Asembler 6502 na Atari XL/XE

Pętle, warunki i tryby adresowania w MADS

Trzy lekcje z zadaniami i rozwiązaniami

Spis treści

Lekcja 1: Pętle i znaczniki

- Lekcja 1 – zadania
- Lekcja 1 – rozwiązania

Lekcja 2: IF, IF…ELSE i CASE

- Lekcja 2 – zadania
- Lekcja 2 – rozwiązania

Lekcja 3: Tryby adresowania

- Lekcja 3 – zadania
- Lekcja 3 – rozwiązania

Lekcja 1: Pętle i znaczniki

1. Znaczniki: fundament każdej pętli

Procesor nie ma instrukcji „powtórz N razy". Pętla to zawsze trzy kroki: **zmień licznik → procesor ustawia znaczniki → skok warunkowy patrzy na znacznik**.

Znacznik	Kiedy = 1	Skoki
Z (zero)	wynik ostatniej operacji = 0	`BEQ` (Z=1), `BNE` (Z=0)
N (negative)	bit 7 wyniku = 1 (wartość $80–$FF)	`BMI` (N=1), `BPL` (N=0)
C (carry)	przeniesienie z dodawania, brak pożyczki przy odejmowaniu, wynik porównania	`BCS` (C=1), `BCC` (C=0)

Co ustawia znaczniki:

- **N i Z:** `LDA/LDX/LDY`, `INX/INY/DEX/DEY`, `INC/DEC`, `TAX/TAY/TXA/TYA`, `PLA`, `AND/ORA/EOR`
- **N, Z i C:** `ADC/SBC`, `CMP/CPX/CPY`, `ASL/LSR/ROL/ROR`
- **Nic nie zmieniają:** `STA/STX/STY`, `PHA`, `JSR`, `RTS`, `JMP`, skoki warunkowe

Uwaga: `INX/DEX/INC/DEC` **nie ruszają C**.

Porównanie `CMP #n` (tak samo `CPX`, `CPY`) to „odejmij na niby":

- `Z=1` → rejestr **równy** n
- `C=1` → rejestr **≥** n (bez znaku)
- `C=0` → rejestr **<** n

Złota zasada: skok patrzy na znaczniki z *ostatniej* instrukcji, która je zmieniła. Jeśli między `DEX` a `BNE` wstawisz `LDA`, skok sprawdzi A, a nie X.

2. Pętle 1-bajtowe (do 256 obrotów)

a) Malejąca, `DEX` + `BNE` (najprostsza)

```
        ldx #10         ; licznik = 10
loop    ; ... ciało ...
        dex             ; X=X-1, ustawia Z
        bne loop        ; X<>0 -> jeszcze raz
```

Wykona się 10 razy, X idzie 10, 9 … 1. Dla X=0 ciało się **nie** wykona, więc jako indeks tablicy pomija element 0.

b) Malejąca z zerem, `DEX` + `BPL`

```
        ldx #LEN-1      ; ostatni indeks
loop    lda src,x
        sta dst,x       ; STA nie psuje znaczników
        dex
        bpl loop        ; dopóki X >= 0
```

X idzie 9 … 0. Po zerze `DEX` daje $FF (bit 7 = 1, N=1) i pętla się kończy. **Ograniczenie:** indeks startowy najwyżej $80, bo wyższe wartości od razu „wyglądają" na ujemne.

c) Rosnąca, `INX` + `CPX`

```
        ldx #0
loop    lda src,x
        sta dst,x
        inx
        cpx #LEN        ; doszliśmy do końca?
        bne loop        ; (albo BCC loop = "dopóki X < LEN")
```

Koszt: dodatkowe `CPX` w każdym obrocie (2 bajty, 2 cykle).

d) Pełne 256 obrotów

```
        ldx #0
loop    sta bufor,x
        inx
        bne loop        ; po $FF X wraca na 0 -> koniec
```

e) Pętla „do znacznika końca" (napis zakończony zerem)

```
        ldy #0
loop    lda tekst,y     ; LDA ustawia Z
        beq koniec      ; bajt 0 = koniec napisu
        sta bufor,y     ; przykładowa obróbka znaku
        iny
        bne loop        ; zabezpieczenie: max 256 znaków
koniec
```

Tu nie ma licznika. Warunkiem jest **sama dana**.

f) Indeksowanie ujemne (rosnąca bez `CPX`)

Pomysł: X startuje od „−LEN" i rośnie do zera, więc wystarczy `BNE`. Adres bazowy przesuwamy tak, żeby się zgadzało.

```
LEN     = 10
        ldx #256-LEN            ; $F6 = "-10"
loop    lda src+LEN-256,x       ; dla X=$F6 to dokładnie src+0
        sta dst+LEN-256,x
        inx                     ; $F6,$F7...$FF,$00
        bne loop
```

Liczenie: `src+10−256+246 = src`. Ostatni obrót to X=$FF, czyli `src+9`. Dostajesz kolejność rosnącą przy koszcie pętli malejącej.

3. To samo na komórce pamięci

Gdy X i Y są zajęte, licznikiem może być bajt w pamięci:

```
licz    = $cb           ; strona zerowa ($CB-$CF są na Atari wolne)

        lda #10
        sta licz
loop    ; ... ciało (A, X, Y do dowolnego użytku) ...
        dec licz        ; DEC ustawia N i Z jak DEX
        bne loop
```

- `DEC` na stronie zerowej: 5 cykli, `DEX`: 2 cykle. Jest wolniej, ale rejestry zostają wolne.
- Komórki nie użyjesz jako indeksu (`lda tab,licz` nie istnieje). Trzeba zrobić `ldx licz`.
- Wersja rosnąca: `inc licz` / `lda licz` / `cmp #LEN` / `bne loop`.

4. Ochrona rejestrów

Problem: w środku pętli wołasz `JSR`, a procedura niszczy X. Procedury systemu Atari (CIO) niszczą **A, X i Y**. Przykładowa procedura wypisująca znak z A:

```
putchar pha
        ldx #0          ; kanał 0 = ekran
        lda #11         ; komenda PUT
        sta $342,x      ; ICCOM
        lda #0
        sta $348,x      ; ICBLL=0
        sta $349,x      ; ICBLH=0 -> znak z A
        pla
        jmp $e456       ; CIOV
```

Sposób 1: komórka tymczasowa (najszybszy)

```
        ldx #0
loop    stx savex       ; schowaj
        lda tekst,x
        jsr putchar     ; niszczy X
        ldx savex       ; odzyskaj
        inx
        cpx #LEN
        bne loop
```

Sposób 2: stos. 6502 w Atari nie ma `PHX/PHY`, więc idziemy przez A:

```
loop    txa
        pha             ; X na stos
        lda tekst,x
        jsr putchar
        pla
        tax             ; X ze stosu
        inx
        cpx #LEN
        bne loop
```

Każde `PHA` musi mieć swoje `PLA`. Jeśli wyskoczysz z pętli pomiędzy nimi, `RTS` wróci pod zły adres.

Sposób 3: procedura sama po sobie sprząta

```
proc1   pha
        txa
        pha
        tya
        pha
        ; ... robota ...
        pla             ; zdejmujemy w ODWROTNEJ kolejności
        tay
        pla
        tax
        pla
        rts
```

Sposób 4: samomodyfikacja (trik, szybki)

```
loop    stx odtworz+1   ; wpisz X w argument rozkazu niżej
        jsr putchar
odtworz ldx #$00        ; to $00 jest podmieniane
```

Pułapka ze znacznikami: `PLA`, `TAX` i `LDX savex` też ustawiają N/Z. Dlatego odtwarzaj rejestr **przed** `DEX/INX`, nigdy między `DEX` a `BNE`.

Pętle zagnieżdżone: X dla zewnętrznej, Y dla wewnętrznej:

```
        ldx #24         ; 24 wiersze
wiersz  ldy #39         ; 40 kolumn
kolumna ; ... ciało ...
        dey
        bpl kolumna
        dex
        bne wiersz
```

5. Pętle 2-bajtowe (do 65536 obrotów)

Liczba 16-bitowa to dwie komórki: **najpierw młodszy bajt, potem starszy** (`licz`,

`licz+1`).

a) Malejąca

```
        lda #<1000
        sta licz
        lda #>1000
        sta licz+1
loop    ; ... ciało ...
        lda licz        ; młodszy = 0?
        bne nizej
        dec licz+1      ; tak -> pożycz ze starszego
nizej   dec licz
        lda licz
        ora licz+1      ; ORA: wynik 0 tylko gdy OBA bajty = 0
        bne loop
```

`ORA` to standardowy sposób na „czy cała liczba = 0".

b) Rosnąca z porównaniem do granicy

```
loop    ; ... ciało ...
        inc licz
        bne dalej
        inc licz+1      ; młodszy się przekręcił -> starszy +1
dalej   lda licz
        cmp #<GRANICA   ; ustawia C
        lda licz+1
        sbc #>GRANICA   ; odejmowanie z pożyczką z młodszego
        bcc loop        ; C=0 -> licz < GRANICA
```

`CMP` + `SBC` to porównanie wielobajtowe. Końcowe C mówi „≥" albo „<".

c) Licznik ujemny (najkrótsza)

Wpisujemy **−N** i zwiększamy do zera, bez żadnych porównań:

```
        lda #<($10000-1000)
        sta licz
        lda #>($10000-1000)
        sta licz+1
loop    ; ... ciało ...
        inc licz
        bne loop
        inc licz+1
        bne loop        ; oba doszły do 0 -> koniec
```

d) Kopiowanie dużego bloku (X = strony, Y = indeks)

```
src     = $cb           ; wskaźniki na stronie zerowej
dst     = $cd
        ; wcześniej: src i dst ustawione na adresy bloków
        ldy #0
        ldx #>LEN       ; ile pełnych stron (po 256 b)
        beq reszta
strona  lda (src),y
        sta (dst),y
        iny
        bne strona      ; 256 bajtów
        inc src+1       ; następna strona
        inc dst+1
        dex
        bne strona
reszta  ldx #<LEN       ; pozostałe bajty
        beq koniec
ogon    lda (src),y
        sta (dst),y
        iny
        dex
        bne ogon
koniec
```

6. Pętle 4-bajtowe (32 bity)

a) Rosnąca z licznikiem ujemnym: po prostu dłuższy łańcuszek

```
loop    ; ... ciało ...
        inc licz
        bne loop
        inc licz+1
        bne loop
        inc licz+2
        bne loop
        inc licz+3
        bne loop
```

Prawie zawsze wykonuje się tylko pierwsze `INC` + `BNE`, więc jest bardzo szybka.

b) Malejąca: odejmowanie z pożyczką przez C

```
loop    ; ... ciało ...
        sec             ; C=1 = "brak pożyczki"
        lda licz
        sbc #1
        sta licz
        lda licz+1
        sbc #0          ; odejmie 1 tylko gdy była pożyczka
        sta licz+1
        lda licz+2
        sbc #0
        sta licz+2
        lda licz+3
        sbc #0
        sta licz+3
        ora licz+2      ; test zera wszystkich 4 bajtów
        ora licz+1
        ora licz
        bne loop
```

c) To samo odejmowanie zrobione pętlą w pętli

```
        ldx #0          ; indeks bajtu 0..3
        ldy #4          ; 4 bajty
        sec
odejm   lda licz,x
        sbc tab1,x      ; tab1: dta 1,0,0,0
        sta licz,x
        inx             ; INX/DEY nie ruszają C!
        dey
        bne odejm
```

Działa dlatego, że `INX` i `DEY` nie zmieniają C, więc pożyczka przechodzi między obrotami.

7. Gdy skok nie sięga: „Branch out of range"

Skoki warunkowe sięgają tylko ok. ±127 bajtów. Przy długim ciele pętli odwróć warunek i dołóż `JMP`:

```
        dex
        beq wyjscie     ; warunek odwrócony: koniec pętli
        jmp loop        ; JMP sięga wszędzie
wyjscie
```

Lekcja 1 – zadania

MADS, Atari XL/XE. Poziom rośnie łagodnie: od jednej pętli na rejestrze do licznika 4-bajtowego. Rozwiązania są w następnym rozdziale. Zajrzyj tam dopiero po własnej próbie.

Zanim zaczniesz

Szablon programu (wspólny dla wszystkich zadań):

```
        org $2000

start   ; ... tutaj Twój kod ...

        jmp *           ; zatrzymaj program (wyjście: RESET)

        run start
```

Jak zobaczyć wynik. Prawie wszystkie zadania piszą prosto do pamięci ekranu, więc efekt widać od razu.

- Adres początku ekranu leży w komórkach `$58` (młodszy) i `$59` (starszy).
- Zapis znaku na pozycji Y w pierwszym wierszu: `sta ($58),y`
- Ekran tekstowy ma 40 kolumn i 24 wiersze, razem 960 bajtów.
- Ekran używa **kodów wewnętrznych**, innych niż ATASCII. Litera `A` ma kod wewnętrzny 33, spacja ma kod 0.
- Napis w kodach wewnętrznych: `dta d'ATARI'`. Napis w ATASCII: `dta c'ATARI'`.

Wolne komórki strony zerowej: `$CB`–`$CF` (5 bajtów).

Poziom 1 – jedna pętla, jeden rejestr

Zadanie 1. Wiersz liter (pętla malejąca)

Wypełnij cały pierwszy wiersz ekranu (40 znaków) literą `A`. Użyj rejestru Y i pętli **malejącej**, bez instrukcji `CPY`.

Podpowiedź: indeksy to 39…0. Który skok kończy pętlę dopiero po przejściu przez zero?

Zadanie 2. Wiersz liter (pętla rosnąca)

To samo co w zadaniu 1, ale Y ma iść od 0 w górę.

Podpowiedź: tym razem potrzebne będzie porównanie. Zastanów się, czy po nim lepszy jest `BNE`, czy `BCC`, i dlaczego oba działają.

Zadanie 3. Cały zestaw znaków (256 obrotów)

Wypisz na ekranie wszystkie 256 znaków: na pozycji 0 znak o kodzie 0, na pozycji 1 znak o kodzie 1 itd. Pętla ma mieć dokładnie 256 obrotów i żadnego porównania.

Podpowiedź: kod znaku jest równy indeksowi. Jaki rozkaz przenosi Y do A? Co się stanie z Y po `INY`, gdy miał wartość `$FF`?

Poziom 2 – warunek w danych, dwa rejestry

Zadanie 4. Napis ze znacznikiem końca

Wypisz na ekranie napis, którego długości program **nie zna**. Koniec napisu oznacza bajt `$FF`.

```
tekst   dta d'ATARI XE',$ff
```

Podpowiedź: zero nie nadaje się tu na znacznik końca, bo to kod spacji. Sprawdzaj `$FF` rozkazem `CMP` zaraz po pobraniu znaku, przed zapisem na ekran.

Zadanie 5. Napis wspak (dwa rejestry naraz)

Wypisz napis od tyłu: ostatnia litera na pozycji 0 ekranu.

```
LEN     = 5
tekst   dta d'ATARI'
```

Użyj X do czytania napisu (malejąco) i Y do pisania na ekran (rosnąco).

Podpowiedź: w pętli są dwa rozkazy zmieniające znaczniki (`INY` i `DEX`). Skok ma zależeć od X. W jakiej kolejności musisz je ustawić?

Zadanie 6. Suma tablicy (indeksowanie ujemne)

Zsumuj 8 bajtów tablicy i wynik zapisz na pozycji 0 ekranu.

```
LEN     = 8
tab     dta 1,2,3,4,5,6,7,5
```

Tablicę czytaj **rosnąco** (od pierwszego elementu), ale bez `CPX`: użyj indeksowania ujemnego. Jeśli wszystko działa, na ekranie pojawi się litera `A` (suma = 33).

Podpowiedź: X startuje od `256-LEN`, a adres w rozkazie to `tab+LEN-256`. `CLC` wystarczy raz przed pętlą. Dlaczego `INX` nie psuje dodawania?

Poziom 3 – komórka pamięci i ochrona rejestrów

Zadanie 7. Licznik w komórce pamięci

Wypisz 20 liter `A` w pierwszym wierszu. Licznikiem obrotów ma być komórka `$CB`, a nie rejestr. Y służy tylko jako indeks i rośnie od 0. Bez `CPY`.

Podpowiedź: `DEC` ustawia znacznik Z tak samo jak `DEX`.

Zadanie 8. Procedura, która niszczy rejestry

Wypisz napis przez system (CIO), a nie przez pamięć ekranu. Koniec napisu to bajt `$9B`, którego nie drukujesz.

```
tekst   dta c'Hello Atari!',$9b

putchar pha             ; wypisuje znak z A, NISZCZY A, X, Y
        ldx #0
        lda #11
        sta $342,x
        lda #0
        sta $348,x
        sta $349,x
        pla
        jmp $e456
```

Indeksem ma być X. Zrób dwie wersje:

- **8a:** X chowasz w komórce pamięci,
- **8b:** X chowasz na stosie.

Podpowiedź do 8b: `TXA` nadpisuje A, w którym masz właśnie znak. Pilnuj też, żeby pętla nie kończyła się z bajtem pozostawionym na stosie.

Poziom 4 – pętle 2-bajtowe

Zadanie 9. Cały ekran

Wypełnij cały ekran (960 bajtów) literą `A`.

Zasady:

- nie zmieniaj komórek `$58/$59`, skopiuj adres do własnego wskaźnika `$CB/$CC`,
- X liczy pełne strony (po 256 bajtów), Y jest indeksem wewnątrz strony,
- po pełnych stronach dopisz resztę.

Podpowiedź: 960 = 3 × 256 + 192. Te liczby poda Ci MADS: `#>960` i `#<960`.

Zadanie 10. Migająca ramka (licznik ujemny 16-bitowy)

Zrób opóźnienie na 50000 obrotów pustej pętli, po nim zwiększ kolor ramki (`inc $2c8`) i zacznij od nowa, bez końca. Ramka powinna zmieniać kolor kilka razy na sekundę.

Licznik 2-bajtowy w `$CB/$CC` ma startować od wartości **−50000** i rosnąć do zera. W pętli nie może być żadnego porównania.

Podpowiedź: −50000 zapisz jako `$10000-50000`. Młodszy bajt: `#<(...)`, starszy: `#>(...)`.

Zadanie 11. Migająca ramka (licznik malejący 16-bitowy)

To samo co w zadaniu 10, ale licznik startuje od 50000 i **maleje** do zera.

Podpowiedź: potrzebujesz dwóch rzeczy: pożyczki ze starszego bajtu, gdy młodszy jest zerem, oraz testu „czy oba bajty są zerem" (`ORA`).

Poziom 5 – pętle 4-bajtowe

Zadanie 12. Wolno migająca ramka (licznik 32-bitowy)

Opóźnienie na 200000 obrotów. To już nie mieści się w 2 bajtach. Licznik 4-bajtowy w `$CB`–`$CE`.

Wartość początkową wczytaj z tablicy krótką pętlą 1-bajtową (4 obroty). Gotowe tablice:

```
plus    dta $40,$0d,$03,$00     ;  200000 = $00030D40
minus   dta $c0,$f2,$fc,$ff     ; -200000 = $FFFCF2C0
```

Zrób dwie wersje:

- **12a:** licznik ujemny, rosnący do zera (łańcuszek `INC` / `BNE`),
- **12b:** licznik malejący, odejmowanie 1 łańcuszkiem `SEC` / `SBC`, na końcu test zera czterech bajtów.

Podpowiedź do 12b: `SEC` dajesz tylko raz, przed najmłodszym bajtem. W starszych bajtach odejmujesz `#0`, a ewentualną pożyczkę przenosi znacznik C.

Dla chętnych

- W zadaniu 12b zastąp cztery powtórzone fragmenty odejmowania pętlą po rejestrze X. Dlaczego licznikiem tej pętli nie może być `CPX`?
- W zadaniu 9 wypełnij ekran kolejnymi znakami zamiast samą literą `A`.
- W zadaniu 4 dodaj zabezpieczenie na wypadek braku znacznika `$FF` w napisie.

Lekcja 1 – rozwiązania

Rozwiązania do zadań z poprzedniego rozdziału. Każde pokazuje jeden poprawny sposób. Jeśli Twój kod wygląda inaczej, a działa, to też jest dobrze. Porównaj wtedy długość i liczbę rozkazów w pętli.

Każdy fragment wstaw do szablonu:

```
        org $2000
start   ; ... kod ...
        jmp *
        run start
```

Zadanie 1. Wiersz liter (malejąca)

```
start   lda #33         ; kod wewnętrzny litery A
        ldy #39         ; ostatnia kolumna
loop    sta ($58),y     ; STA nie zmienia znaczników
        dey             ; 39,38...0, potem $FF
        bpl loop        ; dopóki Y >= 0
        jmp *
```

Dlaczego tak: `BNE` zakończyłby pętlę przy Y=0 i kolumna 0 zostałaby pusta. `BPL` kończy dopiero, gdy Y przekręci się na `$FF` (bit 7 = 1).

Zadanie 2. Wiersz liter (rosnąca)

```
start   lda #33
        ldy #0
loop    sta ($58),y
        iny
        cpy #40         ; porównaj Y z 40
        bne loop        ; różne -> dalej
        jmp *
```

Uwagi:

- Zamiast `bne loop` można dać `bcc loop` („dopóki Y < 40"). Tutaj oba działają identycznie. `BCC` jest bezpieczniejszy, gdy licznik skacze o więcej niż 1 i mógłby przeskoczyć granicę.
- W porównaniu z zadaniem 1 pętla ma o jeden rozkaz więcej. To jest koszt liczenia w górę.

Zadanie 3. Cały zestaw znaków

```
start   ldy #0
loop    tya             ; kod znaku = indeks
        sta ($58),y
        iny             ; po $FF wraca na 0
        bne loop        ; Z=1 dopiero po 256 obrotach
        jmp *
```

Dlaczego tak: start od 0 i `INY` + `BNE` dają dokładnie 256 obrotów, bo zero pojawia się dopiero po przekręceniu licznika.

Zadanie 4. Napis ze znacznikiem końca

```
start   ldy #0
loop    lda tekst,y
        cmp #$ff        ; znacznik końca?
        beq koniec      ; tak -> wychodzimy, nic nie piszemy
        sta ($58),y
        iny
        bne loop        ; zawsze skacze (chyba że napis > 255)
koniec  jmp *

tekst   dta d'ATARI XE',$ff
```

Uwagi:

- Test jest **na początku** pętli, przed zapisem. Dzięki temu znacznik `$FF` nie trafia na ekran, a pusty napis też działa.
- Końcowe `bne loop` pełni rolę `jmp loop`, ale jest o bajt krótsze i przy okazji chroni przed nieskończoną pętlą.

Zadanie 5. Napis wspak

```
LEN     = 5

start   ldx #LEN-1      ; ostatnia litera napisu
        ldy #0          ; pierwsza pozycja ekranu
loop    lda tekst,x
        sta ($58),y
        iny             ; najpierw Y...
        dex             ; ...DEX jako OSTATNI przed skokiem
        bpl loop        ; skok patrzy na znaczniki po DEX
        jmp *

tekst   dta d'ATARI'
```

Typowy błąd: kolejność `dex` / `iny` / `bpl`. Wtedy skok sprawdza Y, które jest zawsze dodatnie, i pętla nie kończy się we właściwym momencie.

Zadanie 6. Suma tablicy (indeksowanie ujemne)

```
LEN     = 8

start   lda #0          ; suma = 0
        clc             ; raz, przed pętlą
        ldx #256-LEN    ; $F8 = "-8"
loop    adc tab+LEN-256,x   ; dla X=$F8 to tab+0
        inx             ; $F8...$FF, potem 0
        bne loop
        ldy #0
        sta ($58),y     ; 33 = litera A
        jmp *

tab     dta 1,2,3,4,5,6,7,5
```

Dlaczego tak:

- Adres: `tab+8-256+$F8 = tab+0`, a w ostatnim obrocie `tab+8-256+$FF = tab+7`.
- `INX` zmienia tylko N i Z, znacznika C nie rusza, więc nie zakłóca `ADC`.
- Jedno `CLC` wystarcza, bo suma nie przekracza 255 i C zostaje wyzerowane po każdym dodawaniu.

Zadanie 7. Licznik w komórce pamięci

```
licz    = $cb

start   lda #20
        sta licz
        lda #33
        ldy #0
loop    sta ($58),y
        iny             ; ustawia znaczniki, ale...
        dec licz        ; ...DEC ustawia je na nowo
        bne loop        ; skok patrzy na licz
        jmp *
```

Uwaga: A cały czas trzyma kod znaku, bo `INY` i `DEC` go nie zmieniają. Gdyby licznik trzeba było ładować do A, znak należałoby wczytywać w każdym obrocie od nowa.

Zadanie 8a. Ochrona X w komórce pamięci

```
savex   = $cb

start   ldx #0
loop    lda tekst,x
        cmp #$9b
        beq koniec
        stx savex       ; schowaj X
        jsr putchar     ; niszczy A, X, Y
        ldx savex       ; odzyskaj X
        inx
        bne loop
koniec  jmp *

tekst   dta c'Hello Atari!',$9b

putchar pha
        ldx #0
        lda #11
        sta $342,x
        lda #0
        sta $348,x
        sta $349,x
        pla
        jmp $e456
```

Dlaczego tak: `LDX savex` zmienia znaczniki, ale zaraz po nim jest `INX`, który ustawia je ponownie. Skok widzi więc wynik `INX`.

Zadanie 8b. Ochrona X na stosie

```
start   ldx #0
loop    lda tekst,x
        cmp #$9b
        beq koniec      ; wyjście PRZED odłożeniem na stos
        txa
        pha             ; X na stos (A zniszczone)
        lda tekst,x     ; wczytaj znak jeszcze raz
        jsr putchar
        pla
        tax             ; X ze stosu
        inx
        bne loop
koniec  jmp *
```

Na co uważać:

- `TXA` nadpisuje znak w A, dlatego jest drugie `lda tekst,x`.
- Test końca stoi przed `PHA`. Gdyby był po nim, pętla kończyłaby się z jednym bajtem

zostawionym na stosie, a w procedurze zakończonej `RTS` oznaczałoby to powrót pod zły adres.
- Wersja 8a jest krótsza i szybsza. Stos wybiera się wtedy, gdy procedura może wywoływać samą siebie albo brakuje wolnych komórek.

Zadanie 9. Cały ekran

```
wsk     = $cb

start   lda $58         ; kopia adresu ekranu
        sta wsk
        lda $59
        sta wsk+1

        lda #33
        ldy #0
        ldx #>960       ; 3 pełne strony
strona  sta (wsk),y
        iny
        bne strona      ; 256 bajtów
        inc wsk+1       ; wskaźnik o stronę dalej
        dex
        bne strona

        ldx #<960       ; reszta: 192 bajty
ogon    sta (wsk),y     ; Y = 0 po pętli stron
        iny
        dex
        bne ogon
        jmp *
```

Dlaczego tak:

- Pętla wewnętrzna (Y) robi 256 obrotów, zewnętrzna (X) powtarza ją 3 razy. Przejście na następną stronę to `inc wsk+1`.
- Po ostatniej stronie Y równa się 0, więc pętla `ogon` zaczyna od właściwego miejsca bez dodatkowego `ldy #0`.
- Gdyby długość mogła być mniejsza niż 256 albo równa wielokrotności 256, przed każdą z pętli trzeba dodać `beq` omijające ją przy X=0.

Zadanie 10. Migająca ramka (licznik ujemny 16-bitowy)

```
licz    = $cb

start   lda #<($10000-50000)
        sta licz
        lda #>($10000-50000)
        sta licz+1
czekaj  inc licz
        bne czekaj      ; młodszy bajt jeszcze nie 0
        inc licz+1      ; młodszy się przekręcił
        bne czekaj      ; starszy jeszcze nie 0
        inc $2c8        ; zmień kolor ramki
        jmp start
```

Dlaczego tak: licznik idzie od −50000 do 0. Przez 255 na 256 obrotów wykonują się tylko dwa rozkazy (`inc` + `bne`). To najkrótsza możliwa pętla 16-bitowa.

Zadanie 11. Migająca ramka (licznik malejący 16-bitowy)

```
licz    = $cb

start   lda #<50000
        sta licz
        lda #>50000
        sta licz+1
czekaj  lda licz        ; młodszy = 0?
        bne nizej
        dec licz+1      ; tak -> pożyczka ze starszego
nizej   dec licz
        lda licz
        ora licz+1      ; 0 tylko gdy oba bajty = 0
        bne czekaj
        inc $2c8
        jmp start
```

Porównanie z zadaniem 10: ta pętla ma 6–7 rozkazów na obrót zamiast 2, więc ramka miga wyraźnie wolniej przy tej samej liczbie obrotów. Dlatego do samego odliczania wybiera się licznik ujemny. Licznik malejący jest potrzebny wtedy, gdy jego bieżąca wartość ma coś znaczyć (np. „zostało N bajtów").

Zadanie 12a. Licznik 32-bitowy ujemny

```
licz    = $cb

start   ldx #3          ; wczytaj 4 bajty wartości startowej
init    lda minus,x
        sta licz,x
        dex
        bpl init

czekaj  inc licz
        bne czekaj
        inc licz+1
        bne czekaj
        inc licz+2
        bne czekaj
        inc licz+3
        bne czekaj
        inc $2c8
        jmp start

minus   dta $c0,$f2,$fc,$ff     ; -200000
```

Zadanie 12b. Licznik 32-bitowy malejący

```
licz    = $cb

start   ldx #3
init    lda plus,x
        sta licz,x
        dex
        bpl init

czekaj  sec             ; C=1: brak pożyczki na wejściu
        lda licz
        sbc #1          ; najmłodszy bajt - 1
        sta licz
        lda licz+1
        sbc #0          ; odejmie 1 tylko przy pożyczce
        sta licz+1
        lda licz+2
        sbc #0
        sta licz+2
        lda licz+3
        sbc #0
        sta licz+3
        ora licz+2      ; A = licz+3, doklejamy pozostałe
        ora licz+1
        ora licz
        bne czekaj      ; którykolwiek bajt <> 0 -> dalej
        inc $2c8
        jmp start

plus    dta $40,$0d,$03,$00     ; 200000
```

Dlaczego tak:

- `STA` nie zmienia znacznika C, więc pożyczka przechodzi bez przeszkód z bajtu do bajtu.
- Po ostatnim `sta licz+3` w A nadal jest najstarszy bajt, dlatego test zera zaczyna się od razu od `ora`.
- 12a miga kilka razy szybciej niż 12b, bo w 12b każdy obrót wykonuje wszystkie rozkazy łańcuszka.

Dla chętnych – odejmowanie pętlą

```
czekaj  ldx #0          ; indeks bajtu
        ldy #4          ; liczba bajtów
        sec
odejm   lda licz,x
        sbc jeden,x
        sta licz,x
        inx             ; nie rusza C
        dey             ; nie rusza C
        bne odejm
        lda licz
        ora licz+1
        ora licz+2
        ora licz+3
        bne czekaj
        ...

jeden   dta 1,0,0,0
```

Dlaczego nie `CPX #4`: `CPX` ustawia znacznik C i zniszczyłoby pożyczkę przenoszoną między bajtami. `INX` i `DEY` zmieniają tylko N i Z, dlatego licznikiem jest tu Y liczony w dół.

Lekcja 2: IF, IF…ELSE i CASE

Główna myśl całej lekcji: w języku wysokiego poziomu piszesz „jeśli warunek, to wykonaj blok". W asemblerze robisz **odwrotnie**: „jeśli warunek NIE jest spełniony, to przeskocz blok".

1. IF … THEN

```
; IF zycia = 0 THEN ramka := czerwona
        lda zycia
        cmp #0          ; (po LDA zbędne, Z już ustawione)
        bne endif       ; NIE równe -> omiń blok
        lda #$34
        sta $2c8        ; blok THEN
endif
```

Ściąga: który skok omija blok

Po `CMP #n` (liczby bez znaku, 0–255):

Warunek w IF	Skok omijający blok
A = n	`BNE`
A <> n	`BEQ`
A < n	`BCS`
A >= n	`BCC`
A > n	`CMP #n+1`, potem `BCC`
A <= n	`CMP #n+1`, potem `BCS`

Dla „>" i „<=" nie ma pojedynczego skoku, więc porównujesz z liczbą o 1 większą: „A > 9" to to samo co „A >= 10".

Gdy porównujesz ze **zmienną** (nie da się dodać 1), zamień strony:

```
; IF a > b THEN ...   (to samo co: b < a)
        lda b
        cmp a
        bcs endif       ; b >= a -> omiń
        ; ... blok ...
endif
```

Warunki bez CMP

Wiele testów nie wymaga porównania, bo znaczniki ustawia już samo `LDA`:

```
        lda zmienna
        beq endif       ; IF zmienna <> 0 THEN
        ; ...

        lda zmienna
        bpl endif       ; IF bit 7 ustawiony THEN
        ; ...

        lda zmienna
        and #%00000100  ; IF bit 2 ustawiony THEN
        beq endif
        ; ...
```

Rozkaz `BIT` testuje komórkę **bez niszczenia A**: bit 7 komórki trafia do N, bit 6 do V, a Z mówi, czy `A AND komórka` dało zero.

```
        bit flagi
        bpl endif       ; IF bit 7 flag THEN
        bvc endif2      ; IF bit 6 flag THEN (BVC/BVS)
```

Warunki złożone

AND: każdy niespełniony warunek od razu wyskakuje za blok.

```
; IF znak >= 'A' AND znak <= 'Z' THEN ...
        lda znak
        cmp #'A'
        bcc endif       ; < 'A' -> nie
        cmp #'Z'+1
        bcs endif       ; > 'Z' -> nie
        ; ... blok: to wielka litera ...
endif
```

`CMP` nie zmienia A, więc drugi test nie potrzebuje ponownego `LDA`.

OR: każdy spełniony warunek skacze do bloku, a ostatni test jest odwrócony.

```
; IF a = 1 OR a = 5 THEN ...
        lda a
        cmp #1
        beq then        ; spełniony -> do bloku
        cmp #5
        bne endif       ; ostatni: odwrócony
then    ; ... blok ...
endif
```

Porównania 2-bajtowe

```
; IF punkty = rekord THEN (równość)
        lda punkty
        cmp rekord
        bne endif
        lda punkty+1
        cmp rekord+1
        bne endif
        ; ... blok ...
endif

; IF punkty >= rekord THEN (CMP + SBC z lekcji o pętlach)
        lda punkty
        cmp rekord
        lda punkty+1
        sbc rekord+1
        bcc endif       ; punkty < rekord -> omiń
        ; ... blok ...
endif
```

Liczby ze znakiem (−128…127)

Samo `BMI` po odejmowaniu nie wystarcza, bo wynik może się przepełnić. Pewny wzorzec:

```
; IF a < b THEN (ze znakiem)
        lda a
        sec
        sbc b
        bvc bezprz      ; brak przepełnienia
        eor #$80        ; przepełnienie -> odwróć znak
bezprz  bpl endif       ; nieujemny -> a >= b, omiń
        ; ... blok ...
endif
```

2. IF … THEN … ELSE

Schemat: blok THEN kończy się skokiem **ponad** blok ELSE.

```
; IF zycia < 3 THEN ramka := czerwona ELSE ramka := zielona
        lda zycia
        cmp #3
        bcs else        ; >= 3 -> do ELSE
        lda #$34        ; THEN
        sta $2c8
        jmp endif       ; nie wpadnij w ELSE!
else    lda #$c4        ; ELSE
        sta $2c8
endif
```

Najczęstszy błąd to zapomniane `jmp endif`. Wtedy po THEN wykonuje się też ELSE.

Trik 1: skok „zawsze" zamiast JMP. Gdy znasz stan znacznika, użyj skoku warunkowego (2 bajty zamiast 3). Tu po `lda #$34` Z jest zawsze 0:

```
        lda #$34
        sta $2c8
        bne endif       ; zawsze skacze
```

Trik 2: wartość domyślna zamiast ELSE. Gdy obie gałęzie tylko wybierają wartość, wpisz najpierw wariant ELSE i ewentualnie go podmień:

```
        ldx #$c4        ; domyślnie zielona
        lda zycia
        cmp #3
        bcs gotowe
        ldx #$34        ; jednak czerwona
gotowe  stx $2c8
```

Jest krócej, bez `JMP`, i wspólna część (`stx`) występuje raz.

IF … ELSE IF … ELSE to po prostu łańcuszek:

```
        lda temp
        cmp #10
        bcs nie1
        ; ... temp < 10 ...
        jmp endif
nie1    cmp #30
        bcs nie2
        ; ... 10 <= temp < 30 ...
        jmp endif
nie2    ; ... temp >= 30 (ELSE) ...
endif
```

Za długi blok? Skok warunkowy sięga ok. ±127 bajtów. Przy błędzie „Branch out of range" odwróć warunek i przeskocz nim rozkaz `JMP`:

```
        lda zycia
        beq wykonaj     ; warunek spełniony -> do bloku
        jmp endif       ; niespełniony: daleki skok za blok
wykonaj ; ... długi blok ...
endif
```

3. CASE … OF / SWITCH

Sposób A: łańcuszek porównań

Przykład na joysticku (`$278`: 14 = góra, 13 = dół, 11 = lewo, 7 = prawo):

```
        lda $278
        cmp #14
        beq gora
        cmp #13
        beq dol
        cmp #11
        beq lewo
        cmp #7
        beq prawo
        jmp endcase     ; ELSE/default: nic

gora    dec y
        jmp endcase
dol     inc y
        jmp endcase
lewo    dec x
        jmp endcase
prawo   inc x
endcase
```

- Dobry do kilku wartości (do ok. 5–6), także gdy wartości są „porozrzucane".
- Każdy przypadek kończy `jmp endcase`. Bez niego kod „wpada" w następny, jak `switch` bez `break` w C.
- Najczęstsze przypadki sprawdzaj na początku, bo ostatni czeka na wszystkie porównania.

Sposób B: zakresy

```
; CASE punkty OF 0..9, 10..49, reszta
        lda punkty
        cmp #10
        bcc slabo       ; < 10
        cmp #50
        bcc srednio     ; < 50 (i >= 10)
        ; ... świetnie ...
```

Sprawdzasz granice rosnąco i każde `BCC` „odcina" kolejny przedział.

Sposób C: tablica skoków

Gdy wartości to 0, 1, 2, 3..., nie porównujesz niczego. Wartość jest **indeksem** do tablicy adresów.

```
        lda wybor
        cmp #4          ; zabezpieczenie zakresu!
        bcs default
        asl @           ; *2, bo adres ma 2 bajty
        tax
        lda tab,x
        sta wektor
        lda tab+1,x
        sta wektor+1
        jmp (wektor)    ; skok pod adres z komórek

tab     dta a(opcja0),a(opcja1),a(opcja2),a(opcja3)
```

- Czas jest taki sam dla każdego przypadku, niezależnie od ich liczby.
- Bez testu zakresu zła wartość skoczy „w kosmos".
- `wektor` (2 bajty) nie może leżeć na granicy strony (adres `$xxFF`), bo `JMP ()` ma tam błąd w procesorze.

Sposób D: tablica skoków przez RTS

Odkładasz adres na stos i „wracasz" pod niego. Nie potrzeba komórek na wektor.

```
        ldx wybor       ; (po teście zakresu)
        lda tabhi,x
        pha             ; najpierw starszy bajt
        lda tablo,x
        pha
        rts             ; skok do opcji

tablo   dta <(opcja0-1),<(opcja1-1),<(opcja2-1)
tabhi   dta >(opcja0-1),>(opcja1-1),>(opcja2-1)
```

`RTS` dodaje 1 do zdjętego adresu, stąd **−1** w tablicach. Jeśli cały ten fragment jest procedurą wywołaną przez `JSR`, to `rts` na końcu każdej opcji wraca prosto do wywołującego.

Sposób E: tablica wartości zamiast kodu

Jeśli gałęzie różnią się tylko **danymi**, kod rozgałęziać się nie musi:

```
; CASE poziom OF 0: kolor:=$34; 1: kolor:=$c4; 2: kolor:=$86
        ldx poziom
        lda kolory,x
        sta $2c8

kolory  dta $34,$c4,$86
```

Trzy rozkazy zamiast całego CASE. Zanim napiszesz łańcuszek, zawsze sprawdź, czy nie wystarczy tablica.

Sposób F: wartości nieregularne + tablica

Połączenie pętli szukającej z tablicą skoków:

```
N       = 4
        lda $278
        ldx #N-1
szukaj  cmp klucze,x
        beq jest
        dex
        bpl szukaj
        jmp default     ; nie znaleziono
jest    ; X = numer przypadku -> sposób C, D lub E

klucze  dta 14,13,11,7
```

Co wybrać

- **2–5 przypadków, dowolne wartości:** łańcuszek (A)
- **Przedziały:** zakresy (B)
- **Wartości 0…N, różny kod:** tablica skoków (C lub D)
- **Wartości 0…N, różne tylko dane:** tablica wartości (E)
- **Dużo nieregularnych wartości:** szukanie + tablica (F)

Lekcja 2 – zadania

MADS, Atari XL/XE. Rozwiązania są w następnym rozdziale.

Zanim zaczniesz

Szablon programu jak poprzednio:

```
        org $2000
start   ; ... kod ...
        jmp *
        run start
```

Przydatne komórki Atari:

Adres	Co zawiera
`$284`	przycisk joysticka: 0 = wciśnięty, 1 = puszczony
`$278`	joystick: 15 = środek, 14 = góra, 13 = dół, 11 = lewo, 7 = prawo
`$D01F`	klawisze konsoli: bit 0 = START, bit 1 = SELECT, bit 2 = OPTION; bit = 0 oznacza wciśnięty
`$2C8`	kolor ramki
`$2C6`	kolor tła ekranu tekstowego
`$58/$59`	adres pamięci ekranu

Kody wewnętrzne znaków: `A` = 33, `B` = 34 … `Z` = 58, `a` = 97 … `z` = 122, cyfra `0` = 16, spacja = 0.

Kolory: `$34` czerwony, `$C4` zielony, `$86` niebieski.

Poziom 1 – proste IF

Zadanie 1. IF … THEN

W nieskończonej pętli: **jeśli** przycisk joysticka jest wciśnięty, **to** zwiększ kolor ramki (`inc $2c8`).

Podpowiedź: po `LDA` znacznik Z jest już ustawiony, `CMP #0` jest zbędne.

Zadanie 2. IF … THEN … ELSE

W nieskończonej pętli: przycisk wciśnięty → ramka czerwona, **w przeciwnym razie** → zielona.

Zrób dwie wersje:

- **2a:** klasyczna, z dwoma blokami i skokiem ponad ELSE,
- **2b:** z wartością domyślną (bez bloku ELSE i bez `JMP` w środku).

Zadanie 3. Porównanie „mniejsze niż"

Masz zmienną:

```
liczba  dta 7
```

Jeśli `liczba < 10`, wypisz na pozycji 0 ekranu literę `A`, w przeciwnym razie `B`. Przetestuj dla wartości 7, 9, 10 i 200.

Podpowiedź: po `CMP` znacznik C=0 oznacza „mniejsze".

Zadanie 4. IF … ELSE IF … ELSE

Rozbuduj zadanie 3 do trzech przedziałów:

- `liczba < 10` → `A`
- `10 <= liczba < 20` → `B`
- `liczba >= 20` → `C`

Zapis na ekran ma wystąpić w kodzie **tylko raz**.

Podpowiedź: `CMP` nie zmienia A, więc drugie porównanie nie wymaga ponownego `LDA`. Zamiast `JMP` spróbuj skoku, który „zawsze" skacze.

Poziom 2 – warunki złożone

Zadanie 5. Warunek AND (zakres)

Wypisz napis na ekranie, zamieniając **małe litery na wielkie**. Pozostałe znaki zostają bez zmian. Koniec napisu to `$FF`.

```
tekst   dta d'Atari Xe 130',$ff
```

Warunek: `IF znak >= 97 AND znak <= 122 THEN znak := znak - 64`.

Podpowiedź: „<= 122" sprawdź jako „< 123". Przed `SBC` pamiętaj o `SEC`.

Zadanie 6. Warunek OR

Policz, ile razy w napisie występuje litera `A` **lub** `E`. Wynik (jedna cyfra) wypisz na pozycji 0 ekranu.

```
tekst   dta d'ATARI XE GAME',$ff
```

Poprawny wynik: `5`.

Podpowiedź: Y jako indeks napisu, X jako licznik. Pierwszy spełniony warunek skacze od razu do zliczania, drugi jest odwrócony. Cyfra na ekranie to licznik + 16.

Zadanie 7. Testowanie bitów

W nieskończonej pętli sprawdzaj klawisze konsoli:

- wciśnięty START → ramka czerwona,
- wciśnięty SELECT → ramka zielona,
- żaden → ramka bez zmian.

Użyj `AND` z maską, a nie `CMP`.

Podpowiedź: klawisz wciśnięty to bit **równy 0**, więc po `AND` blok omijasz skokiem `BNE`.

Zadanie 8. Porównanie 2-bajtowe

```
punkty  dta a(1200)
rekord  dta a(1000)
```

Jeśli `punkty >= rekord`, przepisz punkty do rekordu i wypisz `A` (nowy rekord). W przeciwnym razie wypisz `B`. Przetestuj też dla punktów 999, 1000 i 256.

Podpowiedź: para `CMP` (młodsze bajty) + `SBC` (starsze bajty), potem jeden skok zależny od C.

Poziom 3 – CASE

Zadanie 9. CASE jako łańcuszek

W nieskończonej pętli czytaj joystick i wypisuj na pozycji 0 ekranu:

- góra → `G` (39)
- dół → `D` (36)
- lewo → `L` (44)
- prawo → `P` (48)
- cokolwiek innego → spacja (0)

Zapis na ekran ma być w kodzie tylko raz, na wspólnym końcu.

Zadanie 10. CASE jako tablica wartości

W nieskończonej pętli ustawiaj kolor ramki zależnie od klawiszy konsoli. Odczyt `$D01F` ograniczony do 3 bitów daje liczbę 0–7. Dla każdej z ośmiu wartości wybierz inny kolor.

Rozwiązanie ma mieć **zero porównań i zero skoków warunkowych**.

Podpowiedź: `AND #7`, `TAX`, tablica 8 bajtów.

Zadanie 11. CASE jako tablica skoków

```
wybor   dta 1
```

Zależnie od wartości:

- 0 → ramka czerwona,
- 1 → tło zielone,
- 2 → litera `A` na pozycji 0,
- inna → nic.

Zrób dwie wersje:

- **11a:** `JMP (wektor)`, wektor w `$CB/$CC`,
- **11b:** trik z `RTS`, całość jako procedura wywołana przez `JSR`.

Podpowiedź do 11b: w tablicach adresy pomniejszone o 1, na stos najpierw starszy bajt.

Zadanie 12. Szukanie w tablicy kluczy

Zrób zadanie 9 jeszcze raz, ale bez łańcuszka porównań. Użyj dwóch tablic:

```
klucze  dta 14,13,11,7
znaki   dta 39,36,44,48
```

Pętla szuka odczytu joysticka w `klucze`. Jeśli znajdzie, bierze znak z `znaki` spod tego samego indeksu. Jeśli nie, wypisuje spację.

Dla chętnych

- W zadaniu 5 dodaj odwrotną zamianę: wielkie litery na małe (obie naraz, czyli zamiana wielkości liter).
- W zadaniu 12 dodaj skosy joysticka (10, 6, 9, 5). Ile linii kodu musiałeś zmienić w porównaniu z zadaniem 9?
- W zadaniu 8 obsłuż trzy przypadki: większe, równe, mniejsze (`A`, `B`, `C`).

Lekcja 2 – rozwiązania

Rozwiązania do zadań z poprzedniego rozdziału. Każdy fragment wstaw do szablonu (`org $2000` … `run start`).

Zadanie 1. IF … THEN

```
start   lda $284        ; 0 = wciśnięty
        bne endif       ; nie zero -> omiń blok
        inc $2c8        ; blok THEN
endif   jmp start
```

Dlaczego tak: warunek brzmi „równe 0", więc blok omija skok odwrotny, `BNE`.

Zadanie 2a. IF … ELSE klasycznie

```
start   lda $284
        bne else
        lda #$34        ; THEN: czerwona
        sta $2c8
        jmp endif       ; przeskocz ELSE
else    lda #$c4        ; ELSE: zielona
        sta $2c8
endif   jmp start
```

Zadanie 2b. Wartość domyślna

```
start   ldx #$c4        ; domyślnie zielona
        lda $284
        bne gotowe
        ldx #$34        ; jednak czerwona
gotowe  stx $2c8
        jmp start
```

Porównanie: 2b jest krótsze, a zapis do `$2C8` występuje raz. Kolor trzymamy w X, bo A jest potrzebne do testu.

Zadanie 3. Mniejsze niż

```
start   lda liczba
        cmp #10
        bcs else        ; >= 10 -> ELSE
        lda #33         ; A
        jmp endif
else    lda #34         ; B
endif   ldy #0
        sta ($58),y
        jmp *

liczba  dta 7
```

Wyniki testu: 7 → A, 9 → A, 10 → B, 200 → B. Porównanie jest bez znaku, więc 200 to po prostu duża liczba.

Zadanie 4. Trzy przedziały

```
start   lda liczba
        cmp #10
        bcs nie1
        lda #33         ; < 10
        bne pisz        ; zawsze skacze (33 <> 0)
nie1    cmp #20         ; A nadal = liczba
        bcs nie2
        lda #34         ; 10..19
        bne pisz
nie2    lda #35         ; >= 20
pisz    ldy #0
        sta ($58),y
        jmp *

liczba  dta 7
```

Dlaczego tak: po `lda #33` znacznik Z jest na pewno wyzerowany, więc `bne pisz` działa jak `jmp`, a zajmuje 2 bajty zamiast 3.

Zadanie 5. Małe litery na wielkie (AND)

```
start   ldy #0
loop    lda tekst,y
        cmp #$ff
        beq koniec
        cmp #97         ; < 'a' ?
        bcc pisz        ; tak -> bez zmian
        cmp #123        ; > 'z' ?
        bcs pisz        ; tak -> bez zmian
        sec
        sbc #64         ; mała -> wielka
pisz    sta ($58),y
        iny
        bne loop
koniec  jmp *

tekst   dta d'Atari Xe 130',$ff
```

Dlaczego tak: oba warunki muszą być spełnione, więc każdy niespełniony od razu omija zamianę. Trzy kolejne `CMP` działają na tej samej wartości A.

Zadanie 6. Zliczanie A lub E (OR)

```
start   ldx #0          ; licznik
        ldy #0          ; indeks
loop    lda tekst,y
        cmp #$ff
        beq koniec
        cmp #33         ; A ?
        beq licz        ; tak -> od razu zliczaj
        cmp #37         ; E ?
        bne dalej       ; ostatni warunek odwrócony
licz    inx
dalej   iny
        bne loop
koniec  txa
        clc
        adc #16         ; liczba -> cyfra
        ldy #0
        sta ($58),y
        jmp *

tekst   dta d'ATARI XE GAME',$ff
```

Kolejność ma znaczenie: `INY` jest po `INX`, więc `bne loop` patrzy na Y.

Zadanie 7. Testowanie bitów

```
start   lda $d01f
        and #%00000001  ; START
        bne nstart      ; bit = 1 -> nie wciśnięty
        lda #$34
        sta $2c8
nstart  lda $d01f
        and #%00000010  ; SELECT
        bne nselect
        lda #$c4
        sta $2c8
nselect jmp start
```

Uwaga: `AND` niszczy A, dlatego przed drugim testem jest ponowne `lda $d01f`. To dwa niezależne IF-y, a nie IF…ELSE: przy obu klawiszach naraz wygra drugi.

Zadanie 8. Porównanie 2-bajtowe

```
start   lda punkty
        cmp rekord      ; młodsze bajty, ustawia C
        lda punkty+1
        sbc rekord+1    ; starsze z pożyczką
        bcc else        ; punkty < rekord
        lda punkty      ; THEN: nowy rekord
        sta rekord
        lda punkty+1
        sta rekord+1
        lda #33
        bne pisz
else    lda #34
pisz    ldy #0
        sta ($58),y
        jmp *

punkty  dta a(1200)
rekord  dta a(1000)
```

Wyniki testu: 1200 → A, 1000 → A (równe też spełnia „>="), 999 → B, 256 → B. Przypadek 256 pokazuje, po co jest starszy bajt: młodszy bajt 256 to 0, a młodszy bajt 1000 to 232.

Zadanie 9. CASE jako łańcuszek

```
start   lda $278
        cmp #14
        beq gora
        cmp #13
        beq dol
        cmp #11
        beq lewo
        cmp #7
        beq prawo
        lda #0          ; default: spacja
        beq pisz        ; zawsze skacze (Z=1)
gora    lda #39
        bne pisz
dol     lda #36
        bne pisz
lewo    lda #44
        bne pisz
prawo   lda #48         ; ostatni wpada prosto w pisz
pisz    ldy #0
        sta ($58),y
        jmp start
```

Zadanie 10. CASE jako tablica wartości

```
start   lda $d01f
        and #7          ; zostaw bity 0-2
        tax
        lda kolory,x
        sta $2c8
        jmp start

kolory  dta $34,$c4,$86,$18,$56,$a8,$e6,$00
```

Dlaczego tak: wartość jest indeksem, więc osiem gałęzi mieści się w czterech rozkazach. Indeks 7 (nic nie wciśnięte) daje czarną ramkę.

Zadanie 11a. JMP (wektor)

```
wektor  = $cb

start   lda wybor
        cmp #3
        bcs endcase     ; poza zakresem -> nic
        asl @           ; *2
        tax
        lda tab,x
        sta wektor
        lda tab+1,x
        sta wektor+1
        jmp (wektor)

opcja0  lda #$34
        sta $2c8
        jmp endcase
opcja1  lda #$c4
        sta $2c6
        jmp endcase
opcja2  lda #33
        ldy #0
        sta ($58),y
endcase jmp *

wybor   dta 1
tab     dta a(opcja0),a(opcja1),a(opcja2)
```

Zadanie 11b. Trik z RTS

```
start   jsr wybierz
        jmp *

wybierz ldx wybor
        cpx #3
        bcs wyjdz       ; poza zakresem
        lda tabhi,x
        pha             ; starszy bajt pierwszy
        lda tablo,x
        pha
wyjdz   rts             ; "wraca" do opcji albo do start

opcja0  lda #$34
        sta $2c8
        rts             ; wraca za JSR wybierz
opcja1  lda #$c4
        sta $2c6
        rts
opcja2  lda #33
        ldy #0
        sta ($58),y
        rts

wybor   dta 1
tablo   dta <(opcja0-1),<(opcja1-1),<(opcja2-1)
tabhi   dta >(opcja0-1),>(opcja1-1),>(opcja2-1)
```

Dlaczego tak:

- Jedno `rts` pełni dwie role. Poza zakresem na stosie leży tylko adres powrotu, więc wraca do `start`. W zakresie zdejmuje podłożony adres opcji.
- Opcje kończą się `rts` zamiast `jmp endcase`, bo na stosie nadal czeka adres powrotu z `jsr wybierz`.
- W 11b nie ma mnożenia przez 2 ani komórek na wektor.

Zadanie 12. Szukanie w tablicy kluczy

```
N       = 4

start   lda $278
        ldx #N-1
szukaj  cmp klucze,x
        beq jest
        dex
        bpl szukaj
        lda #0          ; nie znaleziono: spacja
        beq pisz
jest    lda znaki,x
pisz    ldy #0
        sta ($58),y
        jmp start

klucze  dta 14,13,11,7
znaki   dta 39,36,44,48
```

Porównanie z zadaniem 9: dodanie nowego kierunku to dopisanie po jednej liczbie do obu tablic i zmiana `N`. Kod zostaje ten sam.

Lekcja 3: Tryby adresowania

Tryb adresowania odpowiada na pytanie: **skąd rozkaz ma wziąć daną (albo gdzie ją zapisać)?** Ten sam `LDA` może znaczyć „weź liczbę 5", „weź zawartość komórki" albo „weź zawartość komórki, której adres leży w innych komórkach".

1. Tryby bez adresu

Domyślny (implied). Rozkaz sam wie, na czym działa. Zajmuje 1 bajt.

```
        inx
        clc
        rts
```

Akumulator. Rozkazy przesunięć działające na A. W MADS-ie zapisujesz to małpą:

```
        asl @           ; A = A * 2
        lsr @           ; A = A / 2
```

2. Natychmiastowy: `#`

Dana jest **wpisana w rozkaz**. Zajmuje 2 bajty.

```
        lda #5          ; A = liczba 5
        lda #<adres     ; młodszy bajt adresu
        lda #>adres     ; starszy bajt adresu
```

Najczęstszy błąd początkujących to zgubiony `#`:

```
        lda #5          ; A = 5
        lda 5           ; A = zawartość komórki nr 5 (!)
```

3. Strona zerowa i absolutny

Pamięć dzieli się na 256 **stron** po 256 bajtów. Starszy bajt adresu to numer strony: `$2C8` leży na stronie `$02`, a `$00CB` na stronie `$00` (stronie zerowej).

Absolutny. Pełny, 2-bajtowy adres. Zajmuje 3 bajty, trwa 4 cykle.

```
        lda $2c8        ; w pamięci: AD C8 02
```

Adres jest zapisany „od tyłu": najpierw młodszy bajt (`C8`), potem starszy (`02`). Tę kolejność spotkasz wszędzie w 6502.

Strona zerowa. Adres 1-bajtowy, bo strona jest z góry znana (`$00`). Zajmuje 2 bajty, trwa 3 cykle.

```
        lda $cb         ; w pamięci: A5 CB
```

MADS sam wybiera krótszą wersję, gdy adres jest mniejszy niż `$100`. Strona zerowa jest więc krótsza i szybsza, ale jej prawdziwa wartość ujawni się w punkcie 7.

4. Indeksowane: `,X` i `,Y`

Do adresu z rozkazu procesor **dodaje rejestr**.

```
        ldx #3
        lda tab,x       ; czyta komórkę tab+3
        lda tab,y       ; to samo z Y
```

- Zasięg to 256 bajtów (indeks 0–255) od adresu bazowego.
- Adres bazowy jest stały, wpisany w program. Zmienia się tylko indeks.
- Nie każdy rozkaz ma oba warianty: `INC`, `DEC`, `ASL`, `LSR`, `ROL`, `ROR` działają tylko z `,X`.

Strona zerowa indeksowana (`lda $cb,x`) działa tak samo, z jedną pułapką: wynik **nigdy nie wychodzi poza stronę zerową**.

```
        ldx #2
        lda $ff,x       ; czyta $01, a nie $101!
```

Wersję `zp,Y` mają tylko `LDX` i `STX`.

5. Względny: skoki warunkowe

`BNE`, `BEQ`, `BCC` itd. nie przechowują adresu, tylko **odległość** od bieżącego miejsca (1 bajt, od −128 do +127). Stąd błąd „Branch out of range". W kodzie piszesz normalnie etykietę, a odległość liczy MADS.

6. Pośredni: `JMP (adres)`

Tylko dla `JMP`. Nawias znaczy: „adres w nawiasie to nie cel, tylko **miejsce, gdzie leży cel**".

```
        jmp ($000c)     ; skocz tam, gdzie każą komórki $0C/$0D
```

To był mechanizm tablicy skoków z lekcji 2.

7. Pośredni przez stronę zerową: `(zp),Y`

To najważniejszy tryb całego procesora i jedyny wygodny sposób dostępu do **dowolnego adresu wyliczonego w trakcie działania programu**.

Problem

`lda tab,x` ma adres wpisany na stałe i sięga tylko 256 bajtów. A co, jeśli adres poznajesz dopiero w trakcie (np. adres ekranu z `$58/$59`) albo blok ma 8000 bajtów?

Rozwiązanie: wskaźnik

Dwie sąsiednie komórki **strony zerowej** przechowują adres (młodszy bajt, potem starszy). Taka para to **wskaźnik**.

```
wsk     = $cb           ; zajmuje $CB i $CC

        lda #<$4000
        sta wsk         ; $CB = $00
        lda #>$4000
        sta wsk+1       ; $CC = $40

        ldy #5
        lda (wsk),y     ; czyta komórkę $4005
```

Co robi procesor przy `lda (wsk),y`, krok po kroku:

1. Zagląda do `$CB` i `$CC` na stronie zerowej.
2. Składa z nich adres: `$4000`.
3. Dodaje Y: `$4000 + 5 = $4005`.
4. Dopiero stamtąd czyta bajt do A.

Sam rozkaz zajmuje 2 bajty i zawiera tylko numer komórki `$CB`. Docelowy adres może leżeć **gdziekolwiek** w 64 KB: w RAM-ie, na ekranie, w ROM-ie.

Dlaczego akurat strona zerowa

Bo procesor nie daje wyboru: wskaźnik dla trybu `( ),Y` **musi** leżeć na stronie zerowej. Dlatego wolne komórki strony zerowej (`$CB`–`$CF`) są tak cenne i używa się ich głównie na wskaźniki.

Jak się poruszać po pamięci

- **W obrębie 256 bajtów:** zmieniasz Y.
- **Na następną stronę:** `inc wsk+1` (starszy bajt to numer strony).
- **O dowolną wartość:** dodawanie 2-bajtowe, np. następny wiersz ekranu:

```
        clc
        lda wsk
        adc #40         ; +40 znaków
        sta wsk
        bcc ok
        inc wsk+1       ; przeniesienie -> następna strona
ok
```

- **Bez indeksu:** 6502 w Atari nie ma trybu `(wsk)`, więc dajesz `ldy #0`.

8. Pośredni indeksowany przed: `(zp,X)`

Rzadki kuzyn poprzedniego. Tu X wybiera **który wskaźnik**, a nie przesunięcie w danych.

```
        ldx #2
        lda ($cb,x)     ; użyj wskaźnika z $CD/$CE
```

Kolejność: najpierw `$CB + X`, potem odczyt adresu spod tej pary i odczyt bajtu. Przydaje się

przy tablicy wskaźników na stronie zerowej. W praktyce używasz go rzadko.

Różnica w jednym zdaniu: w `(zp),Y` indeks jest dodawany **do adresu docelowego**, w `(zp,X)` **do adresu wskaźnika**.

9. Kopiowanie bloków pamięci

A. Do 256 bajtów: absolutny indeksowany

```
        ldx #0
kopiuj  lda zrodlo,x
        sta cel,x
        inx
        cpx #DLUG
        bne kopiuj
```

B. Kilka stron, stałe adresy: pętla rozwinięta

Dla bloku 1024 bajtów (4 strony) o adresach znanych z góry:

```
        ldx #0
kopiuj  lda zrodlo,x
        sta cel,x
        lda zrodlo+$100,x
        sta cel+$100,x
        lda zrodlo+$200,x
        sta cel+$200,x
        lda zrodlo+$300,x
        sta cel+$300,x
        inx
        bne kopiuj
```

Bardzo szybkie, ale działa tylko dla stałego rozmiaru będącego wielokrotnością 256.

C. Dowolny blok: `(zp),Y`

```
src     = $cb
dst     = $cd
DLUG    = 3000

        lda #<zrodlo    ; src = adres źródła
        sta src
        lda #>zrodlo
        sta src+1
        lda #<cel       ; dst = adres celu
        sta dst
        lda #>cel
        sta dst+1

        ldy #0
        ldx #>DLUG      ; liczba pełnych stron
        beq reszta
strona  lda (src),y
        sta (dst),y
        iny
        bne strona      ; 256 bajtów
        inc src+1       ; oba wskaźniki o stronę dalej
        inc dst+1
        dex
        bne strona

reszta  ldx #<DLUG      ; pozostałe bajty
        beq koniec
ogon    lda (src),y
        sta (dst),y
        iny
        dex
        bne ogon
koniec
```

Y chodzi po bajtach strony, X liczy strony, a `inc src+1` przestawia wskaźnik na następną stronę. To standardowa procedura kopiująca na 6502.

D. Bloki nachodzące na siebie: kopiowanie od końca

Gdy cel leży **wyżej** niż źródło i bloki się pokrywają (np. przesuwasz dane o 10 bajtów w górę), kopiowanie od początku nadpisze dane, zanim je przeczytasz. Trzeba iść od końca:

```
        lda #<zrodlo    ; src = adres źródła
        sta src
        lda #>zrodlo
        sta src+1
        lda #<cel       ; dst = adres celu
        sta dst
        lda #>cel
        sta dst+1

        clc             ; ustaw wskaźniki na ostatnią,
        lda src+1       ; niepełną stronę
        adc #>DLUG
        sta src+1
        clc
        lda dst+1
        adc #>DLUG
        sta dst+1

        ldy #<DLUG      ; najpierw końcówka
        beq pelne
ogon    dey
        lda (src),y
        sta (dst),y
        cpy #0
        bne ogon

pelne   ldx #>DLUG      ; potem pełne strony w dół
        beq koniec
strona  dec src+1
        dec dst+1
kopia   dey             ; 0 -> $FF, czyli koniec strony
        lda (src),y
        sta (dst),y
        cpy #0
        bne kopia
        dex
        bne strona
koniec
```

Zasada: **cel wyżej niż źródło → kopiuj od końca; cel niżej → od początku** (wersja C).

Ściąga

Tryb	Zapis	Bajty
Domyślny	`inx`	1
Akumulator	`asl @`	1
Natychmiastowy	`lda #5`	2
Strona zerowa	`lda $cb`	2
Strona zerowa,X / ,Y	`lda $cb,x`	2
Absolutny	`lda $2c8`	3
Absolutny,X / ,Y	`lda tab,x`	3
Względny	`bne loop`	2
Pośredni	`jmp ($c)`	3
Pośredni,Y	`lda (wsk),y`	2
Pośredni,X	`lda (wsk,x)`	2

Lekcja 3 – zadania

MADS, Atari XL/XE. Rozwiązania są w następnym rozdziale.

Zanim zaczniesz

Szablon programu jak poprzednio (`org $2000` … `jmp *` … `run start`).

- Adres ekranu: `$58/$59`. Ekran ma 40 × 24 = 960 bajtów.
- Wolne komórki strony zerowej: `$CB`–`$CF`. Starczy na dwa wskaźniki (`$CB/$CC` i `$CD/$CE`).
- Wolna pamięć na bufor: od `$4000`.
- Kody wewnętrzne: `A` = 33, `B` = 34, kreska pionowa `|` = 124.
- Klawisz START: bit 0 komórki `$D01F`, wciśnięty = 0.

Poziom 1 – zadania „na kartce"

Zadanie 1. Ile bajtów zajmuje rozkaz?

Podaj długość każdego rozkazu i nazwę trybu adresowania (`tab` = `$3000`):

```
a)      inx
b)      lda #0
c)      lda $80
d)      lda $0600
e)      lda tab,x
f)      bne petla
g)      lda ($cb),y
h)      jmp ($000c)
i)      asl @
j)      stx $80,y
```

Zadanie 2. Skąd procesor weźmie bajt?

Stan pamięci i rejestrów:

```
$CB = $40    $CC = $9C    $CD = $00    $CE = $60
X = $02      Y = $10
```

Podaj adres komórki, z której czyta każdy rozkaz:

```
a)      lda $cb
b)      lda #$cb
c)      lda $cb,x
d)      lda ($cb),y
e)      lda ($cb,x)
f)      lda $9c40,y
g)      lda $ff,x
```

Poziom 2 – indeksowanie absolutne

Zadanie 3. Bufor i ekran

1. Wypełnij 256 bajtów bufora pod `$4000` kolejnymi wartościami 0…255 (tryb `absolutny,X`).
2. Skopiuj ten bufor na ekran (odczyt `absolutny,Y`, zapis `(zp),Y`).

Na ekranie powinien pojawić się cały zestaw znaków.

Poziom 3 – własny wskaźnik

Zadanie 4. Drugi wiersz

Wypisz literę `A` na początku **drugiego** wiersza ekranu, mając w Y wartość 0.

Kroki: skopiuj `$58/$59` do własnego wskaźnika w `$CB/$CC`, dodaj do niego 40 (dodawanie 2-bajtowe), zapisz znak.

Podpowiedź: nie zmieniaj samych komórek `$58/$59`, system ich używa.

Zadanie 5. Przekątna

Narysuj literami `A` przekątną przez wszystkie 24 wiersze: wiersz 0 kolumna 0, wiersz 1 kolumna 1 itd.

Y ma być stale równe 0. Przesuwa się tylko wskaźnik.

Podpowiedź: o ile bajtów trzeba przesunąć wskaźnik, żeby trafić wiersz niżej i kolumnę w prawo?

Zadanie 6. Dwie pionowe linie

Narysuj znakiem `|` (kod 124) lewą i prawą krawędź ekranu: kolumny 0 i 39 we wszystkich 24 wierszach.

Tym razem wskaźnik wskazuje początek wiersza, a Y wybiera kolumnę.

Podpowiedź: dodawanie do wskaźnika niszczy A. Gdzie w pętli musi stać `lda #124`?

Poziom 4 – dwa wskaźniki, kopiowanie

Zadanie 7. Kopia wiersza

1. Wypisz napis `ATARI` w pierwszym wierszu.
2. Skopiuj cały pierwszy wiersz (40 bajtów) do ostatniego, 24. wiersza.

Użyj dwóch wskaźników: `src` w `$CB/$CC` i `dst` w `$CD/$CE`.

Podpowiedź: ostatni wiersz zaczyna się 23 × 40 = 920 bajtów za początkiem ekranu. Dodaj 920 jako liczbę 2-bajtową (`#<920`, `#>920`).

Zadanie 8. Zapamiętaj i odtwórz ekran

Napisz procedurę `kopiuj`, która kopiuje 960 bajtów z adresu we wskaźniku `src` pod adres we wskaźniku `dst` (pełne strony + reszta).

Program główny:

1. kopiuje ekran do bufora `$4000`,
2. wypełnia cały ekran literą `A`,
3. czeka na START,
4. kopiuje bufor z powrotem na ekran.

Podpowiedź: procedura zmienia starsze bajty wskaźników, więc przed każdym wywołaniem ustaw oba od nowa.

Zadanie 9. Przesunięcie ekranu w prawo

Przesuń całą zawartość ekranu o jeden znak w prawo: skopiuj 959 bajtów z adresu ekranu pod adres ekranu + 1.

Bloki nachodzą na siebie, a cel leży wyżej niż źródło.

Najpierw eksperyment: użyj procedury z zadania 8 (kopiowanie od początku) i zobacz, co się stanie. Potem napisz wersję kopiującą od końca.

Poziom 5 – ciekawostka

Zadanie 10. Tryb `(zp,X)`

Ustaw dwa wskaźniki: `$CB/$CC` na początek pierwszego wiersza, `$CD/$CE` na początek drugiego. Zapisz `A` przez pierwszy wskaźnik i `B` przez drugi, używając w obu przypadkach **tego samego** rozkazu `sta ($cb,x)`, tylko z inną wartością X.

Dla chętnych

- Zadanie 9 w drugą stronę: przesuń ekran o jeden znak w lewo. Który kierunek kopiowania jest tu właściwy?
- Zadanie 8 z parametrem: niech długość bloku będzie w dwóch komórkach pamięci, a nie stałą wpisaną w kod.
- Zadanie 5: narysuj drugą przekątną, od prawego górnego rogu w dół w lewo.

Lekcja 3 – rozwiązania

Rozwiązania do zadań z poprzedniego rozdziału. Każdy fragment kodu wstaw do szablonu (`org $2000` … `run start`).

Zadanie 1. Długości rozkazów

	Rozkaz	Bajty	Tryb
a	`inx`	1	domyślny
b	`lda #0`	2	natychmiastowy
c	`lda $80`	2	strona zerowa
d	`lda $0600`	3	absolutny
e	`lda tab,x`	3	absolutny,X
f	`bne petla`	2	względny
g	`lda ($cb),y`	2	pośredni,Y
h	`jmp ($000c)`	3	pośredni
i	`asl @`	1	akumulator
j	`stx $80,y`	2	strona zerowa,Y

Reguła: 1 bajt to sam rozkaz, 2 bajty to rozkaz + jeden bajt (liczba, adres na stronie zerowej albo odległość skoku), 3 bajty to rozkaz + pełny adres.

Zadanie 2. Skąd procesor weźmie bajt

	Rozkaz	Adres	Jak policzone
a	`lda $cb`	`$00CB`	wprost
b	`lda #$cb`	żaden	A = liczba `$CB`
c	`lda $cb,x`	`$00CD`	`$CB + 2`
d	`lda ($cb),y`	`$9C50`	wskaźnik `$9C40` + `$10`
e	`lda ($cb,x)`	`$6000`	wskaźnik spod `$CD/$CE`
f	`lda $9c40,y`	`$9C50`	`$9C40 + $10`
g	`lda $ff,x`	`$0001`	`$FF + 2` zawija się na stronie zerowej

Warto zauważyć: d i f czytają tę samą komórkę. Różnica: w f adres `$9C40` jest wpisany w program na stałe, w d leży w komórkach i można go zmienić w trakcie działania.

Zadanie 3. Bufor i ekran

```
bufor   = $4000

start   ldx #0
wypeln  txa
        sta bufor,x     ; absolutny,X
        inx
        bne wypeln

        ldy #0
kopiuj  lda bufor,y     ; absolutny,Y
        sta ($58),y     ; pośredni,Y
        iny
        bne kopiuj
        jmp *
```

Dlaczego Y w drugiej pętli: zapis `($58),y` wymaga Y, a odczyt może używać tego samego rejestru w trybie `absolutny,Y`. Jeden indeks obsługuje oba adresy.

Zadanie 4. Drugi wiersz

```
wsk     = $cb

start   clc
        lda $58
        adc #40         ; młodszy bajt + 40
        sta wsk
        lda $59
        adc #0          ; starszy + ewentualne przeniesienie
        sta wsk+1

        lda #33
        ldy #0
        sta (wsk),y
        jmp *
```

Dlaczego `adc #0`: jeśli młodszy bajt po dodaniu 40 przekroczy 255, przeniesienie w C trzeba doliczyć do starszego bajtu. Kopiowanie i dodawanie zrobione są tu w jednym kroku.

Zadanie 5. Przekątna

```
wsk     = $cb

start   lda $58
        sta wsk
        lda $59
        sta wsk+1
        ldx #24         ; 24 wiersze
        ldy #0
loop    lda #33
        sta (wsk),y
        clc
        lda wsk
        adc #41         ; wiersz niżej (40) + kolumna w prawo (1)
        sta wsk
        bcc ok
        inc wsk+1
ok      dex
        bne loop
        jmp *
```

Dlaczego tak: `BCC` + `INC` to krótsza wersja dodawania przeniesienia do starszego bajtu. `lda #33` stoi wewnątrz pętli, bo dodawanie do wskaźnika używa A.

Zadanie 6. Dwie pionowe linie

```
wsk     = $cb

start   lda $58
        sta wsk
        lda $59
        sta wsk+1
        ldx #24
loop    lda #124        ; wewnątrz pętli!
        ldy #0
        sta (wsk),y     ; kolumna 0
        ldy #39
        sta (wsk),y     ; kolumna 39
        clc
        lda wsk
        adc #40         ; następny wiersz
        sta wsk
        bcc ok
        inc wsk+1
ok      dex
        bne loop
        jmp *
```

Podział ról: wskaźnik wybiera wiersz, Y wybiera kolumnę w wierszu. Tak wygląda większość procedur rysujących po ekranie.

Zadanie 7. Kopia wiersza

```
src     = $cb
dst     = $cd

start   ldy #4
napis   lda tekst,y
        sta ($58),y
        dey
        bpl napis

        lda $58         ; src = pierwszy wiersz
        sta src
        lda $59
        sta src+1

        clc             ; dst = ekran + 920
        lda $58
        adc #<920
        sta dst
        lda $59
        adc #>920
        sta dst+1

        ldy #39
kopiuj  lda (src),y
        sta (dst),y
        dey
        bpl kopiuj
        jmp *

tekst   dta d'ATARI'
```

Uwaga: jako źródła można by użyć bezpośrednio `($58),y`, bez wskaźnika `src`. Wersja z dwoma własnymi wskaźnikami jest wzorcem do kolejnych zadań.

Zadanie 8. Zapamiętaj i odtwórz ekran

```
src     = $cb
dst     = $cd
DLUG    = 960
BUFOR   = $4000
```

```
start   lda $58         ; 1. ekran -> bufor
        sta src
        lda $59
        sta src+1
        lda #<BUFOR
        sta dst
        lda #>BUFOR
        sta dst+1
        jsr kopiuj

        lda $58         ; 2. wypełnij ekran
        sta dst
        lda $59
        sta dst+1
        lda #33
        ldy #0
        ldx #>DLUG
wyp1    sta (dst),y
        iny
        bne wyp1
        inc dst+1
        dex
        bne wyp1
        ldx #<DLUG
wyp2    sta (dst),y
        iny
        dex
        bne wyp2

czekaj  lda $d01f       ; 3. czekaj na START
        and #1
        bne czekaj

        lda #<BUFOR     ; 4. bufor -> ekran
        sta src
        lda #>BUFOR
        sta src+1
        lda $58
        sta dst
        lda $59
        sta dst+1
        jsr kopiuj
        jmp *

; --- kopiuje DLUG bajtów z (src) do (dst) ---
kopiuj  ldy #0
        ldx #>DLUG      ; pełne strony
```

```
        beq reszta
strona  lda (src),y
        sta (dst),y
        iny
        bne strona
        inc src+1
        inc dst+1
        dex
        bne strona
reszta  ldx #<DLUG      ; reszta
        beq koniec
ogon    lda (src),y
        sta (dst),y
        iny
        dex
        bne ogon
koniec  rts
```

Na co uważać: procedura przesuwa `src+1` i `dst+1`, więc przed drugim wywołaniem oba wskaźniki są ustawiane od nowa. Ta sama procedura kopiuje w obie strony, bo adresy dostaje we wskaźnikach, a nie na stałe.

Zadanie 9. Przesunięcie ekranu w prawo

Wynik eksperymentu: przy kopiowaniu od początku pierwszy znak ekranu „rozlewa się" na cały ekran. Bajt 0 trafia do bajtu 1, potem ten sam (już nadpisany) bajt 1 trafia do bajtu 2 i tak dalej.

Wersja poprawna, od końca:

```
src     = $cb
dst     = $cd
DLUG    = 959

start   lda $58         ; src = ekran
        sta src
        lda $59
        sta src+1

        clc             ; dst = ekran + 1
        lda $58
        adc #1
        sta dst
        lda $59
        adc #0
        sta dst+1

        clc             ; oba wskaźniki na ostatnią,
        lda src+1       ; niepełną stronę bloku
        adc #>DLUG
        sta src+1
        clc
        lda dst+1
        adc #>DLUG
        sta dst+1

        ldy #<DLUG      ; końcówka, od tyłu
        beq pelne
ogon    dey
        lda (src),y
        sta (dst),y
        cpy #0
        bne ogon

pelne   ldx #>DLUG      ; pełne strony, w dół
        beq koniec
strona  dec src+1
        dec dst+1
kopia   dey             ; z 0 na $FF = koniec strony
        lda (src),y
        sta (dst),y
        cpy #0
        bne kopia
        dex
        bne strona
koniec  jmp *
```

Dlaczego tak:

- `DEY` stoi przed kopiowaniem, więc pętla obsługuje indeksy od `długość-1` do 0.
- `CPY #0` jest potrzebne, bo `STA` nie ustawia znaczników, a `LDA` ustawia je według danych, nie według Y.
- Po pętli `ogon` Y równa się 0. Pierwsze `DEY` w pętli `kopia` daje `$FF`, czyli ostatni bajt poprzedniej strony.

Zasada do zapamiętania: cel wyżej niż źródło → kopiuj od końca. Cel niżej → od początku.

Zadanie 10. Tryb `(zp,X)`

```
start   lda $58         ; wskaźnik 1: pierwszy wiersz
        sta $cb
        lda $59
        sta $cc

        clc             ; wskaźnik 2: drugi wiersz
        lda $58
        adc #40
        sta $cd
        lda $59
        adc #0
        sta $ce

        ldx #0
        lda #33
        sta ($cb,x)     ; przez wskaźnik $CB/$CC

        ldx #2
        lda #34
        sta ($cb,x)     ; przez wskaźnik $CD/$CE
        jmp *
```

Dlaczego X = 2: wskaźnik zajmuje 2 bajty, więc kolejne wskaźniki leżą co 2 komórki. X wybiera tu wskaźnik, a nie bajt w danych.

Dla chętnych – wskazówka do przesunięcia w lewo

Źródło to ekran + 1, cel to ekran, długość 959. Cel leży **niżej** niż źródło, więc właściwe jest zwykłe kopiowanie od początku, czyli procedura `kopiuj` z zadania 8 ze zmienioną długością.