Wartość	Znaczenie
$40	urządzenie przysyła dane do komputera (odczyt)
$80	komputer wysyła dane do urządzenia (zapis)
$00	rozkaz bez danych

Rozkazy stacji dysków:

Kod	Znak	Rozkaz	Dane
$52	R	odczyt sektora	odbiór: 128 lub 256 bajtów
$50	P	zapis sektora bez weryfikacji	wysłanie: 128 lub 256 bajtów
$57	W	zapis sektora z weryfikacją	wysłanie: 128 lub 256 bajtów
$53	S	odczyt statusu stacji	odbiór: 4 bajty
$21	!	formatowanie w pojedynczej gęstości	odbiór: 128 bajtów (lista złych sektorów)
$22	”	formatowanie w rozszerzonej gęstości	odbiór: 128 bajtów
$4E	N	odczyt konfiguracji (blok PERCOM)	odbiór: 12 bajtów
$4F	O	zapis konfiguracji (blok PERCOM)	wysłanie: 12 bajtów

Po powrocie z SIO status znajduje się w rejestrze Y i w polu DSTATS; znacznik N jest ustawiony, gdy wystąpił błąd:

Status	Znaczenie
1	operacja udana
138 ($8A)	urządzenie nie odpowiada — brak stacji o tym numerze
139 ($8B)	urządzenie odrzuciło rozkaz (NAK) — np. nieznany rozkaz, zły numer sektora
143 ($8F)	błąd sumy kontrolnej transmisji
144 ($90)	urządzenie nie wykonało rozkazu — uszkodzony sektor, dyskietka zabezpieczona przed zapisem

27.4 Odczyt i zapis sektora

Plik: sektor_io.asm

```
; ------------------------------------------------------------------------
; sektor_io.asm - odczyt i zapis sektora dyskietki przez SIO
; Wejście:  sio_stacja = numer stacji (1..8)
;           sio_sektor = numer sektora (2 bajty, liczony od 1)
;           sio_bufor  = adres bufora (2 bajty)
;           sio_rozmiar = rozmiar sektora w bajtach (2 bajty: 128 lub 256)
; Wyjście:  Y = status; N = 1 oznacza błąd
; ------------------------------------------------------------------------
DDEVIC  = $0300                 ; identyfikator urządzenia
DUNIT   = $0301                 ; numer urządzenia
DCOMND  = $0302                 ; rozkaz
DSTATS  = $0303                 ; kierunek transmisji / status
DBUFLO  = $0304                 ; adres bufora: młodszy bajt
DBUFHI  = $0305                 ; adres bufora: starszy bajt
DTIMLO  = $0306                 ; limit czasu
DBYTLO  = $0308                 ; liczba bajtów: młodszy bajt
DBYTHI  = $0309                 ; liczba bajtów: starszy bajt
DAUX1   = $030A                 ; parametr 1 (np. młodszy bajt numeru sektora)
DAUX2   = $030B                 ; parametr 2 (np. starszy bajt numeru sektora)
SIOV    = $E459                 ; wejście do procedury SIO

sio_stacja = $80                ; numer stacji
sio_sektor = $81                ; numer sektora (2 bajty)
sio_bufor = $83                 ; adres bufora (2 bajty)
sio_rozmiar = $85               ; rozmiar sektora (2 bajty)

czytaj_sektor lda #'R'          ; rozkaz: odczyt sektora
        ldx #$40                ; kierunek: dane ze stacji do komputera
        bne sio_wspolne         ; (skok zawsze, bo X <> 0)

zapisz_sektor lda #'W'          ; rozkaz: zapis sektora z weryfikacją
        ldx #$80                ; kierunek: dane z komputera do stacji

sio_wspolne sta DCOMND          ; kod rozkazu
        stx DSTATS              ; kierunek transmisji danych
        lda #$31                ; identyfikator urządzenia: stacja dysków
        sta DDEVIC              ; zapisz A do DDEVIC - identyfikator urządzenia
        lda sio_stacja          ; numer stacji
        sta DUNIT               ; zapisz A do DUNIT - numer urządzenia
        lda sio_bufor           ; adres bufora
        sta DBUFLO              ; zapisz A do DBUFLO - adres bufora: młodszy bajt
        lda sio_bufor+1         ; A = bajt spod adresu sio_bufor+1
        sta DBUFHI              ; zapisz A do DBUFHI - adres bufora: starszy bajt
        lda #7                  ; limit czasu: około 7 sekund
        sta DTIMLO              ; zapisz A do DTIMLO - limit czasu
        lda sio_rozmiar         ; liczba bajtów do przesłania
        sta DBYTLO              ; zapisz A do DBYTLO - liczba bajtów: młodszy bajt
        lda sio_rozmiar+1       ; A = bajt spod adresu sio_rozmiar+1
        sta DBYTHI              ; zapisz A do DBYTHI - liczba bajtów: starszy bajt
        lda sio_sektor          ; numer sektora: młodszy bajt
        sta DAUX1               ; zapisz A do DAUX1 - parametr 1 (np. młodszy bajt numeru sektora)
        lda sio_sektor+1        ; numer sektora: starszy bajt
        sta DAUX2               ; zapisz A do DAUX2 - parametr 2 (np. starszy bajt numeru sektora)
        jmp SIOV                ; wykonaj operację; SIO wróci do naszego wywołującego
```

SIO samo ponawia nieudaną transmisję kilkanaście razy, zanim zgłosi błąd, więc program nie musi organizować własnych powtórzeń.

> **Wskazówka**
>
> System ma też wygodniejszą nakładkę: **DSKINV** (`$E453`). Wystarczy ustawić DUNIT, DCOMND, adres bufora i numer sektora — identyfikator urządzenia, kierunek, limit czasu i liczbę bajtów procedura uzupełni sama (rozmiar sektora bierze ze zmiennej DSCTLN, `$02D5`, domyślnie 128).

27.5 Program: podgląd sektorów

Program czyta kolejne sektory dyskietki w stacji D1: i wyświetla ich zawartość szesnastkowo. Dowolny klawisz przechodzi do następnego sektora, ESC kończy pracę.

Plik: sektor.asm

```
; ------------------------------------------------------------------------
; sektor.asm - podgląd zawartości sektorów dyskietki (D1:, sektory 128-bajtowe)
; ------------------------------------------------------------------------
DDEVIC  = $0300                 ; blok DCB
DUNIT   = $0301                 ; numer urządzenia
DCOMND  = $0302                 ; rozkaz
DSTATS  = $0303                 ; kierunek transmisji / status
DBUFLO  = $0304                 ; adres bufora: młodszy bajt
DBUFHI  = $0305                 ; adres bufora: starszy bajt
DTIMLO  = $0306                 ; limit czasu
DBYTLO  = $0308                 ; liczba bajtów: młodszy bajt
DBYTHI  = $0309                 ; liczba bajtów: starszy bajt
DAUX1   = $030A                 ; parametr 1 (np. młodszy bajt numeru sektora)
DAUX2   = $030B                 ; parametr 2 (np. starszy bajt numeru sektora)
SIOV    = $E459                 ; transmisja szeregowa
ICCOM   = $0342                 ; blok IOCB 0
ICBAL   = $0344                 ; adres bufora: młodszy bajt
ICBAH   = $0345                 ; adres bufora: starszy bajt
ICBLL   = $0348                 ; długość bufora: młodszy bajt
ICBLH   = $0349                 ; długość bufora: starszy bajt
CIOV    = $E456                 ; centralna procedura wejścia/wyjścia
KEYBDV  = $E420                 ; tablica wektorów procedury obsługi klawiatury
EOL     = $9B                   ; znak końca linii

sektor  = $80                   ; numer bieżącego sektora (2 bajty)
poz     = $82                   ; pozycja w buforze sektora (0..127)
tmp     = $83                   ; zmienna pomocnicza

        org $2000               ; adres asemblacji: $2000

start   lda #1                  ; zaczynamy od sektora 1
        sta sektor              ; zapisz A do sektor
        lda #0                  ; A = 0
        sta sektor+1            ; zapisz A pod adres sektor+1

pokaz   jsr czytaj              ; wczytaj bieżący sektor do bufora
        bpl dobrze              ; N = 0: bez błędu
        jmp blad                ; skocz do blad

dobrze  lda #<t_sektor          ; wypisz nagłówek: "Sektor $"
        ldy #>t_sektor          ; Y = starszy bajt adresu t_sektor
        jsr print               ; wywołaj podprogram print
        lda sektor+1            ; numer sektora szesnastkowo: starszy bajt...
        jsr hex                 ; wywołaj podprogram hex
        lda sektor              ; ...i młodszy
        jsr hex                 ; wywołaj podprogram hex
        lda #EOL                ; A = EOL
        jsr putchar             ; wywołaj podprogram putchar

        lda #0                  ; --- wypisz 128 bajtów: 16 wierszy po 8 bajtów ---
        sta poz                 ; zapisz A do poz
wiersz  lda poz                 ; na początku wiersza: pozycja pierwszego bajtu
        jsr hex                 ; wywołaj podprogram hex
        lda #':'                ; A = kod ATASCII znaku ':'
        jsr putchar             ; wywołaj podprogram putchar
bajt    lda #' '                ; odstęp przed każdym bajtem
        jsr putchar             ; wywołaj podprogram putchar
        ldx poz                 ; pobierz bajt z bufora
        lda bufor,x             ; A = bajt spod adresu bufor+X
        jsr hex                 ; wypisz go szesnastkowo
        inc poz                 ; następna pozycja
        lda poz                 ; A = poz
        and #%00000111          ; czy to wielokrotność 8 ?
        bne bajt                ; nie: kontynuuj wiersz
        lda #EOL                ; tak: koniec wiersza
```

```
        jsr putchar             ; wywołaj podprogram putchar
        lda poz                 ; A = poz
        cmp #128                ; cały sektor?
        bne wiersz              ; skocz do wiersz, gdy Z = 0 (różne / nie zero)

        jsr getkey              ; czekaj na klawisz
        cmp #27                 ; ESC ?
        beq koniec              ; tak: wyjście
        inc sektor              ; nie: następny sektor (zwiększenie 16-bitowe)
        bne pokaz               ; skocz do pokaz, gdy Z = 0 (różne / nie zero)
        inc sektor+1            ; zwiększ o 1 bajt spod adresu sektor+1
        jmp pokaz               ; skocz do pokaz

blad    sty tmp                 ; zapamiętaj numer błędu
        lda #<t_blad            ; wypisz "Blad SIO $"
        ldy #>t_blad            ; Y = starszy bajt adresu t_blad
        jsr print               ; wywołaj podprogram print
        lda tmp                 ; i numer błędu szesnastkowo
        jsr hex                 ; wywołaj podprogram hex
        lda #EOL                ; A = EOL
        jsr putchar             ; wywołaj podprogram putchar
        jsr getkey              ; poczekaj na klawisz przed powrotem do DOS-u
koniec  rts                     ; powrót z podprogramu

; --- czytaj: wczytuje sektor "sektor" ze stacji D1: do bufora --------------
czytaj  lda #$31                ; urządzenie: stacja dysków
        sta DDEVIC              ; zapisz A do DDEVIC - identyfikator urządzenia
        lda #1                  ; stacja numer 1
        sta DUNIT               ; zapisz A do DUNIT - numer urządzenia
        lda #'R'                ; rozkaz: odczyt sektora
        sta DCOMND              ; zapisz A do DCOMND - rozkaz
        lda #$40                ; kierunek: odbiór danych
        sta DSTATS              ; zapisz A do DSTATS - kierunek transmisji / status
        lda #<bufor             ; adres bufora
        sta DBUFLO              ; zapisz A do DBUFLO - adres bufora: młodszy bajt
        lda #>bufor             ; A = starszy bajt adresu bufor
        sta DBUFHI              ; zapisz A do DBUFHI - adres bufora: starszy bajt
        lda #7                  ; limit czasu
        sta DTIMLO              ; zapisz A do DTIMLO - limit czasu
        lda #<128               ; liczba bajtów: 128
        sta DBYTLO              ; zapisz A do DBYTLO - liczba bajtów: młodszy bajt
        lda #>128               ; A = starszy bajt wartości 128
        sta DBYTHI              ; zapisz A do DBYTHI - liczba bajtów: starszy bajt
        lda sektor              ; numer sektora
        sta DAUX1               ; zapisz A do DAUX1 - parametr 1 (np. młodszy bajt numeru sektora)
        lda sektor+1            ; A = bajt spod adresu sektor+1
        sta DAUX2               ; zapisz A do DAUX2 - parametr 2 (np. starszy bajt numeru sektora)
        jmp SIOV                ; Y = status, N = 1 przy błędzie

; --- hex: wypisuje A jako dwie cyfry szesnastkowe --------------------------
hex     pha                     ; zachowaj bajt
        lsr @                   ; starszy półbajt na miejsce młodszego
        lsr @                   ; przesuń A o 1 bit w prawo (bit 0 -> C)
        lsr @                   ; przesuń A o 1 bit w prawo (bit 0 -> C)
        lsr @                   ; przesuń A o 1 bit w prawo (bit 0 -> C)
        tax                     ; X = A
        lda cyfry,x             ; znak cyfry szesnastkowej
        jsr putchar             ; wywołaj podprogram putchar
        pla                     ; odtwórz bajt
        and #$0F                ; młodszy półbajt
        tax                     ; X = A
        lda cyfry,x             ; A = bajt spod adresu cyfry+X
        jmp putchar             ; wypisz i wróć

; --- putchar: wypisuje znak z A --------------------------------------------
putchar ldx #$00                ; X = $00
        pha                     ; odłóż A na stos
        lda #11                 ; A = 11
        sta ICCOM,x             ; zapisz A pod adres ICCOM+X - rozkaz
        lda #0                  ; A = 0
        sta ICBLL,x             ; zapisz A pod adres ICBLL+X - długość bufora: młodszy bajt
        sta ICBLH,x             ; zapisz A pod adres ICBLH+X - długość bufora: starszy bajt
```

```
        pla                     ; zdejmij A ze stosu
        jmp CIOV                ; skocz do CIOV - centralna procedura wejścia/wyjścia

; --- print: wypisuje DL_TXT znaków spod adresu A/Y (bez przejścia do nowej linii) ---
print   ldx #$00                ; X = $00
        sta ICBAL,x             ; zapisz A pod adres ICBAL+X - adres bufora: młodszy bajt
        tya                     ; A = Y
        sta ICBAH,x             ; zapisz A pod adres ICBAH+X - adres bufora: starszy bajt
        lda #11                 ; PUT CHARACTERS: dokładnie tyle znaków, ile podano
        sta ICCOM,x             ; zapisz A pod adres ICCOM+X - rozkaz
        lda #DL_TXT             ; oba teksty mają tę samą długość
        sta ICBLL,x             ; zapisz A pod adres ICBLL+X - długość bufora: młodszy bajt
        lda #0                  ; A = 0
        sta ICBLH,x             ; zapisz A pod adres ICBLH+X - długość bufora: starszy bajt
        jmp CIOV                ; skocz do CIOV - centralna procedura wejścia/wyjścia

; --- getkey: czeka na klawisz, zwraca kod ATASCII w A ----------------------
getkey  lda KEYBDV+5            ; A = bajt spod adresu KEYBDV+5
        pha                     ; odłóż A na stos
        lda KEYBDV+4            ; A = bajt spod adresu KEYBDV+4
        pha                     ; odłóż A na stos
        rts                     ; powrót z podprogramu

cyfry   dta c'0123456789ABCDEF'  ; tekst w kodach ATASCII
t_sektor dta c'Sektor   $'      ; oba teksty mają po 10 znaków
DL_TXT  = *-t_sektor            ; długość tekstu (obliczana przez asembler)
t_blad  dta c'Blad SIO $'       ; tekst w kodach ATASCII
bufor   .ds 128                 ; bufor na dane sektora

        org $02E0               ; wektor RUNAD: adres uruchomienia programu
        dta a(start)            ; adres uruchomienia: start
```

Program używa PUT CHARACTERS z jawną długością, żeby po tekście nagłówka nie przechodzić do nowego wiersza — numer sektora ma się pojawić w tej samej linii.

27.6 Rozpoznawanie formatu dyskietki

Program narzędziowy musi wiedzieć, ile sektorów ma dyskietka i jakiej są wielkości. Są dwa źródła tej wiedzy.

27.6.1 Blok PERCOM

Nowsze stacje (XF551, stacje innych firm, wszystkie współczesne urządzenia i emulatory) obsługują rozkaz `$4E` — odczyt 12-bajtowego bloku konfiguracji:

Bajt	Zawartość	Typowa wartość
0	liczba ścieżek	40
1	szybkość przesuwu głowicy	
2, 3	liczba sektorów na ścieżce (**starszy, młodszy** bajt)	18 lub 26
4	liczba stron minus 1	0
5	sposób zapisu: 0 = FM (pojedyncza gęstość), 4 = MFM	
6, 7	rozmiar sektora w bajtach (**starszy, młodszy** bajt)	128 lub 256
8	stacja gotowa (`$FF`)	
9–11	zarezerwowane	

Pułapka

Liczby 16-bitowe w bloku PERCOM zapisane są **odwrotnie** niż wszystko inne w Atari: najpierw starszy bajt, potem młodszy.

Łączna liczba sektorów = ścieżki × sektory na ścieżce × liczba stron.

Plik: percom.asm

```
; czytaj_percom - odczytuje blok PERCOM ze stacji dysków
; Wejście:  A = numer stacji (1..8)
; Wyjście:  12 bajtów w buforze "percom"; Y = status, N = 1: stacja nie zna tego rozkazu
DDEVIC  = $0300                 ; identyfikator urządzenia
DUNIT   = $0301                 ; numer urządzenia
DCOMND  = $0302                 ; rozkaz
DSTATS  = $0303                 ; kierunek transmisji / status
DBUFLO  = $0304                 ; adres bufora: młodszy bajt
DBUFHI  = $0305                 ; adres bufora: starszy bajt
DTIMLO  = $0306                 ; limit czasu
DBYTLO  = $0308                 ; liczba bajtów: młodszy bajt
DBYTHI  = $0309                 ; liczba bajtów: starszy bajt
DAUX1   = $030A                 ; parametr 1 (np. młodszy bajt numeru sektora)
DAUX2   = $030B                 ; parametr 2 (np. starszy bajt numeru sektora)
SIOV    = $E459                 ; transmisja szeregowa

czytaj_percom sta DUNIT         ; numer stacji
        lda #$31                ; urządzenie: stacja dysków
        sta DDEVIC              ; zapisz A do DDEVIC - identyfikator urządzenia
        lda #$4E                ; rozkaz: odczyt bloku PERCOM
        sta DCOMND              ; zapisz A do DCOMND - rozkaz
        lda #$40                ; kierunek: odbiór danych
        sta DSTATS              ; zapisz A do DSTATS - kierunek transmisji / status
        lda #<percom            ; adres bufora
        sta DBUFLO              ; zapisz A do DBUFLO - adres bufora: młodszy bajt
        lda #>percom            ; A = starszy bajt adresu percom
        sta DBUFHI              ; zapisz A do DBUFHI - adres bufora: starszy bajt
        lda #7                  ; limit czasu
        sta DTIMLO              ; zapisz A do DTIMLO - limit czasu
        lda #12                 ; liczba bajtów: 12
        sta DBYTLO              ; zapisz A do DBYTLO - liczba bajtów: młodszy bajt
        lda #0                  ; A = 0
        sta DBYTHI              ; zapisz A do DBYTHI - liczba bajtów: starszy bajt
        sta DAUX1               ; parametry rozkazu: nieużywane
        sta DAUX2               ; zapisz A do DAUX2 - parametr 2 (np. starszy bajt numeru sektora)
        jmp SIOV                ; skocz do SIOV - transmisja szeregowa

percom  .ds 12                  ; bufor na blok PERCOM
```

27.6.2 Rozkaz statusu

Oryginalne stacje Atari 810 i 1050 nie znają rozkazu PERCOM (odpowiadają błędem 139). Wtedy pozostaje rozkaz statusu `$53`, który zwraca 4 bajty. W pierwszym z nich:

Bit	Znaczenie, gdy ustawiony
3	dyskietka zabezpieczona przed zapisem
5	podwójna gęstość (sektory 256-bajtowe)
7	rozszerzona gęstość (1040 sektorów po 128 bajtów)

Pewny schemat rozpoznania: najpierw odczytaj sektor 1 (to zmusza stację do rozpoznania dyskietki), potem spróbuj PERCOM, a jeśli zwróci błąd — odczytaj status i sprawdź bity 7 i 5. Gdy oba są zerami, dyskietka ma 720 sektorów po 128 bajtów.

27.7 Sektor startowy: program uruchamiany bez DOS-u

Po włączeniu komputera system operacyjny próbuje wczytać program startowy ze stacji D1:. Dzieje się to tak:

1. System czyta **sektor 1** i analizuje jego pierwsze 6 bajtów — nagłówek.
2. Wczytuje podaną w nagłówku liczbę sektorów (kolejnych, od pierwszego) pod podany adres.
3. Wywołuje rozkazem `jsr` kod pod adresem **początek + 6**. Procedura powinna wrócić przez `rts` z **C = 0**; ustawiony znacznik C powoduje komunikat *BOOT ERROR*.

4. Wywołuje procedurę inicjującą, której adres był w nagłówku (zapisuje go też w wektorze DOSINI — będzie wywoływana po każdym naciśnięciu RESET).
5. Na koniec skacze przez wektor DOSVEC (`$0A`).

Bajt	Zawartość nagłówka
0	znaczniki (zwykle 0)
1	liczba sektorów do wczytania
2, 3	adres ładowania
4, 5	adres procedury inicjującej
6...	kod: tutaj system wykonuje `jsr` po wczytaniu

Program startowy może po prostu nigdy nie wrócić z kroku 3 — tak robi większość gier. Jeżeli wraca, powinien wpisać adres swojej części głównej do DOSVEC.

27.7.1 Kod sektora startowego

Kod musi być asemblowany dla adresu, pod którym będzie działał, i mieć długość będącą wielokrotnością 128 bajtów. W MADS dyrektywa `opt h-` daje plik bez nagłówków:

Plik: boot.asm

```
; ---------------------------------------------------------------------------
; boot.asm - jednosektorowy program startowy. Wynikiem asemblacji jest plik
; o długości dokładnie 128 bajtów - obraz sektora 1 dyskietki.
; ---------------------------------------------------------------------------
        opt h-                  ; bez nagłówków pliku: surowe bajty

ICCOM   = $0342                 ; rozkaz
ICBAL   = $0344                 ; adres bufora: młodszy bajt
ICBAH   = $0345                 ; adres bufora: starszy bajt
ICBLL   = $0348                 ; długość bufora: młodszy bajt
ICBLH   = $0349                 ; długość bufora: starszy bajt
CIOV    = $E456                 ; centralna procedura wejścia/wyjścia

        org $0700               ; adres, pod który system wczyta sektor

; --- nagłówek: 6 bajtów -----------------------------------------------------
        dta 0                   ; znaczniki
        dta 1                   ; liczba sektorów do wczytania: 1
        dta a($0700)            ; adres ładowania
        dta a(init)             ; adres procedury inicjującej

; --- adres $0706: tutaj system skacze po wczytaniu --------------------------
boot    ldx #0                  ; kanał 0: edytor ekranowy (jest już otwarty)
        lda #9                  ; PUT RECORD
        sta ICCOM,x             ; zapisz A pod adres ICCOM+X - rozkaz
        lda #<tekst             ; A = młodszy bajt adresu tekst
        sta ICBAL,x             ; zapisz A pod adres ICBAL+X - adres bufora: młodszy bajt
        lda #>tekst             ; A = starszy bajt adresu tekst
        sta ICBAH,x             ; zapisz A pod adres ICBAH+X - adres bufora: starszy bajt
        lda #40                 ; A = 40
        sta ICBLL,x             ; zapisz A pod adres ICBLL+X - długość bufora: młodszy bajt
        lda #0                  ; A = 0
        sta ICBLH,x             ; zapisz A pod adres ICBLH+X - długość bufora: starszy bajt
        jsr CIOV                ; wypisz komunikat
stoj    jmp stoj                ; i zatrzymaj się - to tylko demonstracja

init    rts                     ; procedura inicjująca: nic nie robi

tekst   dta c'Start z sektora 1 - dziala!',$9B  ; tekst w kodach ATASCII

        .align $0780,0          ; dopełnij zerami do pełnych 128 bajtów ($0700..$077F)
```

Otrzymany 128-bajtowy plik trzeba umieścić w sektorze 1 dyskietki albo jej obrazu. W emulatorze najprościej zbudować obraz dyskietki w formacie ATR: to 16 bajtów nagłówka, po których następują kolejne sektory. W ar-

chiwum z przykładami znajduje się gotowy obraz `boot.atr` z tym sektorem startowym — wystarczy uruchomić z niego emulator. Na prawdziwym sprzęcie sektor zapisze procedura `zapisz_sektor` z tego rozdziału.

> **Uwaga**
>
> **Zapis sektora 1 niszczy dotychczasową zawartość dyskietki** jako dyskietki systemowej, a zapis w obszarze katalogu lub mapy zajętości — wszystkie pliki. Eksperymenty z zapisem sektorów prowadź wyłącznie na dyskietkach (i obrazach), których zawartość nie ma wartości.

Gdy kod startowy ma zostać **osadzony w innym programie** (na przykład w narzędziu, które nagrywa go na dyskietkę), przydaje się dyrektywa `.local nazwa,adres`: kod wewnątrz bloku jest asemblowany tak, jakby leżał pod podanym adresem, choć fizycznie znajduje się w środku programu:

```
obraz                           ; etykieta początku danych w programie nagrywającym
        .local sektor1,$0700    ; kod poniżej jest liczony dla adresu $0700
        dta 0,1                 ; nagłówek sektora startowego
        dta a($0700),a(s_init)  ; słowa 16-bitowe (młodszy bajt pierwszy)
s_start jmp s_start             ; skok liczony względem $0700, nie względem "obraz"
s_init  rts                     ; powrót z podprogramu
        .endl                   ; koniec obszaru lokalnego
```

> **Ćwiczenia**
>
> 1. Rozbuduj `sektor.asm` o wyświetlanie zawartości również w postaci znaków (po prawej stronie każdego wiersza) — pamiętaj o zamianie na kody, które nie zakłócą wyświetlania (znaki sterujące zastąp kropką).
> 2. Napisz program kopiujący sektor 1 z D1: na D2: z użyciem procedur `czytaj_sektor` i `zapisz_sektor`.
> 3. Napisz program, który odczytuje blok PERCOM i wypisuje łączną liczbę sektorów dyskietki (potrzebne będzie mnożenie z rozdziału 14 i wypisywanie liczby z rozdziału 16).

ROZDZIAŁ 28

Pamięć dodatkowa: PORTB, RAM pod ROM-em, banki

Procesor 6502 adresuje 64 KB i ani bajta więcej. A jednak Atari 130XE ma 128 KB pamięci, popularne rozszerzenia — 320 KB, 576 KB, a nawet ponad 1 MB. Również „zwykłe" 64 KB w modelach XL/XE nie jest dostępne w całości, bo część przestrzeni adresowej zasłania ROM. Kluczem do całej tej pamięci jest jeden rejestr: **PORTB** układu PIA pod adresem `$D301`.

Zasada jest zawsze ta sama: w pewnym obszarze adresów procesor może „widzieć" jedną z kilku fizycznych pamięci, a bity rejestru PORTB wybierają którą. Nazywa się to **przełączaniem banków**.

28.1 Rejestr PORTB

Bit	Wartość	Znaczenie w Atari XL/XE
0	`$01`	**1** = ROM systemu operacyjnego włączony; **0** = w obszarach `$C000–$CFFF` i `$D800–$FFFF` widać RAM
1	`$02`	**0** = ROM BASIC-a włączony w `$A000–$BFFF`; **1** = widać tam RAM
2, 3	`$04`, `$08`	numer banku pamięci dodatkowej (130XE: banki 0–3)
4	`$10`	**0** = procesor widzi wybrany bank w `$4000–$7FFF`; **1** = widzi pamięć podstawową
5	`$20`	**0** = układ ANTIC widzi wybrany bank w `$4000–$7FFF`; **1** = pamięć podstawową (tylko 130XE)
6	`$40`	nieużywany w 130XE; w rozszerzeniach — dodatkowy bit numeru banku
7	`$80`	**0** = ROM programu testującego włączony w `$5000–$57FF`; **1** = widać RAM

Stan typowy po uruchomieniu programu z DOS-u (BASIC wyłączony) to **`$FF`**: system włączony, BASIC wyłączony, pamięć podstawowa, bez programu testującego.

Zapamiętaj

Trzy zasady pracy z PORTB:

1. **Zmieniaj tylko te bity, które cię interesują.** Odczytaj rejestr, zmodyfikuj rozkazami `and` / `ora`, zapisz. Nie zakładaj, że pozostałe bity mają określoną wartość.
2. **Zapamiętaj wartość początkową i przywróć ją** przed powrotem do DOS-u i przed każdym wywołaniem procedur systemowych.
3. **Kod, który przełącza pamięć, nie może leżeć w przełączanym obszarze** — inaczej procesor wykona rozkaz zapisu do PORTB, a następnego rozkazu już pod tym adresem nie znajdzie.

28.2 Wyłączenie BASIC-a

W modelach XL/XE interpreter BASIC-a jest wbudowany i po włączeniu komputera zajmuje obszar `$A000–$BFFF`. Użytkownik wyłącza go, trzymając klawisz OPTION podczas włączania. Program może to zrobić sam:

```
PORTB   = $D301                 ; sterowanie pamięcią
BASICF  = $03F8                 ; znacznik wyłączenia BASIC-a

        lda PORTB               ; odczytaj bieżący stan
```

```
        ora #%00000010          ; ustaw bit 1: wyłącz ROM BASIC-a
        sta PORTB               ; od tej chwili w $A000-$BFFF jest pamięć RAM
        lda #1                  ; A = 1
        sta BASICF              ; zaznacz systemowi, że BASIC ma pozostać wyłączony po RESET
```

To odsłania 8 KB pamięci RAM. Jest jednak pewien szczegół: jeżeli komputer wystartował **z włączonym** BASIC-em, system ustawił szczyt pamięci RAMTOP na `$A0` i umieścił pamięć ekranu tuż pod adresem `$A000`. Odsłonięty obszar `$A000–$BFFF` jest wolny i można go używać na dane, ale ekran pozostaje na starym miejscu. Aby system przeniósł ekran pod nowy szczyt pamięci, trzeba zmienić RAMTOP i ponownie otworzyć edytor ekranowy:

Plik: basic_off.asm

```
; basic_off - wyłącza BASIC i przenosi ekran pod nowy szczyt pamięci ($C000)
; Niszczy: A, X, Y.
PORTB   = $D301                 ; sterowanie pamięcią
BASICF  = $03F8                 ; znacznik wyłączenia BASIC-a
RAMTOP  = $6A                   ; szczyt pamięci RAM (numer strony)
ICCOM   = $0342                 ; rozkaz
ICBAL   = $0344                 ; adres bufora: młodszy bajt
ICBAH   = $0345                 ; adres bufora: starszy bajt
ICAX1   = $034A                 ; parametr 1
ICAX2   = $034B                 ; parametr 2
CIOV    = $E456                 ; centralna procedura wejścia/wyjścia

basic_off lda PORTB             ; A = PORTB - sterowanie pamięcią
        ora #%00000010          ; bit 1 = 1: ROM BASIC-a wyłączony
        sta PORTB               ; zapisz A do PORTB - sterowanie pamięcią
        lda #1                  ; A = 1
        sta BASICF              ; BASIC pozostanie wyłączony po RESET
        lda #$C0                ; A = $C0
        sta RAMTOP              ; nowy szczyt pamięci RAM: strona $C0

        ldx #$00                ; zamknij edytor ekranowy (kanał 0)...
        lda #12                 ; A = 12
        sta ICCOM,x             ; zapisz A pod adres ICCOM+X - rozkaz
        jsr CIOV                ; wywołaj CIOV: centralna procedura wejścia/wyjścia
        ldx #$00                ; ...i otwórz go ponownie: system zbuduje ekran
        lda #3                  ; pod nowym szczytem pamięci
        sta ICCOM,x             ; zapisz A pod adres ICCOM+X - rozkaz
        lda #<bo_e              ; A = młodszy bajt adresu bo_e
        sta ICBAL,x             ; zapisz A pod adres ICBAL+X - adres bufora: młodszy bajt
        lda #>bo_e              ; A = starszy bajt adresu bo_e
        sta ICBAH,x             ; zapisz A pod adres ICBAH+X - adres bufora: starszy bajt
        lda #12                 ; odczyt i zapis
        sta ICAX1,x             ; zapisz A pod adres ICAX1+X - parametr 1
        lda #0                  ; A = 0
        sta ICAX2,x             ; zapisz A pod adres ICAX2+X - parametr 2
        jmp CIOV                ; skocz do CIOV - centralna procedura wejścia/wyjścia

bo_e    dta c'E:',$9B           ; nazwa urządzenia: edytor ekranowy
```

Taką procedurę najlepiej umieścić w segmencie INIT na początku pliku (rozdział 20) — wtedy dalsze segmenty programu mogą ładować się wprost do obszaru `$A000–$BFFF`.

28.3 Pamięć RAM pod ROM-em systemu

Pod ROM-em systemu operacyjnego kryje się 14 KB pamięci RAM: `$C000–$CFFF` (4 KB) i `$D800–$FFFF` (10 KB). Obszar `$D000–$D7FF` to rejestry sprzętowe — pod nimi pamięci nie ma.

Kłopot polega na tym, że wyłączając ROM, **wyłączamy system operacyjny**: znikają procedury obsługi przerwań, a wektory procesora pod `$FFFA–$FFFF` zaczynają wskazywać przypadkowe adresy z pamięci RAM. Pierwsze przerwanie zawiesiłoby komputer. Są trzy sposoby poradzenia sobie z tym, od najprostszego do najdalej idącego.

28.3.1 Sposób 1: krótki dostęp przy wyłączonych przerwaniach

Jeśli pamięć pod ROM-em ma służyć jako **magazyn danych** (grafika następnego poziomu, tablice, bufor), wystarczy wyłączać ROM na czas kopiowania:

Plik: pod_rom.asm

```
; rom_wylacz / rom_wlacz - odsłonięcie i zasłonięcie pamięci RAM pod ROM-em systemu
; Między wywołaniami nie wolno używać żadnych procedur systemowych!
PORTB   = $D301                 ; sterowanie pamięcią
NMIEN   = $D40E                 ; zapis: maska przerwań NMI
pr_portb = $80                  ; zapamiętany stan rejestru PORTB

rom_wylacz sei                  ; zablokuj przerwania IRQ
        lda #0                  ; A = 0
        sta NMIEN               ; wyłącz przerwania NMI (VBI, DLI)
        lda PORTB               ; A = PORTB - sterowanie pamięcią
        sta pr_portb            ; zapamiętaj stan PORTB
        and #%11111110          ; bit 0 = 0: wyłącz ROM systemu
        sta PORTB               ; teraz pod $C000-$CFFF i $D800-$FFFF jest RAM
        rts                     ; powrót z podprogramu

rom_wlacz lda pr_portb          ; przywróć zapamiętany stan PORTB (ROM włączony)
        sta PORTB               ; zapisz A do PORTB - sterowanie pamięcią
        lda #$40                ; A = $40
        sta NMIEN               ; włącz przerwanie VBI
        cli                     ; odblokuj IRQ
        rts                     ; powrót z podprogramu
```

Użycie: `jsr rom_wylacz`, kopiowanie danych zwykłą pętlą (np. `memcpy` z rozdziału 18), `jsr rom_wlacz`. Dopóki ROM jest wyłączony, ekran może na ułamek sekundy pokazać przypadkowe znaki (zestaw znaków leży w ROM-ie pod `$E000`), stoi zegar i nie działa klawiatura — dlatego rób to krótko.

Pułapka

Zapis pod adres z obszaru ROM-u **przy włączonym ROM-ie** nie trafia do pamięci RAM pod spodem — po prostu przepada. Żeby zapisać coś pod ROM, trzeba ROM wyłączyć.

28.3.2 Sposób 2: system operacyjny skopiowany do RAM-u

Można skopiować cały ROM systemu do leżącej pod nim pamięci RAM i wyłączyć ROM. Komputer działa wtedy dokładnie tak samo, ale system jest w pamięci RAM — i można go **modyfikować**: podmienić zestaw znaków na miejscu (polskie litery widoczne także w DOS-ie), poprawić procedury, przyspieszyć transmisję. Ponieważ źródło i cel mają te same adresy, kopiujemy strona po stronie przez bufor:

Plik: os_do_ram.asm

```
; os_do_ram - kopiuje ROM systemu operacyjnego do pamięci RAM pod nim i wyłącza ROM
; Po wykonaniu system działa z pamięci RAM i może być modyfikowany.
; Niszczy: A, X, Y, zmienną or_wsk. Wymaga 256-bajtowego bufora "or_bufor"
; leżącego poza obszarem ROM-u. Procedura także musi leżeć poza tym obszarem.
PORTB   = $D301                 ; sterowanie pamięcią
NMIEN   = $D40E                 ; zapis: maska przerwań NMI
or_wsk  = $80                   ; wskaźnik na kopiowaną stronę (2 bajty)

os_do_ram sei                   ; zablokuj IRQ
        lda #0                  ; A = 0
        sta NMIEN               ; wyłącz NMI
        sta or_wsk              ; młodszy bajt wskaźnika = 0 (kopiujemy całe strony)
        lda #$C0                ; zaczynamy od strony $C0
        sta or_wsk+1            ; zapisz A pod adres or_wsk+1
or_1    lda or_wsk+1            ; czy to obszar rejestrów sprzętowych $D0..$D7 ?
        cmp #$D0                ; porównaj A z: $D0
        bcc or_2                ; strona < $D0: kopiuj
        cmp #$D8                ; porównaj A z: $D8
        bcc or_4                ; strona $D0..$D7: pomiń
or_2    ldy #0                  ; --- krok 1: ROM -> bufor (ROM jest włączony) ---
or_3a   lda (or_wsk),y          ; A = bajt spod adresu (or_wsk) + Y
        sta or_bufor,y          ; zapisz A pod adres or_bufor+Y
        iny                     ; Y = Y + 1
        bne or_3a               ; skocz do or_3a, gdy Z = 0 (różne / nie zero)
        lda PORTB               ; --- krok 2: wyłącz ROM ---
        and #%11111110          ; A = A AND %11111110
        sta PORTB               ; zapisz A do PORTB - sterowanie pamięcią
or_3b   lda or_bufor,y          ; --- krok 3: bufor -> RAM pod tym samym adresem (Y = 0) ---
```

```
        sta (or_wsk),y          ; zapisz A pod adres (or_wsk) + Y
        iny                     ; Y = Y + 1
        bne or_3b               ; skocz do or_3b, gdy Z = 0 (różne / nie zero)
        lda PORTB               ; --- krok 4: włącz ROM z powrotem ---
        ora #%00000001          ; A = A OR %00000001
        sta PORTB               ; zapisz A do PORTB - sterowanie pamięcią
or_4    inc or_wsk+1            ; następna strona
        bne or_1                ; po stronie $FF licznik wraca do zera: koniec
        lda PORTB               ; wszystko skopiowane: wyłącz ROM na stałe -
        and #%11111110          ; system działa odtąd z pamięci RAM
        sta PORTB               ; zapisz A do PORTB - sterowanie pamięcią
        lda #$40                ; A = $40
        sta NMIEN               ; włącz VBI
        cli                     ; odblokuj IRQ
        rts                     ; powrót z podprogramu
```

Naciśnięcie klawisza RESET przywraca ROM, więc program, który chce utrzymać system w pamięci RAM, musi przechwycić wektor DOSINI i powtarzać operację po każdym resecie.

28.3.3 Sposób 3: całkowita rezygnacja z systemu

Gry i dema zwykle w ogóle nie potrzebują systemu operacyjnego. Wyłączają ROM na stałe, zyskując ciągły obszar pamięci RAM aż do `$FFFF` (z dziurą na rejestry sprzętowe), i **instalują własne procedury przerwań** bezpośrednio w wektorach procesora. Cena: żadnych procedur systemowych, żadnych cieni — klawiaturę, joystick i kolory obsługuje się, pisząc wprost do sprzętu.

Plik: bez_os.asm

```
; ------------------------------------------------------------------------
; bez_os.asm - praca z wyłączonym ROM-em systemu i własną obsługą przerwania VBI.
; Ramka ekranu zmienia kolor co 1/50 s. Klawisz START przywraca system i kończy.
; ------------------------------------------------------------------------
PORTB   = $D301                 ; sterowanie pamięcią
COLBK   = $D01A                 ; sprzętowy rejestr koloru tła / ramki
CONSOL  = $D01F                 ; klawisze konsoli
CHBASE  = $D409                 ; sprzętowy rejestr adresu zestawu znaków
NMIEN   = $D40E                 ; maska przerwań NMI
NMIRES  = $D40F                 ; zapis: skasowanie zgłoszenia NMI

ZNAKI   = $3000                 ; kopia zestawu znaków w pamięci RAM (adres podzielny przez $400)

zr      = $80                   ; wskaźnik źródła
cel     = $82                   ; wskaźnik celu
stary   = $84                   ; zapamiętany stan PORTB
licznik = $85                   ; licznik ramek (zwiększany w przerwaniu)

        org $2000               ; adres asemblacji: $2000

; --- 1. skopiuj zestaw znaków z ROM-u do RAM-u (po wyłączeniu ROM-u zniknie) ---
start   lda #<$E000             ; A = młodszy bajt wartości $E000
        sta zr                  ; zapisz A do zr
        lda #>$E000             ; A = starszy bajt wartości $E000
        sta zr+1                ; zapisz A pod adres zr+1
        lda #<ZNAKI             ; A = młodszy bajt wartości ZNAKI
        sta cel                 ; zapisz A do cel
        lda #>ZNAKI             ; A = starszy bajt wartości ZNAKI
        sta cel+1               ; zapisz A pod adres cel+1
        ldx #4                  ; 4 strony = 1024 bajty
        ldy #0                  ; Y = 0
kop     lda (zr),y              ; A = bajt spod adresu (zr) + Y
        sta (cel),y             ; zapisz A pod adres (cel) + Y
        iny                     ; Y = Y + 1
        bne kop                 ; skocz do kop, gdy Z = 0 (różne / nie zero)
        inc zr+1                ; zwiększ o 1 bajt spod adresu zr+1
        inc cel+1               ; zwiększ o 1 bajt spod adresu cel+1
        dex                     ; X = X - 1
        bne kop                 ; skocz do kop, gdy Z = 0 (różne / nie zero)

; --- 2. wyłącz przerwania i ROM systemu ----------------------------------
        sei                     ; IRQ zablokowane na cały czas pracy bez systemu
        lda #0                  ; A = 0
```

```
        sta NMIEN               ; NMI wyłączone na czas przełączania
        lda PORTB               ; A = PORTB - sterowanie pamięcią
        sta stary               ; zapamiętaj stan PORTB
        and #%11111110          ; bit 0 = 0: ROM systemu wyłączony
        sta PORTB               ; zapisz A do PORTB - sterowanie pamięcią
        lda #>ZNAKI             ; każ układowi ANTIC brać znaki z naszej kopii -
        sta CHBASE              ; zapis wprost do sprzętu, bo cieni nikt już nie przepisuje

; --- 3. wpisz własne adresy do wektorów procesora (teraz to pamięć RAM) ------
        lda #<nmi               ; wektor NMI: $FFFA/$FFFB
        sta $FFFA               ; zapisz A do komórki $FFFA
        lda #>nmi               ; A = starszy bajt adresu nmi
        sta $FFFB               ; zapisz A do komórki $FFFB
        lda #<irq               ; wektor IRQ: $FFFE/$FFFF (na wszelki wypadek)
        sta $FFFE               ; zapisz A do komórki $FFFE
        lda #>irq               ; A = starszy bajt adresu irq
        sta $FFFF               ; zapisz A do komórki $FFFF
        lda #$40                ; A = $40
        sta NMIEN               ; włącz przerwanie VBI - obsłuży je nasza procedura

; --- 4. program główny: czekaj na klawisz START ----------------------------
petla   lda CONSOL              ; odczyt wprost ze sprzętu
        and #%00000001          ; bit 0: START (0 = wciśnięty)
        bne petla               ; skocz do petla, gdy Z = 0 (różne / nie zero)

; --- 5. przywróć system ----------------------------------------------------
        lda #0                  ; A = 0
        sta NMIEN               ; wyłącz NMI na czas przełączania
        lda stary               ; A = stary
        sta PORTB               ; włącz ROM systemu
        lda #$40                ; A = $40
        sta NMIEN               ; włącz VBI - znów obsługuje je system; przepisze też cienie,
        cli                     ; przywracając zestaw znaków i kolory
        rts                     ; powrót z podprogramu

; --- własna obsługa przerwania NMI (tu: tylko VBI) --------------------------
nmi     pha                     ; system nam nie pomoże: sami zachowujemy rejestry
        sta NMIRES              ; skasuj zgłoszenie przerwania w układzie ANTIC
        inc licznik             ; policz ramkę
        lda licznik             ; A = licznik
        sta COLBK               ; kolor ramki = licznik ramek (zapis wprost do sprzętu)
        pla                     ; zdejmij A ze stosu
        rti                     ; powrót z przerwania

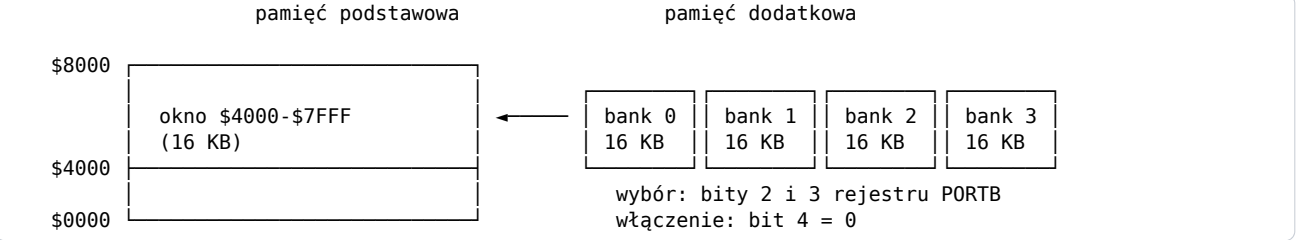
irq     rti                     ; IRQ są zablokowane; gdyby jednak wystąpiło - po prostu wróć

        org $02E0               ; wektor RUNAD: adres uruchomienia programu
        dta a(start)            ; adres uruchomienia: start
```

Gdyby włączone było także przerwanie DLI, procedura `nmi` musiałaby zacząć od rozpoznania źródła: rozkaz `bit NMIST` (`$D40F`) kopiuje do znacznika N bit 7 rejestru, ustawiony właśnie dla DLI.

28.4 Banki pamięci Atari 130XE

Atari 130XE ma dodatkowe 64 KB pamięci podzielone na **cztery banki po 16 KB**. W danej chwili jeden z nich może zastąpić pamięć podstawową w **oknie `$4000–$7FFF`**:



Wartości PORTB dla 130XE (system włączony, BASIC wyłączony, ANTIC widzi pamięć podstawową):

Co widać w oknie `$4000–$7FFF`	PORTB	Bity 7–0
pamięć podstawowa	`$FF`	`1111 1111`
bank 0	`$E3`	`1110 0011`
bank 1	`$E7`	`1110 0111`
bank 2	`$EB`	`1110 1011`
bank 3	`$EF`	`1110 1111`

Bit 5 pozwala w 130XE osobno zdecydować, co widzi **układ ANTIC**. Dzięki temu obraz może leżeć w banku, podczas gdy procesor pracuje na pamięci podstawowej — albo odwrotnie. To cecha tylko oryginalnego 130XE; większość rozszerzeń jej nie ma.

28.4.1 Zasady bezpiecznej pracy z bankami

1. **Kod przełączający banki — poza oknem.** Cały kod działający przy włączonym banku oraz jego dane muszą leżeć poniżej `$4000` albo powyżej `$7FFF`. Najprościej: umieść program od `$2000` i pilnuj, by nie przekroczył `$3FFF`, albo zacznij go od `$8000`.
2. **Procedury przerwań i ich dane — poza oknem.** Przerwanie VBI może nadejść w chwili, gdy bank jest włączony.
3. **Pamięć obrazu, display list, zestaw znaków, dane duszków — poza oknem** (chyba że świadomie korzystasz z bitu 5 w 130XE). Systemowa pamięć ekranu leży tuż pod szczytem pamięci RAM, a więc bezpiecznie powyżej okna; uważać trzeba na własne bufory obrazu.
4. **Stos i strona zerowa** nie są przełączane — można z nich korzystać zawsze.
5. **Przed wywołaniem systemu przywróć pamięć podstawową**, chyba że dokładnie wiesz, co robisz. DOS i jego bufory mogą leżeć w oknie.
6. **Sprawdź, czy banki nie są zajęte.** RAM-dysk (np. program RAMDISK.COM z DOS 2.5 albo RAM-dysk SpartaDOS-u) przechowuje w bankach pliki. Zapis do banków zniszczy jego zawartość.
7. **Nie zakładaj, że banki są** — wykryj je (poniżej).

28.4.2 Przełączanie i kopiowanie

Numer banku najwygodniej zamieniać na wartość PORTB tablicą — wypełnioną przez procedurę wykrywającą albo, gdy program jest przeznaczony wyłącznie dla 130XE, wpisaną na stałe:

Plik: bank_io.asm

```
; Podstawowe operacje na bankach pamięci. Wszystkie procedury i tablica
; muszą leżeć poza obszarem $4000-$7FFF.
PORTB   = $D301                 ; sterowanie pamięcią
bk_portb = $80                  ; zapamiętany stan PORTB dla pamięci podstawowej
bk_wektor = $81                 ; adres procedury wywoływanej w banku (2 bajty)

; bank_init - zapamiętuje stan PORTB odpowiadający pamięci podstawowej.
; Wywołaj raz, na początku programu.
bank_init lda PORTB             ; A = PORTB - sterowanie pamięcią
        ora #%00010000          ; bit 4 = 1: procesor widzi pamięć podstawową
        sta bk_portb            ; zapisz A do bk_portb
        rts                     ; powrót z podprogramu

; bank_wlacz - włącza bank o numerze X (0 = pierwszy bank z tablicy).
bank_wlacz lda tab_bankow,x     ; wartość PORTB dla tego banku
        sta PORTB               ; od tej chwili w $4000-$7FFF widać bank
        rts                     ; powrót z podprogramu

; bank_wylacz - przywraca pamięć podstawową w oknie $4000-$7FFF.
bank_wylacz lda bk_portb        ; A = bk_portb
        sta PORTB               ; zapisz A do PORTB - sterowanie pamięcią
        rts                     ; powrót z podprogramu

; bank_wywolaj - wywołuje procedurę umieszczoną w banku.
; Wejście: X = numer banku, bk_wektor = adres procedury (w obszarze $4000-$7FFF).
; Rejestr Y jest przekazywany do procedury i z powrotem; A jest niszczony.
bank_wywolaj lda tab_bankow,x   ; A = bajt spod adresu tab_bankow+X
```

```
        sta PORTB               ; włącz bank z kodem procedury
        jsr bk_skok             ; wywołaj ją przez skok pośredni
        lda bk_portb            ; po powrocie: przywróć pamięć podstawową
        sta PORTB               ; zapisz A do PORTB - sterowanie pamięcią
        rts                     ; powrót z podprogramu
bk_skok jmp (bk_wektor)         ; trampolina: RTS procedury wróci do bank_wywolaj

; Tablica wartości PORTB dla kolejnych banków - tu wpisana na stałe dla 130XE.
tab_bankow dta $E3,$E7,$EB,$EF  ; dane: bajty
```

Kopiowanie danych do banku nie wymaga niczego szczególnego — pod warunkiem, że źródło leży **poza oknem**:

```
; Skopiuj 4 KB danych spod $8000 (pamięć podstawowa) na początek banku 2
        ldx #2                  ; bank numer 2
        jsr bank_wlacz          ; w oknie $4000-$7FFF widać teraz bank 2
        lda #<$8000             ; źródło: poza oknem, więc nadal widoczne
        sta mc_zr               ; zapisz A do mc_zr
        lda #>$8000             ; A = starszy bajt wartości $8000
        sta mc_zr+1             ; zapisz A pod adres mc_zr+1
        lda #<$4000             ; cel: początek okna, czyli początek banku
        sta mc_cel              ; zapisz A do mc_cel
        lda #>$4000             ; A = starszy bajt wartości $4000
        sta mc_cel+1            ; zapisz A pod adres mc_cel+1
        lda #<$1000             ; długość: 4096 bajtów
        sta mc_dl               ; zapisz A do mc_dl
        lda #>$1000             ; A = starszy bajt wartości $1000
        sta mc_dl+1             ; zapisz A pod adres mc_dl+1
        jsr memcpy              ; zwykła procedura kopiująca (leżąca poza oknem!)
        jsr bank_wylacz         ; przywróć pamięć podstawową
```

Aby skopiować dane **z banku do banku** albo z okna pamięci podstawowej do banku, potrzebny jest bufor pośredni poza oknem: bank źródłowy → bufor → bank docelowy, porcjami.

28.4.3 Wykrywanie banków

Jak sprawdzić, ile banków ma komputer? Pomysł jest prosty: wpisujemy do tego samego adresu okna, przy kolejnych ustawieniach PORTB, **różne liczby**. Jeżeli dwa ustawienia pokazują w rzeczywistości tę samą pamięć, późniejszy zapis zamaże wcześniejszy. Na koniec sprawdzamy, które liczby przetrwały — tyle jest różnych banków.

Poniższa procedura sprawdza wszystkie 64 kombinacje sześciu bitów używanych przez różne rozszerzenia (bity 1, 2, 3, 5, 6, 7), więc działa zarówno na 130XE, jak i na rozszerzeniach do 1 MB. Kombinacje bada w takiej kolejności, by spośród równoważnych ustawień wybrać to o najwyższej wartości — z wyłączonym BASIC-em i programem testującym. Na 130XE daje to dokładnie wartości `$E3`, `$E7`, `$EB`, `$EF`.

Plik: banki.asm

```
; ------------------------------------------------------------------------
; banki.asm - wykrywa banki pamięci dodatkowej i wypisuje ich wartości PORTB
; UWAGA: cały program musi leżeć poza obszarem $4000-$7FFF.
; ------------------------------------------------------------------------
PORTB   = $D301                 ; sterowanie pamięcią
NMIEN   = $D40E                 ; zapis: maska przerwań NMI
ICCOM   = $0342                 ; rozkaz
ICBAL   = $0344                 ; adres bufora: młodszy bajt
ICBAH   = $0345                 ; adres bufora: starszy bajt
ICBLL   = $0348                 ; długość bufora: młodszy bajt
ICBLH   = $0349                 ; długość bufora: starszy bajt
CIOV    = $E456                 ; centralna procedura wejścia/wyjścia
CH      = $02FC                 ; kod klawisza
EOL     = $9B                   ; znak końca linii

PROBA   = $7FFF                 ; komórka testowa w oknie banków (poza ROM-em programu testującego)

tmp     = $80                   ; zmienna pomocnicza
glowna  = $81                   ; wartość PORTB dla pamięci podstawowej
zapas_gl = $82                  ; zachowany bajt komórki testowej pamięci podstawowej
ile_bankow = $83                ; wynik: liczba wykrytych banków (0..64)
numer   = $84                   ; licznik pętli wypisującej
```

```
        org $2000               ; adres asemblacji: $2000

start   jsr wykryj_banki        ; wykryj banki; wynik w ile_bankow i tab_bankow

        lda #<t_ile             ; wypisz "Banki: "
        ldy #>t_ile             ; Y = starszy bajt adresu t_ile
        jsr print               ; wywołaj podprogram print
        lda ile_bankow          ; liczba banków szesnastkowo
        jsr hex                 ; wywołaj podprogram hex
        lda #EOL                ; A = EOL
        jsr putchar             ; wywołaj podprogram putchar

        lda #0                  ; wypisz wartości PORTB kolejnych banków
        sta numer               ; zapisz A do numer
wyp     lda numer               ; A = numer
        cmp ile_bankow          ; wszystkie wypisane?
        beq koniec              ; skocz do koniec, gdy Z = 1 (równe / zero)
        tax                     ; X = A
        lda tab_bankow,x        ; wartość PORTB banku numer X
        jsr hex                 ; wywołaj podprogram hex
        lda #' '                ; A = kod ATASCII znaku ' '
        jsr putchar             ; wywołaj podprogram putchar
        inc numer               ; numer = numer + 1
        bne wyp                 ; (skok zawsze)

koniec  lda #EOL                ; A = EOL
        jsr putchar             ; wywołaj podprogram putchar
        lda #$FF                ; czekaj na klawisz
        sta CH                  ; zapisz A do CH - kod klawisza
czekaj  lda CH                  ; A = CH - kod klawisza
        cmp #$FF                ; porównaj A z: $FF
        beq czekaj              ; skocz do czekaj, gdy Z = 1 (równe / zero)
        lda #$FF                ; A = $FF
        sta CH                  ; zapisz A do CH - kod klawisza
        rts                     ; powrót z podprogramu

; ------------------------------------------------------------------------
; wykryj_banki - wykrywa banki pamięci dodatkowej
; Wyjście:  ile_bankow = liczba banków (0 = brak pamięci dodatkowej)
;           tab_bankow = wartości PORTB kolejnych banków
; Niszczy:  A, X, Y. Zawartość banków i pamięci podstawowej pozostaje bez zmian.
; ------------------------------------------------------------------------
wykryj_banki sei                ; żadnych przerwań podczas przełączania pamięci
        lda #0                  ; A = 0
        sta NMIEN               ; zapisz A do NMIEN - zapis: maska przerwań NMI
        lda PORTB               ; ustal wartość PORTB dla pamięci podstawowej:
        ora #%00010000          ; bieżący stan z bitem 4 = 1
        sta glowna              ; zapisz A do glowna
        sta PORTB               ; zapisz A do PORTB - sterowanie pamięcią
        lda PROBA               ; zachowaj bajt komórki testowej pamięci podstawowej
        sta zapas_gl            ; zapisz A do zapas_gl

        ldx #63                 ; --- krok 1: zachowaj bajt testowy każdej z 64 kombinacji ---
wb_1    jsr ustaw               ; PORTB := kombinacja numer X
        lda PROBA               ; A = PROBA
        sta zapas,x             ; zapisz A pod adres zapas+X
        dex                     ; X = X - 1
        bpl wb_1                ; skocz do wb_1, gdy N = 0 (bit 7 skasowany)

        ldx #0                  ; --- krok 2: wpisz numer kombinacji do komórki testowej ---
wb_2    jsr ustaw               ; (rosnąco: gdy kilka kombinacji wybiera ten sam bank,
        stx PROBA               ; w komórce zostanie numer NAJWYŻSZEJ z nich)
        inx                     ; X = X + 1
        cpx #64                 ; porównaj X z: 64
        bne wb_2                ; skocz do wb_2, gdy Z = 0 (różne / nie zero)

        lda glowna              ; --- krok 3: oznacz pamięć podstawową wartością $FF ---
        sta PORTB               ; (jeśli kombinacja pokazuje pamięć podstawową,
        lda #$FF                ; jej numer zostanie w ten sposób zamazany)
        sta PROBA               ; zapisz A do PROBA

        ldy #0                  ; --- krok 4: policz kombinacje, których numer przetrwał ---
        ldx #0                  ; X = 0
```

```
wb_4    jsr ustaw               ; PORTB := kombinacja numer X; A = wpisana wartość
        cpx PROBA               ; czy w komórce testowej nadal jest numer X ?
        bne wb_5                ; nie: to powtórzenie innego banku albo pamięć podstawowa
        sta tab_bankow,y        ; tak: zapamiętaj wartość PORTB tego banku
        iny                     ; Y = Y + 1
wb_5    inx                     ; X = X + 1
        cpx #64                 ; porównaj X z: 64
        bne wb_4                ; skocz do wb_4, gdy Z = 0 (różne / nie zero)
        sty ile_bankow          ; zapisz Y do ile_bankow

        ldx #63                 ; --- krok 5: przywróć zachowane bajty testowe ---
wb_6    jsr ustaw               ; wywołaj podprogram ustaw
        lda zapas,x             ; A = bajt spod adresu zapas+X
        sta PROBA               ; zapisz A do PROBA
        dex                     ; X = X - 1
        bpl wb_6                ; skocz do wb_6, gdy N = 0 (bit 7 skasowany)
        lda glowna              ; przywróć pamięć podstawową i jej bajt testowy
        sta PORTB               ; zapisz A do PORTB - sterowanie pamięcią
        lda zapas_gl            ; A = zapas_gl
        sta PROBA               ; zapisz A do PROBA

        lda #$40                ; włącz z powrotem przerwania
        sta NMIEN               ; zapisz A do NMIEN - zapis: maska przerwań NMI
        cli                     ; odblokuj przerwania IRQ
        rts                     ; powrót z podprogramu

; --- ustaw: zamienia numer kombinacji X (0..63) na wartość PORTB i zapisuje ją
; Bity numeru %00abcdef trafiają do PORTB jako %abc0def1:
;   bit 4 = 0 (procesor widzi bank), bit 0 = 1 (ROM systemu włączony).
; Wyjście: A = zapisana wartość PORTB. Zachowuje X i Y.
ustaw   txa                     ; A = X
        and #%00111000          ; bity a, b, c numeru...
        asl @                   ; ...przesuń o dwie pozycje w lewo: na bity 7, 6, 5
        asl @                   ; przesuń A o 1 bit w lewo (bit 7 -> C)
        sta tmp                 ; zapisz A do tmp
        txa                     ; A = X
        and #%00000111          ; bity d, e, f numeru...
        asl @                   ; ...przesuń o jedną pozycję w lewo: na bity 3, 2, 1
        ora tmp                 ; połącz obie części
        ora #%00000001          ; bit 0 = 1: ROM systemu pozostaje włączony
        sta PORTB               ; zapisz A do PORTB - sterowanie pamięcią
        rts                     ; powrót z podprogramu

; --- hex: wypisuje A jako dwie cyfry szesnastkowe ---------------------------
hex     pha                     ; odłóż A na stos
        lsr @                   ; przesuń A o 1 bit w prawo (bit 0 -> C)
        lsr @                   ; przesuń A o 1 bit w prawo (bit 0 -> C)
        lsr @                   ; przesuń A o 1 bit w prawo (bit 0 -> C)
        lsr @                   ; przesuń A o 1 bit w prawo (bit 0 -> C)
        tax                     ; X = A
        lda cyfry,x             ; A = bajt spod adresu cyfry+X
        jsr putchar             ; wywołaj podprogram putchar
        pla                     ; zdejmij A ze stosu
        and #$0F                ; A = A AND $0F
        tax                     ; X = A
        lda cyfry,x             ; A = bajt spod adresu cyfry+X
        jmp putchar             ; skocz do putchar

; --- putchar: wypisuje znak z A ---------------------------------------------
putchar ldx #$00                ; X = $00
        pha                     ; odłóż A na stos
        lda #11                 ; A = 11
        sta ICCOM,x             ; zapisz A pod adres ICCOM+X - rozkaz
        lda #0                  ; A = 0
        sta ICBLL,x             ; zapisz A pod adres ICBLL+X - długość bufora: młodszy bajt
        sta ICBLH,x             ; zapisz A pod adres ICBLH+X - długość bufora: starszy bajt
        pla                     ; zdejmij A ze stosu
        jmp CIOV                ; skocz do CIOV - centralna procedura wejścia/wyjścia

; --- print: wypisuje 7 znaków spod adresu A/Y -------------------------------
print   ldx #$00                ; X = $00
        sta ICBAL,x             ; zapisz A pod adres ICBAL+X - adres bufora: młodszy bajt
```

```
        tya                     ; A = Y
        sta ICBAH,x             ; zapisz A pod adres ICBAH+X - adres bufora: starszy bajt
        lda #11                 ; A = 11
        sta ICCOM,x             ; zapisz A pod adres ICCOM+X - rozkaz
        lda #7                  ; A = 7
        sta ICBLL,x             ; zapisz A pod adres ICBLL+X - długość bufora: młodszy bajt
        lda #0                  ; A = 0
        sta ICBLH,x             ; zapisz A pod adres ICBLH+X - długość bufora: starszy bajt
        jmp CIOV                ; skocz do CIOV - centralna procedura wejścia/wyjścia

cyfry   dta c'0123456789ABCDEF'  ; tekst w kodach ATASCII
t_ile   dta c'Banki: '          ; tekst w kodach ATASCII
tab_bankow .ds 64               ; wartości PORTB wykrytych banków
zapas   .ds 64                  ; zachowane bajty komórki testowej

        .if *>$4000             ; kontrola w czasie asemblacji:
        .error "Program wchodzi w obszar bankow!"  ; przerwij asemblację
        .endif                  ; koniec bloku warunkowego

        org $02E0               ; wektor RUNAD: adres uruchomienia programu
        dta a(start)            ; adres uruchomienia: start
```

Liczba banków jest wypisywana szesnastkowo: `04` dla 130XE, `10` (szesnaście) dla rozszerzenia 320 KB, `40` (sześćdziesiąt cztery) dla 1 MB, `00` dla komputera bez pamięci dodatkowej.

28.4.4 Rozszerzenia pamięci

Rozszerzenia montowane w Atari XL/XE stosują tę samą zasadę co 130XE — okno `$4000`–`$7FFF` i bit 4 jako włącznik — ale używają dodatkowych bitów PORTB do numerowania banków:

Pamięć razem	Banków	Bity numeru banku	Uwagi
128 KB (130XE)	4	2, 3	bit 5: osobny dostęp układu ANTIC
192 KB	8	2, 3, 6	
256 KB (typ RAMBO)	16	2, 3, 5, 6	brak osobnego dostępu ANTIC
320 KB (typ Compy Shop)	16	2, 3, 6, 7	bit 5 jak w 130XE
576 KB	32	1, 2, 3, 6, 7	bit 1 traci funkcję wyłącznika BASIC-a
1088 KB	64	1, 2, 3, 5, 6, 7	

Wniosek jest jeden: **nie wpisuj wartości PORTB na stałe**, jeśli program ma działać na czymś więcej niż fabryczne 130XE. Wykryj banki i korzystaj z tablicy.

28.4.5 Ładowanie danych z pliku wprost do banków

Mechanizm segmentów INIT (rozdział 20) pozwala załadować dane do banku bez żadnego kopiowania: krótka procedura wykonana w trakcie ładowania przełącza bank, następny segment pliku trafia do okna — czyli do banku — a kolejna procedura INIT przywraca pamięć podstawową.

Plik: banki_plik.asm

```
; ------------------------------------------------------------------------
; banki_plik.asm - dane ładowane z pliku bezpośrednio do banku (wersja dla 130XE)
; ------------------------------------------------------------------------
PORTB   = $D301                 ; sterowanie pamięcią

; === segment 1: procedury przełączające - na stronie 6, poza oknem ==========
        org $0600               ; adres asemblacji: $0600
wl_bank0 lda #$E3               ; włącz bank 0 w oknie $4000-$7FFF
        sta PORTB               ; zapisz A do PORTB - sterowanie pamięcią
        rts                     ; powrót z podprogramu
wl_glowna lda #$FF              ; przywróć pamięć podstawową
        sta PORTB               ; zapisz A do PORTB - sterowanie pamięcią
        rts                     ; powrót z podprogramu

; === segment 2: INIT - DOS natychmiast wykona wl_bank0 ======================
        org $02E2               ; wektor INITAD: procedura wykonywana podczas ładowania
```

```
        dta a(wl_bank0)         ; adres procedury: wl_bank0

; === segment 3: te dane załadują się już do BANKU 0 ========================
        org $4000               ; adres asemblacji: $4000
dane_b0 dta c'Te dane leza w banku 0',$9B  ; tekst w kodach ATASCII

; === segment 4: INIT - DOS wykona wl_glowna, wracamy do pamięci podstawowej =
        org $02E2               ; wektor INITAD: procedura wykonywana podczas ładowania
        dta a(wl_glowna)        ; adres procedury: wl_glowna

; === segment 5: program główny w pamięci podstawowej ========================
        org $2000               ; adres asemblacji: $2000
start   jsr wl_bank0            ; włącz bank 0, żeby sięgnąć do danych
        lda dane_b0             ; A = pierwszy bajt danych z banku (litera T)
        sta $0680               ; zapamiętaj go w pamięci podstawowej
        jsr wl_glowna           ; przywróć pamięć podstawową
        rts                     ; powrót z podprogramu

        org $02E0               ; wektor RUNAD: adres uruchomienia programu
        dta a(start)            ; adres uruchomienia: start
```

Uwaga

Ta technika działa pod warunkiem, że **DOS nie korzysta z obszaru okna** podczas ładowania — ani jego kod, ani bufory nie mogą leżeć w `$4000`–`$7FFF`. Klasyczne DOS-y (DOS 2.5, MyDOS) spełniają ten warunek. SpartaDOS X w niektórych konfiguracjach sam używa banków pamięci dodatkowej — wtedy bezpieczniejsze jest wczytanie danych do pamięci podstawowej i skopiowanie ich do banku własną procedurą.

MADS ma własne dyrektywy wspierające banki (`lmb`, `nmb`, `rmb`) i gotową procedurę wykrywania pamięci, opisane w jego dokumentacji. Po lekturze tego rozdziału wiesz już dokładnie, co robią „pod spodem”.

Ćwiczenia

1. Napisz procedurę `bank_czytaj`, która zwraca w A bajt spod adresu `bk_adr` (w obszarze okna) w banku numer X, zachowując bieżący stan PORTB.
2. Napisz procedurę kopiującą 256 bajtów z banku X do banku Y przez bufor w pamięci podstawowej.
3. Zmodyfikuj `banki.asm` tak, by wypisywał łączną ilość pamięci dodatkowej w kilobajtach (liczba banków · 16).

ROZDZIAŁ 29

Warsztat: debugowanie, typowe błędy, optymalizacja

Znajomość rozkazów to jedno; umiejętność doprowadzenia programu do działania — drugie. Ten rozdział zbiera praktyczne rady: jak szukać błędów, czego się wystrzegać i jak sprawić, by program był szybszy lub krótszy.

29.1 Debugowanie w emulatorze Altirra

Altirra ma wbudowany debugger na poziomie profesjonalnych narzędzi. Jeśli obok pliku `program.xex` leżą pliki `program.lab` i `program.lst` (opcje `-t` i `-l` asemblera MADS), debugger wczyta je sam i będzie pokazywał **twoje etykiety i linie źródła**.

29.1.1 Podstawowe klawisze

Klawisz	Działanie
F8	zatrzymaj program / wznów wykonywanie
F11	wykonaj jeden rozkaz (wchodząc do podprogramów)
F10	wykonaj jeden rozkaz (podprogram `jsr` jako całość)
Shift+F11	wykonuj do powrotu z bieżącego podprogramu

W menu *Debug* → *Window* włącz okna: *Registers* (rejestry i znaczniki), *Disassembly* (kod), *Memory* (zawartość pamięci), *Console* (wiersz poleceń), *Watch* (podgląd wybranych zmiennych).

29.1.2 Najprzydatniejsze polecenia konsoli

Polecenie	Działanie
`bp adres`	pułapka: zatrzymaj, gdy procesor dojdzie do adresu (można podać etykietę)
`ba w adres`	zatrzymaj przy **zapisie** do komórki — bezcenne, gdy „coś" psuje zmienną
`ba r adres`	zatrzymaj przy odczycie komórki
`bl`, `bc *`	lista pułapek; usunięcie wszystkich
`g`	kontynuuj wykonywanie
`t`, `s`	jeden rozkaz: z wejściem do podprogramu / bez
`gr`	wykonuj do powrotu z podprogramu
`r`	pokaż rejestry; `r a 10` wpisuje wartość do rejestru
`u adres`	deasembluj kod od adresu
`d adres`	pokaż zawartość pamięci
`e adres bajty`	wpisz bajty do pamięci
`.help`	lista wszystkich poleceń

29.1.3 Technika pracy

1. **Postaw pułapkę na początku podejrzanej procedury** (`bp nazwa`) i przejdź ją krok po kroku, porównując rejestry z tym, czego oczekujesz.

2. **Gdy zmienna ma złą wartość**, a nie wiesz, kto ją zmienił — `ba w adres`. Debugger zatrzyma się dokładnie na winnym rozkazie.
3. **Gdy program „odlatuje”** (skacze w przypadkowe miejsce), prawie zawsze winny jest stos. Postaw pułapkę na `rts` podejrzanej procedury i sprawdź, czy wskaźnik stosu S ma tę samą wartość co przy wejściu.
4. **Rozkaz `brk` w kodzie** działa jak pułapka wpisana na stałe: Altirra zatrzyma się na nim sama.

29.1.4 Debugowanie bez debuggera

Na prawdziwym sprzęcie pomagają proste sztuczki:

```
        lda #$34                ; "doszedłem do tego miejsca": zmień kolor ramki na czerwony
        sta $D01A               ; zapis wprost do sprzętu - widoczny natychmiast

        lda zmienna             ; "jaka jest wartość zmiennej": pokaż ją jako znak
        ldy #0                  ; w lewym górnym rogu ekranu
        sta ($58),y             ; zapisz A pod adres ($58) + Y
```

Zmiana koloru ramki na wejściu i wyjściu z procedury pokazuje też **ile czasu ona zajmuje**: wysokość kolorowego pasa na ekranie odpowiada liczbie linii obrazu, przez które procedura pracowała.

29.2 Lista kontrolna: najczęstsze błędy

Gdy program nie działa, przejdź tę listę. W dziewięciu przypadkach na dziesięć błąd jest jednym z poniższych.

Adresowanie

- Brak `#` przed stałą: `lda 10` zamiast `lda #10`.
- Zbędny `#` przed zmienną: `lda #licznik` ładuje *adres* (i to tylko jego młodszy bajt), a nie zawartość.
- Zapomniane `ldy #0` przed `lda (wsk),y`.
- Wskaźnik umieszczony poza stroną zerową.
- Pomylone `<` i `>` przy ładowaniu adresu albo obie połówki wpisane do tej samej komórki.
- Nawiasy okrągłe zamiast kwadratowych w wyrażeniu: `lda (tab+1),y` to adresowanie pośrednie!

Znaczniki

- Brak `clc` przed `adc` lub `sec` przed `sbc`.
- Rozkaz wstawiony między porównanie a skok, który zmienia znaczniki (`lda`, `ldx`, `inx`, `dey`…).
- Zakładanie, że `inc`/`dec` zmieniają znacznik C.
- Zakładanie, że `sta` ustawia znaczniki.
- `bcc`/`bcs` użyte do liczb ze znakiem (trzeba `bmi`/`bpl` z poprawką na V).
- Pomylone `bcc` z `bcs` — po `cmp`: `bcs` to „większe lub równe”.
- Pętla z `bpl`, której licznik zaczyna się powyżej 128.

Stos i podprogramy

- `pha` bez `pla` na jednej ze ścieżek (zwłaszcza przy wcześniejszym `rts`).
- Skok `jmp` do podprogramu zamiast `jsr` — albo odwrotnie.
- Podprogram niszczy rejestr, którego wywołujący nadal używa.
- Wyjście z podprogramu skokiem `jmp` w inne miejsce, z adresem powrotu pozostawionym na stosie.

Arytmetyka wielobajtowa

- Zła kolejność: odejmowanie lub dodawanie od starszego bajtu.
- Drugie `clc` w środku dodawania wielobajtowego.
- Zmniejszanie liczby 16-bitowej bez sprawdzenia młodszego bajtu **przed** `dec`.
- Pozostawiony tryb dziesiętny (brak `cld`).

Pamięć

- Zmienna i kod pod tym samym adresem; dwa bufory nachodzące na siebie.
- Bufor za mały o jeden bajt (zapomniany znak EOL).
- Dane umieszczone „na drodze" wykonywania, bez `rts` lub `jmp` przed nimi.
- Zmienne strony zerowej współdzielone przez program główny i przerwanie.
- Kod lub dane w oknie `$4000–$7FFF` przy pracy z bankami.

Atari

- Kod ATASCII wpisany do pamięci ekranu (albo kod ekranowy wysłany przez CIO).
- Zapis do rejestru sprzętowego zamiast do cienia — efekt znika po 1/50 sekundy.
- Zapis do cienia w procedurze DLI — efekt pojawia się w złym miejscu.
- Procedura VBI zakończona `rts` zamiast `jmp XITVBV`; procedura DLI bez zachowania rejestrów.
- Niezamknięty plik otwarty do zapisu.
- Brak segmentu z adresem uruchomienia — program się ładuje i „nic się nie dzieje".
- Program nie przywraca wektorów przerwań przed powrotem do DOS-u.

29.3 Optymalizacja szybkości

Najpierw zasada nadrzędna: **optymalizuj tylko to, co wykonuje się często.** Procedura inicjująca wywoływana raz może być napisana dowolnie rozrzutnie; liczy się wewnętrzna pętla, która wykonuje się tysiące razy na sekundę.

Używaj strony zerowej. Każdy dostęp jest o cykl krótszy i o bajt krótszy. Zmienne z najgorętszych pętli powinny tam trafić w pierwszej kolejności.

Licz w dół do zera. `dex / bne` nie potrzebuje porównania. Jeśli kolejność nie ma znaczenia, pętla malejąca jest zawsze szybsza.

Zastępuj obliczenia tablicami. Mnożenie przez stałą, adresy wierszy ekranu, maski bitów, zamiana kodów — odczyt z tablicy to 4–5 cykli.

Wyciągaj obliczenia przed pętlę. Wszystko, co nie zmienia się między obiegami, oblicz raz.

Rozwijaj krótkie pętle. Pętla kopiująca 4 bajty to 4 obiegi po 14 cykli; cztery pary `lda/sta` wypisane wprost zajmują po 8 cykli — prawie dwa razy mniej.

```
; wersja z pętlą: 2 cykle inicjalizacji + 4 obiegi po 14 cykli
        ldx #3                  ; X = 3
kop     lda zrodlo,x            ; A = bajt spod adresu zrodlo+X
        sta cel,x               ; zapisz A pod adres cel+X
        dex                     ; X = X - 1
        bpl kop                 ; skocz do kop, gdy N = 0 (bit 7 skasowany)
```

```
        lda zrodlo              ; to samo bez pętli: 4 x 8 cykli; dłuższe, ale szybsze
        sta cel                 ; zapisz A do cel
        lda zrodlo+1            ; A = bajt spod adresu zrodlo+1
        sta cel+1               ; zapisz A pod adres cel+1
        lda zrodlo+2            ; A = bajt spod adresu zrodlo+2
        sta cel+2               ; zapisz A pod adres cel+2
        lda zrodlo+3            ; A = bajt spod adresu zrodlo+3
        sta cel+3               ; zapisz A pod adres cel+3
```

Zastępuj `jsr` + `rts` skokiem `jmp` na końcu podprogramu (rozdział 11).

Korzystaj ze znanych wartości. Jeśli po pętli wiadomo, że X = 0, nie ładuj zera od nowa — użyj `txa`. Jeśli po skoku `bcc` wiadomo, że C = 1, pomiń `sec`.

Unikaj `(wsk),y` w najgorętszych pętlach, jeśli adres jest stały lub zmienia się rzadko — kod samomodyfikujący się z trybem `adres,x` jest szybszy (rozdział 18).

Ustaw najczęstszy przypadek „na wprost". Skok warunkowy, który się nie wykonuje, trwa 2 cykle; wykonany — 3. Układaj warunki tak, by typowa ścieżka przechodziła bez skoków.

29.4 Optymalizacja rozmiaru

Wspólne zakończenia. Kilka procedur kończących się tymi samymi rozkazami może skakać do jednego wspólnego fragmentu.

Przepadanie. Procedura, która kończy się wywołaniem innej, może po prostu leżeć tuż przed nią i „wpaść" w nią bez żadnego skoku:

```
nowa_linia lda #$9B             ; A = znak końca linii...
                                ; ...i wprost "wpadamy" w procedurę putchar poniżej
putchar ldx #0                  ; X = 0
        rts                     ; powrót z podprogramu
```

Skoki warunkowe zamiast `jmp`. Gdy stan znacznika jest pewny, `bne` czy `bcc` jako skok „zawsze" oszczędza bajt.

Tablice zamiast kodu. Dziesięć podobnych wywołań z różnymi parametrami można zastąpić pętlą po tablicy parametrów.

Inicjalizacja pętlą. Zamiast dwudziestu par `lda # / sta` — tablica wartości początkowych kopiowana jedną pętlą.

Pamiętaj, że szybkość i rozmiar zwykle stoją w sprzeczności: tablice i rozwinięte pętle przyspieszają, ale zajmują miejsce.

29.5 Liczenie cykli

W efektach graficznych i procedurach przerwań trzeba wiedzieć, ile czasu zajmuje kod. Liczby cykli podaje leksykon w rozdziale 6. Punkty odniesienia dla systemu PAL:

Odcinek czasu	Cykli procesora
jedna linia obrazu	114
jedna linia trybu tekstowego — czas pozostawiony procesorowi	ok. 60–70 (pierwsza linia wiersza: ok. 25–30)
jedna ramka (312 linii)	35 568
okres wygaszania pionowego (64 linie)	ok. 7 300
sekunda	ok. 1 773 000

Przykład: pętla kopiująca z rozdziału 18 zajmuje 16 cykli na bajt. Przepisanie całego ekranu tekstowego (960 bajtów) to około 15 400 cykli — przy wyświetlanym obrazie mniej więcej pół ramki. Przewijanie ekranu w każdej ramce jest więc możliwe, ale zabiera połowę mocy procesora.

29.6 Nieudokumentowane rozkazy

Z 256 możliwych kodów rozkazów 6502 oficjalnie wykorzystuje 151. Pozostałe też „coś robią" — to efekt uboczny konstrukcji układu. Niektóre z nich są stabilne i użyteczne, bo łączą dwie operacje w jednym rozkazie. MADS zna ich mnemoniki:

Mnemonik	Działanie	Przykładowe zastosowanie
`lax`	załaduj jednocześnie A i X	oszczędza `tax`
`sax`	zapisz do pamięci wynik A AND X	maskowanie bez zmiany A
`dcp`	zmniejsz komórkę, potem porównaj z A	licznik z porównaniem
`isb`	zwiększ komórkę, potem odejmij od A	
`anc`	AND, a potem bit 7 wyniku do C	
`alr`	AND, a potem przesunięcie w prawo	
`sbx`	X := (A AND X) − argument	odejmowanie od X bez użycia A

W demach i grach, gdzie liczy się każdy cykl, używa się ich powszechnie — procesor Atari wykonuje je poprawnie. W zwykłych programach lepiej ich unikać: utrudniają czytanie kodu, nie działają na procesorze 65C02 (spotykanym w niektórych rozszerzeniach i akceleratorach) i nie są potrzebne do nauki. W tej książce nie pojawiają się nigdzie poza tym akapitem.

29.7 Styl i organizacja programu

- **Nazywaj wszystko.** `sta COLOR2` zamiast `sta $02C6`; `cmp #KLAW_ESC` zamiast `cmp #$1C`.
- **Jedna procedura — jedno zadanie.** Jeśli nie umiesz opisać procedury jednym zdaniem, podziel ją.
- **Nagłówek z umową** przy każdej procedurze: wejście, wyjście, niszczone rejestry.
- **Komentuj zamiar, nie rozkaz.** W podręczniku komentarz `; X = X - 1` przy `dex` pomaga w nauce; we własnym programie niczego nie wnosi. `; następny wróg` — to jest informacja, której nie widać w samym rozkazie.
- **Zmienne strony zerowej deklaruj w jednym miejscu**, z adresami liczonymi względem poprzedniej zmiennej.
- **Dziel program na pliki** (`icl`): stałe systemowe, biblioteka, logika, dane.
- **Testuj procedury osobno.** Mnożenie sprawdź na kilku znanych przykładach, zanim użyjesz go w grze.
- **Zapisuj działające wersje.** Gdy coś przestaje działać, najszybciej znaleźć błąd, porównując z ostatnią dobrą wersją.

29.8 Co dalej

Po przerobieniu tej książki znasz komplet rozkazów procesora, wszystkie podstawowe techniki programowania i najważniejsze mechanizmy komputera Atari. Naturalne kolejne kroki:

- **Płynny przesuw ekranu** (rejestry HSCROL i VSCROL) — podstawa gier z przewijanym tłem.
- **Odtwarzacze muzyki** (RMT, CMC, MPT) wywoływane z przerwania VBI.
- **Tryby GTIA** i techniki zwiększania liczby kolorów.
- **Kompresja danych** i ładowanie etapami.
- **Makra i struktury MADS** — teraz, gdy wiesz, co kryje się pod każdym pseudorozkazem, możesz po nie sięgnąć świadomie, jako po skrót, a nie magię.

Źródła, do których warto zaglądać: dokumentacja MAD Assemblera (`mads.atari8.info`), klasyczne opracowania *Mapping the Atari* i *De Re Atari*, dokumentacja techniczna emulatora Altirra (*Altirra Hardware Reference Manual*) — najdokładniejszy istniejący opis sprzętu — oraz polskie serwisy i fora atarowskie (AtariOnline.pl, Atari Area, Atariki).

Najważniejsze jednak to pisać własne programy. Zacznij od małych: zegar, edytor znaków, prosta gra zręcznościowa. Każdy z nich nauczy cię więcej niż kolejny przeczytany rozdział.

Cześć VI

Procedury w MAD Assemblerze

ROZDZIAŁ 30

Procedury `.proc`: pisanie i używanie

W rozdziale 11 podprogram był po prostu etykietą, do której skaczemy rozkazem `jsr`. To w zupełności wystarcza procesorowi — ale w programie liczącym kilka tysięcy linii zaczyna brakować porządku: nazwy etykiet się powtarzają, nie widać, gdzie procedura się kończy, a nieużywany kod zajmuje pamięć. MAD Assembler oferuje na to dyrektywę **`.proc`**.

Najważniejsza rzecz na początek:

> **Zapamiętaj**
>
> **`.proc` nie generuje żadnego kodu.** To dyrektywa porządkująca tekst programu: wyznacza początek i koniec procedury, nadaje jej nazwę i zamyka jej etykiety we własnym zasięgu. Procesor nadal dostaje dokładnie te rozkazy, które napisałeś — `jsr` do wywołania i `rts` do powrotu. Procedury `.proc` pozostają więc w zgodzie z zasadą tej książki: czyste mnemoniki, nic ukrytego.

30.1 Najprostsza procedura

```
        .proc czysc_strone      ; początek procedury; jej nazwa to etykieta pierwszego rozkazu
        lda #0                  ; A = 0: wartość wypełniająca
        tax                     ; X = 0: indeks
petla   sta $0600,x             ; etykieta "petla" istnieje tylko wewnątrz tej procedury
        inx                     ; następny bajt
        bne petla               ; 256 obiegów
        rts                     ; powrót - RTS trzeba napisać samodzielnie!
        .endp                   ; koniec procedury (to NIE jest rozkaz powrotu)
```

Wywołanie niczym nie różni się od wywołania zwykłego podprogramu:

```
        jsr czysc_strone        ; nazwa procedury to zwykła etykieta: adres jej początku
```

> **Pułapka**
>
> **`.endp` to nie `rts`.** Dyrektywa `.endp` tylko zamyka blok w tekście źródłowym. Jeśli zapomnisz o rozkazie `rts`, procesor po ostatnim rozkazie procedury pójdzie dalej — w to, co akurat leży w pamięci za nią.

30.2 Etykiety lokalne

Wszystkie etykiety zdefiniowane między `.proc` a `.endp` są **lokalne**: nie kolidują z etykietami o tych samych nazwach w innych procedurach ani w programie głównym. Koniec z wymyślaniem nazw `petla1`, `petla2`, `petla_kopiuj`...

```
        .proc zeruj_bufor       ; pierwsza procedura
        ldx #39                 ; X = indeks ostatniego bajtu
petla   sta bufor,x             ; "petla" należy do procedury zeruj_bufor
        dex                     ; X = X - 1
        bpl petla               ; skocz do petla, gdy N = 0 (bit 7 skasowany)
        rts                     ; powrót z podprogramu
        .endp                   ; koniec procedury

        .proc kopiuj_wiersz     ; druga procedura
        ldx #39                 ; X = 39
petla   lda zrodlo,x            ; inna "petla" - bez żadnego konfliktu z poprzednią
        sta cel,x               ; zapisz A pod adres cel+X
        dex                     ; X = X - 1
        bpl petla               ; skocz do petla, gdy N = 0 (bit 7 skasowany)
        rts                     ; powrót z podprogramu
```

```
        .endp                   ; koniec procedury
```

30.2.1 Dostęp do etykiet z zewnątrz

Do etykiety wewnętrznej można sięgnąć spoza procedury, poprzedzając ją nazwą procedury i kropką. Asembler zapisuje takie nazwy również w pliku `.lab`, więc zobaczysz je w debuggerze:

```
        jsr kopiuj_wiersz       ; wywołaj procedurę
        jsr kopiuj_wiersz.petla  ; skocz w środek procedury, do jej etykiety "petla"
        lda druk.licznik        ; odczytaj zmienną lokalną procedury "druk"
```

Wchodzenie w środek cudzej procedury jest zwykle złym pomysłem, ale dostęp do **danych** procedury bywa bardzo przydatny — na przykład do ustawienia jej parametrów.

30.2.2 Dostęp do etykiet globalnych

Wewnątrz procedury widać wszystkie etykiety globalne. Jeżeli jednak procedura ma własną etykietę o tej samej nazwie, pierwszeństwo ma lokalna. Do globalnej sięgamy wtedy z dwukropkiem na początku:

```
licznik dta 0                   ; zmienna GLOBALNA programu

        .proc zlicz
        inc licznik             ; zwiększa zmienną LOKALNĄ (zdefiniowaną niżej)
        inc :licznik            ; dwukropek: zwiększa zmienną GLOBALNĄ
        rts                     ; powrót z podprogramu
licznik dta 0                   ; zmienna LOKALNA procedury, o tej samej nazwie
        .endp                   ; koniec procedury
```

W praktyce lepiej unikać takich powtórzeń; dwukropek przydaje się głównie w plikach bibliotecznych, których autor nie wie, jakie nazwy wystąpią w programie.

30.3 Zmienne lokalne procedury

Skoro etykiety są lokalne, procedura może mieć **własne zmienne**, niewidoczne (bez przedrostka) dla reszty programu. Umieszczamy je za rozkazem `rts`, żeby nie leżały na drodze wykonywania:

```
        .proc srednia           ; A := (pierwsza + druga) / 2
        lda pierwsza            ; A = pierwsza liczba
        clc                     ; C = 0 (przed dodawaniem)
        adc druga               ; A = suma (dziewiąty bit w C)
        ror @                   ; podziel przez 2, wsuwając C z lewej strony
        rts                     ; powrót z podprogramu
pierwsza dta 0                  ; zmienne lokalne - parametry procedury
druga   dta 0                   ; dane: bajty
        .endp                   ; koniec procedury

; --- wywołanie: wpisz parametry do zmiennych procedury i wywołaj ją ---
        lda #100                ; A = 100
        sta srednia.pierwsza    ; parametr 1
        lda #50                 ; A = 50
        sta srednia.druga       ; parametr 2
        jsr srednia             ; A = 75
```

To czytelny i w pełni „czysty” sposób przekazywania parametrów: nazwa `srednia.pierwsza` sama mówi, czyja to zmienna.

> **Uwaga**
>
> Zmienne lokalne `.proc` to nadal zwykłe, stałe komórki pamięci — nie leżą na stosie. Procedura z takimi zmiennymi **nie może wywoływać samej siebie** i nie powinna być używana jednocześnie przez program główny i przerwanie. Wskaźniki do adresowania `(wsk),y` muszą pozostać na stronie zerowej — zadeklaruj je zwykłym przypisaniem, np. `wsk = $80`, także wewnątrz procedury.

30.4 Procedury zagnieżdżone

Procedura może zawierać inne procedury. Wewnętrzne są jej prywatnymi pomocnikami; z zewnątrz dostępne są pod nazwą złożoną:

```
        .proc ekran             ; "moduł" grupujący procedury ekranowe

        .proc czysc             ; pełna nazwa: ekran.czysc
        lda #0                  ; A = 0
        rts                     ; powrót z podprogramu
        .endp                   ; koniec procedury

        .proc ramka             ; pełna nazwa: ekran.ramka
        jsr czysc               ; z wnętrza modułu wystarczy krótka nazwa
        rts                     ; powrót z podprogramu
        .endp                   ; koniec procedury

        .endp                   ; koniec procedury

        jsr ekran.czysc         ; z zewnątrz: nazwa modułu, kropka, nazwa procedury
        jsr ekran.ramka         ; wywołaj podprogram ekran.ramka
```

Taki zewnętrzny blok bez własnego kodu działa jak **przestrzeń nazw**: `ekran.czysc`, `dysk.czytaj`, `dzwiek.graj`. Duże programy bardzo na tym zyskują.

30.5 Pomijanie nieużywanych procedur

To największa praktyczna zaleta `.proc`. Asembler wie, gdzie procedura się zaczyna i kończy, więc potrafi sprawdzić, czy ktokolwiek się do niej odwołuje — i **nie umieszczać w pliku wynikowym procedur nieużywanych**. Włącza to opcja wiersza poleceń `-x`:

```
mads program.asm -o:program.xex -x
```

Dzięki temu można zbudować własną bibliotekę kilkudziesięciu procedur w jednym pliku, dołączać ją dyrektywą `icl` do każdego programu — a do pamięci trafią tylko te procedury, które dany program faktycznie wywołuje.

Opcja	Działanie
`-x`	pomiń procedury `.proc`, do których nie ma odwołań
`-xp`	wypisz ostrzeżenia o procedurach, do których nie ma odwołań

Pułapka

Asembler rozpoznaje użycie procedury po odwołaniu do jej nazwy w tekście programu. Jeśli jedynym „odwołaniem" jest adres wyliczony w trakcie działania programu albo wpisany jako liczba, procedura zostanie przy opcji `-x` pominięta. Adresy w tablicach skoków (`dta a(proc0,proc1)`) są zwykłymi odwołaniami i działają poprawnie.

30.6 Kod przeznaczony dla innego adresu

Czasem procedura ma działać w innym miejscu pamięci niż to, w którym leży w pliku: procedura przenoszona na stronę 6, kod kopiowany pod ROM, sektor startowy osadzony w programie. Drugi argument dyrektywy podaje **adres, dla którego kod ma być asemblowany**:

```
obraz                           ; etykieta miejsca, w którym kod FAKTYCZNIE leży w programie
        .proc pomocnik,$0600    ; kod asemblowany tak, jakby leżał pod adresem $0600
        lda dane                ; adres "dane" jest liczony względem $0600
        jmp dalej               ; ten skok również
dalej   rts                     ; powrót z podprogramu
dane    dta 7                   ; dane: bajty
```

```
        .endp                   ; koniec procedury
DLUG    = *-obraz               ; długość kodu do skopiowania
```

Etykieta `pomocnik` ma wartość `$0600`, natomiast `obraz` wskazuje miejsce, skąd kod trzeba najpierw **skopiować** pod `$0600`. Dopiero potem wolno wykonać `jsr pomocnik`. Po dyrektywie `.endp` asembler wraca do poprzedniego toku adresów.

30.7 Długość procedury

Operator `.len` zwraca długość procedury w bajtach — przydatne przy kopiowaniu i do kontroli rozmiaru:

```
        lda #.len czysc_strone  ; A = liczba bajtów zajmowanych przez procedurę
        .print "Procedura czysc_strone: ",.len czysc_strone," bajtow"
```

30.8 Program przykładowy

Program zbiera omówione elementy: moduł z procedurami zagnieżdżonymi, etykiety i zmienne lokalne, dostęp przez nazwę z kropką oraz procedurę przenoszoną na stronę 6.

Plik: procedury.asm

```
; ------------------------------------------------------------------------
; procedury.asm - pisanie i używanie procedur .proc (same czyste rozkazy 6502)
; Asemblacja:  mads procedury.asm -o:procedury.xex -l -t -x
; ------------------------------------------------------------------------
ICCOM   = $0342                 ; pola bloku IOCB numer 0
ICBAL   = $0344                 ; adres bufora: młodszy bajt
ICBAH   = $0345                 ; adres bufora: starszy bajt
ICBLL   = $0348                 ; długość bufora: młodszy bajt
ICBLH   = $0349                 ; długość bufora: starszy bajt
CIOV    = $E456                 ; procedura wejścia/wyjścia systemu
CH      = $02FC                 ; kod klawisza
EOL     = $9B                   ; znak końca linii

        org $2000               ; adres asemblacji: $2000

start   lda #<tekst1            ; A/Y = adres pierwszego napisu
        ldy #>tekst1            ; Y = starszy bajt adresu tekst1
        jsr druk.napis          ; procedura "napis" z modułu "druk"

        lda #3                  ; parametry przez zmienne lokalne procedury:
        sta gwiazdki.ile        ; gwiazdki.ile := 3
        jsr gwiazdki            ; wypisz trzy gwiazdki
        lda #10                 ; A = 10
        sta gwiazdki.ile        ; gwiazdki.ile := 10
        jsr gwiazdki            ; wypisz dziesięć gwiazdek
        jsr druk.nowy_wiersz    ; wywołaj podprogram druk.nowy_wiersz

        jsr przenies            ; skopiuj procedurę "na_stronie6" pod adres $0600
        jsr na_stronie6         ; i wywołaj ją tam (wypisze znak "!")
        jsr druk.nowy_wiersz    ; wywołaj podprogram druk.nowy_wiersz

        lda druk.licznik        ; zmienna lokalna modułu: ile znaków wypisano
        sta $0680               ; zapamiętaj ją (np. do obejrzenia w debuggerze)
        jsr czekaj              ; poczekaj na klawisz
        rts                     ; powrót do DOS-u

; === moduł "druk": procedury wypisywania na ekran ============================
        .proc druk              ; zewnętrzna procedura pełni rolę przestrzeni nazw

        .proc znak              ; druk.znak - wypisuje znak z A; niszczy A, X, Y
        inc licznik             ; policz znak (zmienna modułu "druk" - patrz niżej)
        ldx #$00                ; kanał 0: edytor ekranowy
        pha                     ; zachowaj znak
        lda #11                 ; PUT CHARACTERS
        sta ICCOM,x             ; zapisz A pod adres ICCOM+X - rozkaz
        lda #0                  ; długość 0: znak z akumulatora
```

```
        sta ICBLL,x             ; zapisz A pod adres ICBLL+X - długość bufora: młodszy bajt
        sta ICBLH,x             ; zapisz A pod adres ICBLH+X - długość bufora: starszy bajt
        pla                     ; odtwórz znak
        jmp CIOV                ; wypisz i wróć do wywołującego
        .endp                   ; koniec procedury

        .proc napis             ; druk.napis - wypisuje wiersz spod adresu A/Y
        ldx #$00                ; kanał 0
        sta ICBAL,x             ; młodszy bajt adresu
        tya                     ; A = Y
        sta ICBAH,x             ; starszy bajt adresu
        lda #9                  ; PUT RECORD
        sta ICCOM,x             ; zapisz A pod adres ICCOM+X - rozkaz
        lda #255                ; najwyżej 255 znaków
        sta ICBLL,x             ; zapisz A pod adres ICBLL+X - długość bufora: młodszy bajt
        lda #0                  ; A = 0
        sta ICBLH,x             ; zapisz A pod adres ICBLH+X - długość bufora: starszy bajt
        jmp CIOV                ; skocz do CIOV - centralna procedura wejścia/wyjścia
        .endp                   ; koniec procedury

        .proc nowy_wiersz       ; druk.nowy_wiersz - przejście do nowej linii
        lda #EOL                ; A = znak końca linii
        jmp znak                ; wewnątrz modułu wystarcza krótka nazwa "znak"
        .endp                   ; koniec procedury

licznik dta 0                   ; zmienna modułu: druk.licznik
        .endp                   ; koniec modułu "druk"

; === gwiazdki: wypisuje "ile" gwiazdek ======================================
        .proc gwiazdki
        lda ile                 ; czy jest cokolwiek do wypisania?
        beq koniec              ; nie: od razu wróć
        sta zostalo             ; licznik roboczy := ile
petla   lda #'*'                ; A = znak gwiazdki
        jsr druk.znak           ; wypisz (procedura z innego modułu: pełna nazwa)
        dec zostalo             ; jedna mniej
        bne petla               ; etykiety "petla" i "koniec" są lokalne
koniec  rts                     ; powrót z podprogramu
ile     dta 0                   ; parametr: liczba gwiazdek (gwiazdki.ile)
zostalo dta 0                   ; zmienna robocza
        .endp                   ; koniec procedury

; === czekaj: czeka na dowolny klawisz =======================================
        .proc czekaj
        lda #$FF                ; skasuj poprzedni klawisz
        sta CH                  ; zapisz A do CH - kod klawisza
petla   lda CH                  ; ta "petla" nie koliduje z etykietą w "gwiazdki"
        cmp #$FF                ; porównaj A z: $FF
        beq petla               ; czekaj, aż coś zostanie naciśnięte
        lda #$FF                ; skasuj klawisz
        sta CH                  ; zapisz A do CH - kod klawisza
        rts                     ; powrót z podprogramu
        .endp                   ; koniec procedury

; === przenies: kopiuje kod procedury "na_stronie6" na stronę 6 ==============
        .proc przenies
        ldx #DLUG-1             ; X = indeks ostatniego bajtu kodu
petla   lda obraz,x             ; bajt z miejsca, w którym kod leży w programie
        sta na_stronie6,x       ; pod adres, dla którego został zasemblowany ($0600...)
        dex                     ; X = X - 1
        bpl petla               ; skocz do petla, gdy N = 0 (bit 7 skasowany)
        rts                     ; powrót z podprogramu
        .endp                   ; koniec procedury

; === procedura asemblowana dla adresu $0600, a leżąca tutaj ==================
obraz                           ; stąd trzeba skopiować kod
        .proc na_stronie6,$0600  ; kod liczony dla adresu $0600
        lda wykrzyknik          ; adres zmiennej liczony względem $0600
        jmp druk.znak           ; wypisz znak; skok poza stronę 6 jest dozwolony
wykrzyknik dta c'!'             ; dana lokalna procedury
        .endp                   ; koniec procedury
DLUG    = *-obraz               ; długość kodu do skopiowania
```

```
tekst1  dta c'Procedury .proc w akcji:',EOL  ; tekst w kodach ATASCII

        org $02E0               ; wektor RUNAD
        dta a(start)            ; adres uruchomienia programu
```

Zajrzyj do pliku `procedury.lab` po asemblacji: znajdziesz w nim nazwy `DRUK.ZNAK`, `GWIAZDKI.ILE`, `CZEKAJ.PETLA`, `NA_STRONIE6.WYKRZYKNIK` — każda etykieta lokalna ma tam pełną nazwę i adres.

30.9 Wywołanie z parametrami — skrót, którego nie używamy

Dla porządku trzeba wspomnieć o rozszerzeniu, które spotkasz w cudzych źródłach. MADS pozwala zadeklarować w nagłówku procedury **listę parametrów**, a potem wywołać ją samą nazwą, bez `jsr`. Asembler sam dopisuje wtedy rozkazy ładujące parametry. To już nie jest czysty kod — jedna linia źródła zamienia się w kilka rozkazów — dlatego w tej książce z tego nie korzystamy. Warto jednak wiedzieć, co się za tym kryje. Poniższe rozwinięcia odczytano z listingu MADS 2.1.7.

Parametry w rejestrach (`.reg`) — nazwy parametrów to nazwy rejestrów:

```
        .proc ustaw (.byte x,a) .reg  ; dwa parametry bajtowe: pierwszy w X, drugi w A
        sta $02C6,x             ; zwykły kod procedury
        rts                     ; powrót z podprogramu
        .endp                   ; koniec procedury

        ustaw #5 #$34           ; wywołanie skrótowe; asembler generuje:
;       ldx #5                          ;   pierwszy parametr do rejestru X
;       lda #$34                        ;   drugi parametr do rejestru A
;       jsr ustaw                       ;   wywołanie procedury
```

Parametr dwubajtowy podaje się parą rejestrów: `(.word ya)` oznacza starszy bajt w Y, młodszy w A. Wywołanie `druk #tekst` rozwija się wtedy w `ldy #>tekst` / `lda #<tekst` / `jsr druk`.

Parametry w zmiennych (`.var`) — parametry trafiają do zmiennych lokalnych procedury o podanych nazwach:

```
        .proc suma (.byte p1,p2) .var  ; dwa parametry bajtowe w zmiennych p1 i p2
        lda p1                  ; A = p1
        clc                     ; C = 0 (przed dodawaniem)
        adc p2                  ; A = A + p2 + C
        rts                     ; powrót z podprogramu
p1      dta 0                   ; zmienne trzeba zadeklarować samodzielnie
p2      dta 0                   ; dane: bajty
        .endp                   ; koniec procedury

        suma liczba #3          ; wywołanie skrótowe; asembler generuje:
;       lda liczba                      ;   pierwszy parametr...
;       sta suma.p1                     ;   ...do zmiennej p1 procedury
;       lda #3                          ;   drugi parametr...
;       sta suma.p2                     ;   ...do zmiennej p2
;       jsr suma                        ;   wywołanie procedury
```

Zauważ, że wersja `.var` **niszczy akumulator** przy przekazywaniu parametrów, choć w linii wywołania tego nie widać. To dobry przykład, dlaczego warto znać rozwinięcia: skrót jest wygodny, ale kto nie wie, co się pod nim kryje, prędzej czy później straci na tym godzinę przy szukaniu błędu.

Procedurę zadeklarowaną z listą parametrów nadal można wywoływać „ręcznie” — zwykłym `jsr`, po samodzielnym załadowaniu rejestrów lub zmiennych. Deklaracja parametrów służy wtedy jako dokumentacja.

30.10 Zwykła etykieta, `.local` czy `.proc`?

Cecha	zwykła etykieta	`.local`	`.proc`
generuje dodatkowy kod	nie	nie	nie
etykiety lokalne	nie	tak	tak
dostęp `nazwa.etykieta`	—	tak	tak
widoczny początek i koniec	nie	tak	tak

Cecha	zwykła etykieta	`.local`	`.proc`
pomijanie nieużywanego kodu (`-x`)	nie	nie	**tak**
długość przez `.len`	nie	tak	tak
asemblacja dla innego adresu	—	tak	tak
deklaracja parametrów	nie	nie	tak

Praktyczna rada:

- **`.proc`** — dla wszystkiego, co wywołujesz przez `jsr`: każda procedura programu i każda procedura biblioteczna.
- **`.local`** — dla bloków, które nie są procedurami: obszar danych, fragment programu głównego, kod wstawiany w tok wykonywania.
- **zwykłe etykiety** — wewnątrz procedur oraz w bardzo krótkich programach.

30.11 Dobre obyczaje

1. **Nagłówek z umową** pozostaje obowiązkowy: co procedura dostaje, co zwraca, które rejestry niszczy. Dyrektywa `.proc` porządkuje nazwy, ale nie dokumentuje działania.
2. **Jedno wyjście.** Staraj się, by procedura miała jeden rozkaz `rts` na końcu (ewentualnie skoki do wspólnej etykiety `koniec`) — łatwiej wtedy dopilnować stosu.
3. **Dane za kodem.** Zmienne lokalne i tablice umieszczaj po ostatnim `rts` lub `jmp`.
4. **Nie skacz do środka cudzej procedury.** Jeśli potrzebujesz drugiego punktu wejścia, zrób z niego osobną procedurę.
5. **Grupuj procedury w moduły** (procedura zewnętrzna jako przestrzeń nazw) i trzymaj każdy moduł w osobnym pliku dołączanym przez `icl`.
6. **Asembluj z opcją `-x`**, gdy korzystasz z biblioteki — i z `-xp`, gdy chcesz sprawdzić, czy w programie nie został martwy kod.

Ćwiczenia

1. Przepisz procedurę `mul8f` z rozdziału 14 na postać `.proc`, ze zmiennymi `mnozna` i `mnoznik` jako danymi lokalnymi. Jak będzie wyglądało jej wywołanie?
2. Zbierz procedury `memcpy`, `memset` i `memcmp` z rozdziału 18 w moduł `pamiec` (wywołania: `jsr pamiec.kopiuj` itd.). Wskaźniki zostaw na stronie zerowej.
3. Zasembluj `procedury.asm` z opcją `-x` i bez niej, dopisawszy wcześniej procedurę, której nikt nie wywołuje. Porównaj długość pliku wynikowego.

Odpowiedź do 1: `lda #12` / `sta mul8f.mnozna` / `lda #10` / `sta mul8f.mnoznik` / `jsr mul8f`.

Cześć VII

Dodatki

DODATEK A

Tabele kodów rozkazów

A.1 Rozkazy według mnemoników

Tabela podaje kod maszynowy każdego rozkazu w każdym dostępnym trybie adresowania. Kropka oznacza, że rozkaz w danym trybie nie istnieje. Nagłówki kolumn: `---` implikowany, `A` akumulator, `#` natychmiastowy, `zp` strona zerowa, `abs` absolutny, `(abs)` pośredni, `rel` względny.

	---	A	#	zp	zp,X	zp,Y	abs	abs,X	abs,Y	(abs)	(zp,X)	(zp),Y	rel
ADC	.	.	$69	$65	$75	.	$6D	$7D	$79	.	$61	$71	.
AND	.	.	$29	$25	$35	.	$2D	$3D	$39	.	$21	$31	.
ASL	.	$0A	.	$06	$16	.	$0E	$1E	.	.	.	.	.
BCC	.	.	.	.	.	.	.	.	.	.	.	.	$90
BCS	.	.	.	.	.	.	.	.	.	.	.	.	$B0
BEQ	.	.	.	.	.	.	.	.	.	.	.	.	$F0
BIT	.	.	.	$24	.	.	$2C	.	.	.	.	.	.
BMI	.	.	.	.	.	.	.	.	.	.	.	.	$30
BNE	.	.	.	.	.	.	.	.	.	.	.	.	$D0
BPL	.	.	.	.	.	.	.	.	.	.	.	.	$10
BRK	$00	.	.	.	.	.	.	.	.	.	.	.	.
BVC	.	.	.	.	.	.	.	.	.	.	.	.	$50
BVS	.	.	.	.	.	.	.	.	.	.	.	.	$70
CLC	$18	.	.	.	.	.	.	.	.	.	.	.	.
CLD	$D8	.	.	.	.	.	.	.	.	.	.	.	.
CLI	$58	.	.	.	.	.	.	.	.	.	.	.	.
CLV	$B8	.	.	.	.	.	.	.	.	.	.	.	.
CMP	.	.	$C9	$C5	$D5	.	$CD	$DD	$D9	.	$C1	$D1	.
CPX	.	.	$E0	$E4	.	.	$EC	.	.	.	.	.	.
CPY	.	.	$C0	$C4	.	.	$CC	.	.	.	.	.	.
DEC	.	.	.	$C6	$D6	.	$CE	$DE	.	.	.	.	.
DEX	$CA	.	.	.	.	.	.	.	.	.	.	.	.
DEY	$88	.	.	.	.	.	.	.	.	.	.	.	.
EOR	.	.	$49	$45	$55	.	$4D	$5D	$59	.	$41	$51	.
INC	.	.	.	$E6	$F6	.	$EE	$FE	.	.	.	.	.
INX	$E8	.	.	.	.	.	.	.	.	.	.	.	.
INY	$C8	.	.	.	.	.	.	.	.	.	.	.	.
JMP	.	.	.	.	.	.	$4C	.	.	$6C	.	.	.
JSR	.	.	.	.	.	.	$20	.	.	.	.	.	.
LDA	.	.	$A9	$A5	$B5	.	$AD	$BD	$B9	.	$A1	$B1	.
LDX	.	.	$A2	$A6	.	$B6	$AE	.	$BE	.	.	.	.
LDY	.	.	$A0	$A4	$B4	.	$AC	$BC	.	.	.	.	.
LSR	.	$4A	.	$46	$56	.	$4E	$5E	.	.	.	.	.
NOP	$EA	.	.	.	.	.	.	.	.	.	.	.	.
ORA	.	.	$09	$05	$15	.	$0D	$1D	$19	.	$01	$11	.
PHA	$48	.	.	.	.	.	.	.	.	.	.	.	.
PHP	$08	.	.	.	.	.	.	.	.	.	.	.	.
PLA	$68	.	.	.	.	.	.	.	.	.	.	.	.
PLP	$28	.	.	.	.	.	.	.	.	.	.	.	.
ROL	.	$2A	.	$26	$36	.	$2E	$3E	.	.	.	.	.
ROR	.	$6A	.	$66	$76	.	$6E	$7E	.	.	.	.	.
RTI	$40	.	.	.	.	.	.	.	.	.	.	.	.
RTS	$60	.	.	.	.	.	.	.	.	.	.	.	.
SBC	.	.	$E9	$E5	$F5	.	$ED	$FD	$F9	.	$E1	$F1	.
SEC	$38	.	.	.	.	.	.	.	.	.	.	.	.
SED	$F8	.	.	.	.	.	.	.	.	.	.	.	.
SEI	$78	.	.	.	.	.	.	.	.	.	.	.	.
STA	.	.	.	$85	$95	.	$8D	$9D	$99	.	$81	$91	.
STX	.	.	.	$86	.	$96	$8E	.	.	.	.	.	.
STY	.	.	.	$84	$94	.	$8C	.	.	.	.	.	.
TAX	$AA	.	.	.	.	.	.	.	.	.	.	.	.
TAY	$A8	.	.	.	.	.	.	.	.	.	.	.	.
TSX	$BA	.	.	.	.	.	.	.	.	.	.	.	.
TXA	$8A	.	.	.	.	.	.	.	.	.	.	.	.

```
TXS     $9A     .     .     .     .     .     .     .     .     .     .     .     .
TYA     $98     .     .     .     .     .     .     .     .     .     .     .     .
```

A.2 Rozkazy według kodów

Przydatne przy ręcznym czytaniu kodu maszynowego (np. zrzutu sektora startowego).

```
$00 BRK          $36 ROL zp,X     $6D ADC abs      $A4 LDY zp       $D0 BNE rel
$01 ORA (zp,X)   $38 SEC          $6E ROR abs      $A5 LDA zp       $D1 CMP (zp),Y
$05 ORA zp       $39 AND abs,Y    $70 BVS rel      $A6 LDX zp       $D5 CMP zp,X
$06 ASL zp       $3D AND abs,X    $71 ADC (zp),Y   $A8 TAY          $D6 DEC zp,X
$08 PHP          $3E ROL abs,X    $75 ADC zp,X     $A9 LDA #        $D8 CLD
$09 ORA #        $40 RTI          $76 ROR zp,X     $AA TAX          $D9 CMP abs,Y
$0A ASL @        $41 EOR (zp,X)   $78 SEI          $AC LDY abs      $DD CMP abs,X
$0D ORA abs      $45 EOR zp       $79 ADC abs,Y    $AD LDA abs      $DE DEC abs,X
$0E ASL abs      $46 LSR zp       $7D ADC abs,X    $AE LDX abs      $E0 CPX #
$10 BPL rel      $48 PHA          $7E ROR abs,X    $B0 BCS rel      $E1 SBC (zp,X)
$11 ORA (zp),Y   $49 EOR #        $81 STA (zp,X)   $B1 LDA (zp),Y   $E4 CPX zp
$15 ORA zp,X     $4A LSR @        $84 STY zp       $B4 LDY zp,X     $E5 SBC zp
$16 ASL zp,X     $4C JMP abs      $85 STA zp       $B5 LDA zp,X     $E6 INC zp
$18 CLC          $4D EOR abs      $86 STX zp       $B6 LDX zp,Y     $E8 INX
$19 ORA abs,Y    $4E LSR abs      $88 DEY          $B8 CLV          $E9 SBC #
$1D ORA abs,X    $50 BVC rel      $8A TXA          $B9 LDA abs,Y    $EA NOP
$1E ASL abs,X    $51 EOR (zp),Y   $8C STY abs      $BA TSX          $EC CPX abs
$20 JSR abs      $55 EOR zp,X     $8D STA abs      $BC LDY abs,X    $ED SBC abs
$21 AND (zp,X)   $56 LSR zp,X     $8E STX abs      $BD LDA abs,X    $EE INC abs
$24 BIT zp       $58 CLI          $90 BCC rel      $BE LDX abs,Y    $F0 BEQ rel
$25 AND zp       $59 EOR abs,Y    $91 STA (zp),Y   $C0 CPY #        $F1 SBC (zp),Y
$26 ROL zp       $5D EOR abs,X    $94 STY zp,X     $C1 CMP (zp,X)   $F5 SBC zp,X
$28 PLP          $5E LSR abs,X    $95 STA zp,X     $C4 CPY zp       $F6 INC zp,X
$29 AND #        $60 RTS          $96 STX zp,Y     $C5 CMP zp       $F8 SED
$2A ROL @        $61 ADC (zp,X)   $98 TYA          $C6 DEC zp       $F9 SBC abs,Y
$2C BIT abs      $65 ADC zp       $99 STA abs,Y    $C8 INY          $FD SBC abs,X
$2D AND abs      $66 ROR zp       $9A TXS          $C9 CMP #        $FE INC abs,X
$2E ROL abs      $68 PLA          $9D STA abs,X    $CA DEX
$30 BMI rel      $69 ADC #        $A0 LDY #        $CC CPY abs
$31 AND (zp),Y   $6A ROR @        $A1 LDA (zp,X)   $CD CMP abs
$35 AND zp,X     $6C JMP (abs)    $A2 LDX #        $CE DEC abs
```

A.3 Długości i czasy w pigułce

Tryb adresowania	Bajtów	Cykli: odczyt (`lda`, `adc`, `cmp`...)	Cykli: zapis (`sta`...)	Cykli: modyfikacja (`inc`, `asl`...)
natychmiastowy	2	2	—	—
akumulator	1	—	—	2
strona zerowa	2	3	3	5
strona zerowa,X / ,Y	2	4	4	6
absolutny	3	4	4	6
absolutny,X / ,Y	3	4 (+1 przy zmianie strony)	5	7
(zp,X)	2	6	6	—
(zp),Y	2	5 (+1 przy zmianie strony)	6	—

Pozostałe: rozkazy jednobajtowe (przesłania, znaczniki, `inx`, `dey`, `nop`) — 2 cykle; `pha`, `php` — 3; `pla`, `plp` — 4; `jmp` — 3 (pośredni 5); `jsr`, `rts`, `rti` — 6; `brk` — 7; skoki warunkowe — 2, wykonane 3, ze zmianą strony 4.

DODATEK B

Pseudorozkazy MADS i ich rozwinięcia

W tej książce nie używamy pseudorozkazów, ale spotkasz je w niemal każdym cudzym programie napisanym w MADS. Poniższa tabela pokazuje, na jakie rozkazy procesora asembler je zamienia. Rozwinięcia zostały odczytane z listingów wygenerowanych przez MADS 2.1.7.

B.1 Przesyłanie

Pseudorozkaz	Rozwinięcie	Uwagi
`mva zr cel`	`lda zr` / `sta cel`	niszczy A
`mvx zr cel`	`ldx zr` / `stx cel`	niszczy X
`mvy zr cel`	`ldy zr` / `sty cel`	niszczy Y
`mwa #adr cel`	`lda #<adr` / `sta cel` / `lda #>adr` / `sta cel+1`	przesłanie stałej 16-bitowej
`mwa zr cel`	`lda zr` / `sta cel` / `lda zr+1` / `sta cel+1`	przesłanie zmiennej 16-bitowej
`mwx`, `mwy`	jak `mwa`, ale przez rejestr X lub Y	

B.2 Arytmetyka

Pseudorozkaz	Rozwinięcie	Uwagi
`add arg`	`clc` / `adc arg`	
`sub arg`	`sec` / `sbc arg`	
`inw adr`	`inc adr` / `bne *+4` / `inc adr+1`	zwiększenie 16-bitowe
`inl adr`, `ind adr`	to samo dla 3 i 4 bajtów	
`dew adr`	`lda adr` / `bne *+4` / `dec adr+1` / `dec adr`	zmniejszenie 16-bitowe; **niszczy A**
`adb a b`	`lda a` / `clc` / `adc b` / `sta a`	dodawanie 8-bitowe
`sbb a b`	`lda a` / `sec` / `sbc b` / `sta a`	odejmowanie 8-bitowe
`adw a b`	`clc` / `lda a` / `adc b` / `sta a` / `lda a+1` / `adc b+1` / `sta a+1`	dodawanie 16-bitowe; trzeci argument wskazuje miejsce na wynik
`sbw a b`	`sec` / `lda a` / `sbc b` / `sta a` / `lda a+1` / `sbc b+1` / `sta a+1`	odejmowanie 16-bitowe

B.3 Porównania i stos

Pseudorozkaz	Rozwinięcie	Uwagi
`cpb a b`	`lda a` / `cmp b`	porównanie 8-bitowe
`cpw a b`	`lda a+1` / `cmp b+1` / `bne *+…` / `lda a` / `cmp b`	porównanie 16-bitowe od starszego bajtu
`phr`	`pha` / `txa` / `pha` / `tya` / `pha`	odłożenie A, X, Y
`plr`	`pla` / `tay` / `pla` / `tax` / `pla`	zdjęcie Y, X, A

B.4 Skoki

Pseudorozkaz	Rozwinięcie	Uwagi
`jeq adr`	`beq adr` albo `bne *+5` / `jmp adr`	skok warunkowy dowolnego zasięgu; MADS sam wybiera krótszą postać
`jne`, `jcc`, `jcs`, `jmi`, `jpl`, `jvc`, `jvs`	analogicznie	
`req`, `rne`, `rcc`, `rcs`, `rmi`, `rpl`, `rvc`, `rvs`	skok warunkowy **wstecz, do poprzedniego rozkazu**	np. `dex` / `rne` — powtarzaj `dex`, dopóki X ≠ 0
`seq`, `sne`, `scc`, `scs`, `smi`, `spl`, `svc`, `svs`	skok warunkowy **w przód, za następny rozkaz**	np. `scc` / `inc adr+1` — pomiń `inc`, gdy C = 0

B.5 Inne skróty

Zapis	Znaczenie
`lda:sta adr`	dwa rozkazy z tym samym argumentem: `lda adr` / `sta adr`
`lda:cmp:req 20`	`lda 20` / `cmp 20` / `beq` wstecz — czekanie na zmianę komórki 20 (zegara)
`:4 lsr @`	rozkaz powtórzony 4 razy
`run adr`	segment `org $02E0` / `dta a(adr)`
`ini adr`	segment `org $02E2` / `dta a(adr)`
`#if`, `#else`, `#end`	instrukcja warunkowa rozwijana na porównania i skoki
`#while`, `#end`	pętla rozwijana na porównania i skoki
`.proc`, `.endp`	procedura z własnym zasięgiem etykiet; może być pomijana, gdy nieużywana
`.macro`, `.endm`	definicja makra — nazwanego szablonu kodu z parametrami

Znając rozwinięcia, możesz czytać cudze źródła — i sam zdecydować, których skrótów chcesz używać. Rozsądny kompromis wielu programistów: pseudorozkazy przesyłania (`mva`, `mwa`) tak, bo niczego nie ukrywają; rozbudowane makra i `#if` — ostrożnie, bo łatwo stracić kontrolę nad tym, co faktycznie trafia do pamięci i które rejestry są niszczone.

DODATEK C

Rejestry sprzętowe Atari XL/XE

Wiele adresów ma **inne znaczenie przy zapisie, a inne przy odczycie** — to dwa różne rejestry pod jednym adresem.

C.1 GTIA ($D000–$D01F)

Adres	Zapis	Odczyt
$D000–$D003	HPOSP0–3: pozycje poziome graczy	M0PF–M3PF: kolizje pocisków z polem gry
$D004–$D007	HPOSM0–3: pozycje poziome pocisków	P0PF–P3PF: kolizje graczy z polem gry
$D008–$D00B	SIZEP0–3: szerokości graczy	M0PL–M3PL: kolizje pocisków z graczami
$D00C	SIZEM: szerokości pocisków	P0PL: kolizje gracza 0 z graczami
$D00D–$D00F	GRAFP0–2: kształt graczy (bez DMA)	P1PL–P3PL: kolizje graczy 1–3 z graczami
$D010–$D011	GRAFP3, GRAFM	TRIG0–1: przyciski joysticków
$D012–$D015	COLPM0–3: kolory graczy	— ($D014: PAL — 1 w systemie PAL, 15 w NTSC)
$D016–$D019	COLPF0–3: kolory pola gry	—
$D01A	COLBK: kolor tła	—
$D01B	PRIOR: priorytety, tryby GTIA	—
$D01D	GRACTL: włączenie grafiki PMG	—
$D01E	HITCLR: kasowanie kolizji	—
$D01F	CONSOL: głośnik (bit 3)	CONSOL: klawisze START, SELECT, OPTION

C.2 POKEY ($D200–$D20F)

Adres	Zapis	Odczyt
$D200, $D202, $D204, $D206	AUDF1–4: częstotliwości	POT0, POT2, POT4, POT6: potencjometry (wiosła)
$D201, $D203, $D205, $D207	AUDC1–4: głośność i barwa	POT1, POT3, POT5, POT7
$D208	AUDCTL: sterowanie dźwiękiem	ALLPOT: stan odczytu potencjometrów
$D209	STIMER: start liczników	KBCODE: kod klawisza
$D20A	SKREST: kasowanie błędów portu szeregowego	RANDOM: liczba losowa
$D20B	POTGO: start odczytu potencjometrów	—
$D20D	SEROUT: bajt do wysłania	SERIN: bajt odebrany
$D20E	IRQEN: maska przerwań	IRQST: stan przerwań
$D20F	SKCTL: sterowanie portem szeregowym	SKSTAT: stan portu i klawiatury

C.3 PIA ($D300–$D303)

Adres	Rejestr	Znaczenie
$D300	PORTA	joysticki: bity 0–3 port 1, bity 4–7 port 2 (0 = wychylenie)
$D301	PORTB	sterowanie pamięcią (rozdział 28)
$D302	PACTL	sterowanie portem A; bit 3: silnik magnetofonu
$D303	PBCTL	sterowanie portem B

C.4 ANTIC ($D400–$D40F)

Adres	Zapis	Odczyt
$D400	DMACTL: sterowanie obrazem i duszkami	—
$D401	CHACTL: sposób wyświetlania znaków	—
$D402–$D403	DLISTL/H: adres display list	—
$D404	HSCROL: płynny przesuw poziomy (0–15)	—
$D405	VSCROL: płynny przesuw pionowy (0–15)	—
$D407	PMBASE: adres pamięci duszków	—
$D409	CHBASE: adres zestawu znaków	—
$D40A	WSYNC: zatrzymanie procesora do końca linii	—
$D40B	—	VCOUNT: numer linii obrazu podzielony przez 2
$D40C–$D40D	—	PENH, PENV: pióro świetlne
$D40E	NMIEN: maska przerwań NMI	—
$D40F	NMIRES: kasowanie zgłoszenia NMI	NMIST: źródło NMI (bit 7 DLI, bit 6 VBI)

C.5 Wartości DMACTL / SDMCTL

Bity	Wartość	Znaczenie
0–1	0 / 1 / 2 / 3	obraz wyłączony / wąski (32 znaki) / normalny (40) / szeroki (48)
2	$04	pobieranie danych pocisków
3	$08	pobieranie danych graczy
4	$10	duszki w rozdzielczości jednoliniowej
5	$20	pobieranie display list (wyświetlanie obrazu)

Wartość standardowa: $22. Wpisanie zera do SDMCTL wygasza ekran — i przyspiesza pracę procesora o około 30%, co bywa wykorzystywane podczas długich obliczeń.

DODATEK D

Kody znaków

D.1 ATASCII a kody ekranowe

ATASCII	Znaki	Kod ekranowy
$00–$1F	znaki semigraficzne (z klawiszem CONTROL)	$40–$5F
$20–$2F	spacja ! " # $ % & ' ( ) * + , - . /	$00–$0F
$30–$39	cyfry 0–9	$10–$19
$3A–$40	: ; < = > ? @	$1A–$20
$41–$5A	wielkie litery A–Z	$21–$3A
$5B–$5F	[\ ] ^ _	$3B–$3F
$60	znak karo	$60
$61–$7A	małe litery a–z	$61–$7A
$7B–$7F	znak pik, kreska pionowa, znaki sterujące	$7B–$7F
$80–$FF	te same znaki w negatywie (poza kodami sterującymi)	bit 7 ustawiony

D.2 Kody sterujące edytora ekranowego

Kod	Dziesiętnie	Działanie	Klawisze
$1B	27	ESC — następny znak sterujący zostanie wyświetlony	ESC
$1C	28	kursor w górę	CONTROL + -
$1D	29	kursor w dół	CONTROL + =
$1E	30	kursor w lewo	CONTROL + +
$1F	31	kursor w prawo	CONTROL + *
$7D	125	czyszczenie ekranu	SHIFT + <
$7E	126	cofnięcie z kasowaniem	BACKSPACE
$7F	127	tabulacja	TAB
$9B	155	koniec linii (EOL)	RETURN
$9C	156	usunięcie wiersza	SHIFT + BACKSPACE
$9D	157	wstawienie wiersza	SHIFT + >
$FD	253	sygnał dźwiękowy	CONTROL + 2
$FE	254	usunięcie znaku	CONTROL + BACKSPACE
$FF	255	wstawienie znaku	CONTROL + >

DODATEK E

Słowniczek

Termin	Odpowiednik angielski	Znaczenie
akumulator	accumulator	rejestr A — główny rejestr obliczeniowy
asembler	assembler	program tłumaczący mnemoniki na kod maszynowy; potocznie także sam język
bank pamięci	memory bank	fragment pamięci dodatkowej widoczny w oknie adresowym po przełączeniu
cień rejestru	shadow register	kopia rejestru sprzętowego w pamięci RAM, przepisywana przez system co ramkę
cykl	cycle	jeden takt zegara procesora; jednostka czasu wykonania rozkazów
dyrektywa	directive	polecenie dla asemblera, niebędące rozkazem procesora (`org`, `dta`)
etykieta	label	nazwa nadana adresowi lub stałej
kod maszynowy	machine code	program w postaci bajtów wykonywanych przez procesor
kod rozkazu	opcode	pierwszy bajt rozkazu, określający operację i tryb adresowania
licznik rozkazów	program counter	rejestr PC — adres następnego rozkazu
mnemonik	mnemonic	trzyliterowa nazwa rozkazu (`lda`, `sta`)
nadmiar	overflow	przekroczenie zakresu liczb ze znakiem; znacznik V
przeniesienie	carry	dziewiąty bit wyniku; znacznik C
przerwanie	interrupt	chwilowe zawieszenie programu w celu obsługi zdarzenia
pseudorozkaz	pseudo-instruction	skrót asemblera rozwijany na kilka rozkazów procesora
ramka	frame	jeden pełny obraz na ekranie: 1/50 s (PAL) lub 1/60 s (NTSC)
rozkaz	instruction	pojedyncza operacja procesora
segment	segment	blok pliku wykonywalnego ładowany pod wskazany adres
sektor	sector	blok danych na dyskietce (128 lub 256 bajtów)
stos	stack	obszar `$0100`–`$01FF` do chwilowego przechowywania danych i adresów powrotu
strona	page	256 kolejnych bajtów pamięci o wspólnym starszym bajcie adresu
strona zerowa	zero page	adresy `$00`–`$FF`; krótsze rozkazy i jedyne miejsce na wskaźniki
tryb adresowania	addressing mode	sposób wskazania argumentu rozkazu
wektor	vector	dwubajtowa komórka z adresem procedury
wskaźnik	pointer	zmienna zawierająca adres
wskaźnik stosu	stack pointer	rejestr S — pozycja szczytu stosu
znacznik	flag	pojedynczy bit rejestru stanu procesora