# Rozdział 3

# Architektura komputerów i język assemblera

## 3.1 Przed powstaniem pierwszego komputera

O tym co to znaczy, że jakiś problem jest obliczalny, tj. można rozwiązać go wykonując obliczenia automatycznie bez udziału człowieka, zastanawiano się zanim powstały pierwsze komputery.

Jednym z pierwszych, który teoretycznie rozważał ten problem był Alan Turing (1912-1954).

Zdefiniował on w roku 1936 maszynę abstrakcyjną, nazywaną od jego nazwiska, maszyną Turinga.

Maszyna ta składa się z nieskończonej taśmy podzielonej na pola, w których można zapisywać litery z pewnego skończonego alfabetu $\Sigma$.

Nad taśmą porusza się głowica, która może odczytywać i zapisywać litery na taśmie.

Maszyna może znajdować się w jednym ze skończenie wielu stanów $S = \{s_0, s_1, \ldots, s_{n-1}\}$.

Działanie maszyny opisane jest w postaci programu. Program składa się z nieuporządkowanego zbioru instrukcji. Każda instrukcja jest uporządkowaną piątką

$$\langle s, a, b, r, t \rangle, \tag{3.1}$$

gdzie $s, t \in S$, $a, b \in \Sigma \cup \{\text{␣}\}$, $r \in \{L, R\}$ jest ruchem głowicy (znak ␣to pusty symbol na taśmie oznaczający brak litery w danym miejscu).

Instrukcję 3.1 należy rozumieć w następujący sposób:

> *Jeśli maszyna będąc w stanie $s$ widzi pod głowicą symbol $a$, to głowica zapisuje na taśmie symbol $b$, następnie wykonuje głowicą ruch w lewo albo w prawo w zależności od $r$ i przechodzi do stanu $t$.*

Jeśli w danym stanie maszyna widzi symbol, któremu nie odpowiada żadna instrukcja, to kończy ona swoje działanie.

Tablica 3.1: Program zwiększający o 1 liczbę w zapisie dwójkowym.

| | | | | |
|---|---|---|---|---|
| $s_0$ | 0 | 1 | R | $s_1$ |
| $s_0$ | 1 | 0 | L | $s_0$ |
| $s_1$ | 0 | 0 | R | $s_1$ |
| $s_1$ | 1 | 1 | R | $s_1$ |
| $s_1$ | ␣ | ␣ | L | $s_1$ |

Program wygodnie zapisywać jest w postaci tabelki o liczbie wierszy równej licznie instrukcji a liczbie kolumn równej 5 (tyle co składowych instrukcji).

### 3.1.1 Przykład maszyny Turinga

Rozpatrzmy następujący problem. Na taśmie zapisana jest liczba w systemie dwójkowym za pomocą alfabetu $\Sigma = \{0, 1\}$.

Początkowo głowica znajduje się nad najmniej znaczącym bitem liczby (pierwszy od prawej strony) a maszyna jest w stanie $s_0 \in S$, gdzie zbiór $S$ składa się z trzech stanów $\{s_0, s_1, s_2\}$.

Po wykonaniu programu maszyna zatrzymuje się w stanie $s_2$ obserwując najmniej znaczący bit liczby zapisanej na taśmie. Liczba różni się od tej zapisanej początkowo, że ma wartość o jeden większą.

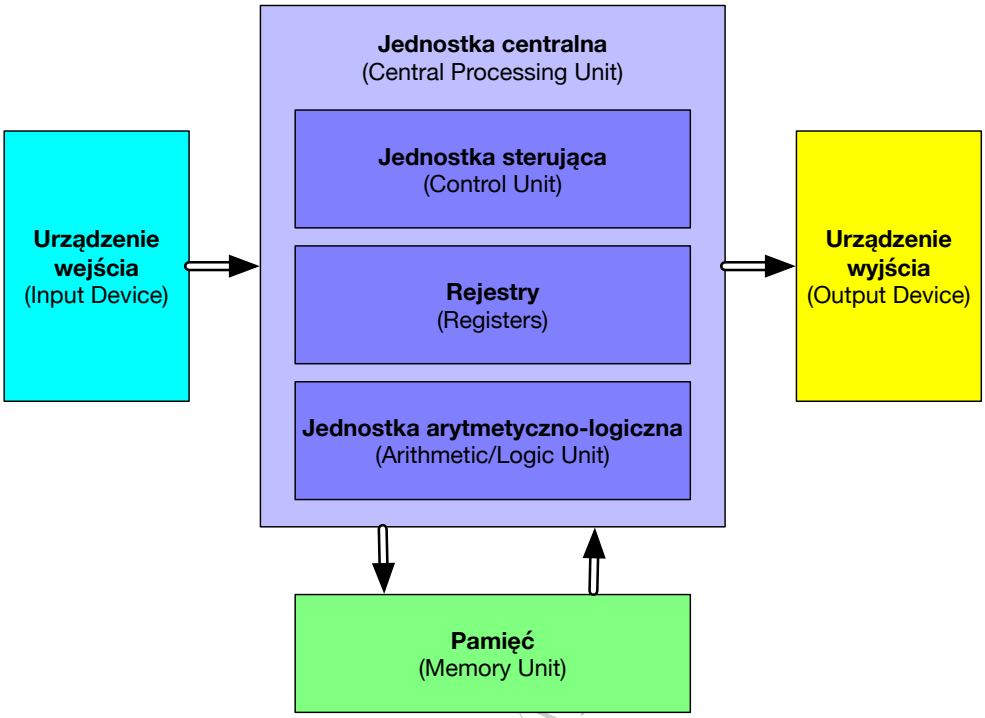
Działanie maszyny można prześledzić na następującym przykładzie (między pionowymi kreskami umieszczono symbol widoczny pod głowicą):

| Stan | Taśma | | |
|---|---|---|---|
| $s_0$ | ␣1011 | 1 | ␣ |
| $s_0$ | ␣101 | 1 | 0␣ |
| $s_0$ | ␣10 | 1 | 00␣ |
| $s_0$ | ␣1 | 0 | 000␣ |
| $s_1$ | ␣11 | 0 | 00␣ |
| $s_1$ | ␣110 | 0 | 0␣ |
| $s_1$ | ␣1100 | 0 | ␣ |
| $s_1$ | ␣11000 | ␣ | |
| $s_2$ | ␣1100 | 0 | ␣ |

Maszyna zakończyła pracę w stanie $s_2$ i jak widać liczba $(10111)_2 = 23$ została przekształcona w liczbę o jeden większą $(11000)_2 = 24$.

## 3.2 Uniwersalność komputera

Maszyna Turinga jest abstrakcyjnym modelem obliczeniowym. Należy pamiętać, że program jest nierozerwalną częścią maszyny, zatem inny program, to inna




Rysunek 3.1: Architektura von Neumanna.

maszyna.

Można rozważać teoretycznie również tzw. uniwersalne maszyny Turinga, w których program jest zapisany obok danych na taśmie a maszyna wykonuje go przekształcając dane. Programem takiej maszyny jest więc interpreterem programu na maszynę Turinga zapisanego symbolicznie na taśmie.

Okazuje się, że taki teoretyczny model stał się podstawą konstrukcji współczesnych komputerów.

W pamięci komputerów przechowywane są nie tylko dane ale i programy, które na nich operują.

Komputer jest więc maszyną uniwersalną. Potrafi rozwiązywać każdy problem, dla którego można podać program rozwiązujący go.

### 3.2.1 Architektura von Neumanna

John von Neumann (1903-1957), opierając się na koncepcji uniwersalnej maszyny Turinga, zaproponował w roku 1945 architekturę komputera, która do dziś jest stosowana. Nazywana jest ona od nazwiska autora *architekturą von Neumanna* albo od nazwy uczelni gdzie ją opracował *architekturą Princeton*.

Architekturę von Neumanna przedstawiono na 3.1. Składa się na nią:

- jednostka centralna (CPU),
- pamięć,
- urządzenia wejścia,
- urządzenia wyjścia.

W jednostce centralne znajduje się jednostka sterująca, która po pobraniu z pamięci kodu instrukcji, analizuje ten kod i odpowiednio steruje CPU celem poprawnego wykonania tej instrukcji.

Obok jednostki sterującej jest jednostka arytmetyczno-logiczna (ALU), która potrafi wykonać poprawnie operacje arytmetyczne oraz logiczne na danych znajdujących się w rejestrach przechowujących dane wewnątrz jednostki sterującej lub danych pobranych z pamięci.

Zmienione dane mogą zostać odłożone do rejestrów lub do pamięci.

Jednostka centralna może pobierać dane z urządzeń wejściowych oraz odkładać je do urządzeń wyjściowych.

Dostęp do pamięci jest powolny dlatego część danych przechowuje się wewnątrz CPU w tzw. rejestrach.

Rejestry dzieli się na:

- rejestry danych
- rejestry adresowe
- rejestr licznika programu PC (ang. program counter)
- rejestr wskaźnika stosu SP (ang. stack pointer)



W niektórych architekturach przyjmuje się ciekawe rozwiązanie, w którym odwzorowuje się urządzenia wejścia/wyjścia w pamięci.

Dla przykładu w mikrokontrolerze ATMEGA168 najmniej znaczący bit wartości zapamiętanej pod adresem 5 odpowiada stanowi na 14. nóżce procesora (0 gdy 0V, 1 gdy +5V).

## 3.3 Assembler

Program maszynowy jest ciągiem kodów instrukcji. Aby łatwiej było pisać programy zamiast kodów stosuje się *mnemoniki* czyli krótkie nazwy instrukcji.

Język assemblera oprócz mnemoników instrukcji posiada również nazwy rejestrów, adresy, adresy symboliczne (etykiety) i dyrektywy (polecenia sterujące programem assemblera tłumaczącym instrukcje symboliczne na kody instrukcji w języku maszynowym).

### 3.3.1 Mikroprocesor MOS Technology 6502

Mikroprocesor 6502 firmy MOS Technology był bardzo popularny i spotykało się go w takich mikrokomputerach jak np. Apple I i II, Atari czy Commodore oraz konsolach Nintendo.

Na rysunku 3.1, prezentującym architekturę mikroprocesora 6502, przedstawiono, poza magistralami danych i adresową, jednostkę arytmetyczno logiczną oraz rejestry:

- ośmiobitowy akumulator $Acc$
- ośmiobitowy rejestr indeksowy $X$
- ośmiobitowy rejestr indeksowy $Y$
- ośmiobitowy wskaźnik stosu $S$
- ośmiobitowy rejestr znaczników stanu $P$
- szesnastobitowy licznik programu $PC$ (podzielony na dwa rejestry ośmiobitowe $PCL$ i $PCH$)



Szyna adresowa jest szesnastobitowa. Pozwala to zaadresować $64KB$ pamięci (adresy od `0x0000` do `0xFFFF`).

Pierwsze 256 bajtów pamięci o adresach od `0x0000` do `0x00FF`, tzw. strona zerowa, służą za 256 szybkich rejestrów ośmiobitowych. Instrukcje odwołujące się do nich mają tylko dwa bajty długości (argument będący adresem jest jednobajtowy).

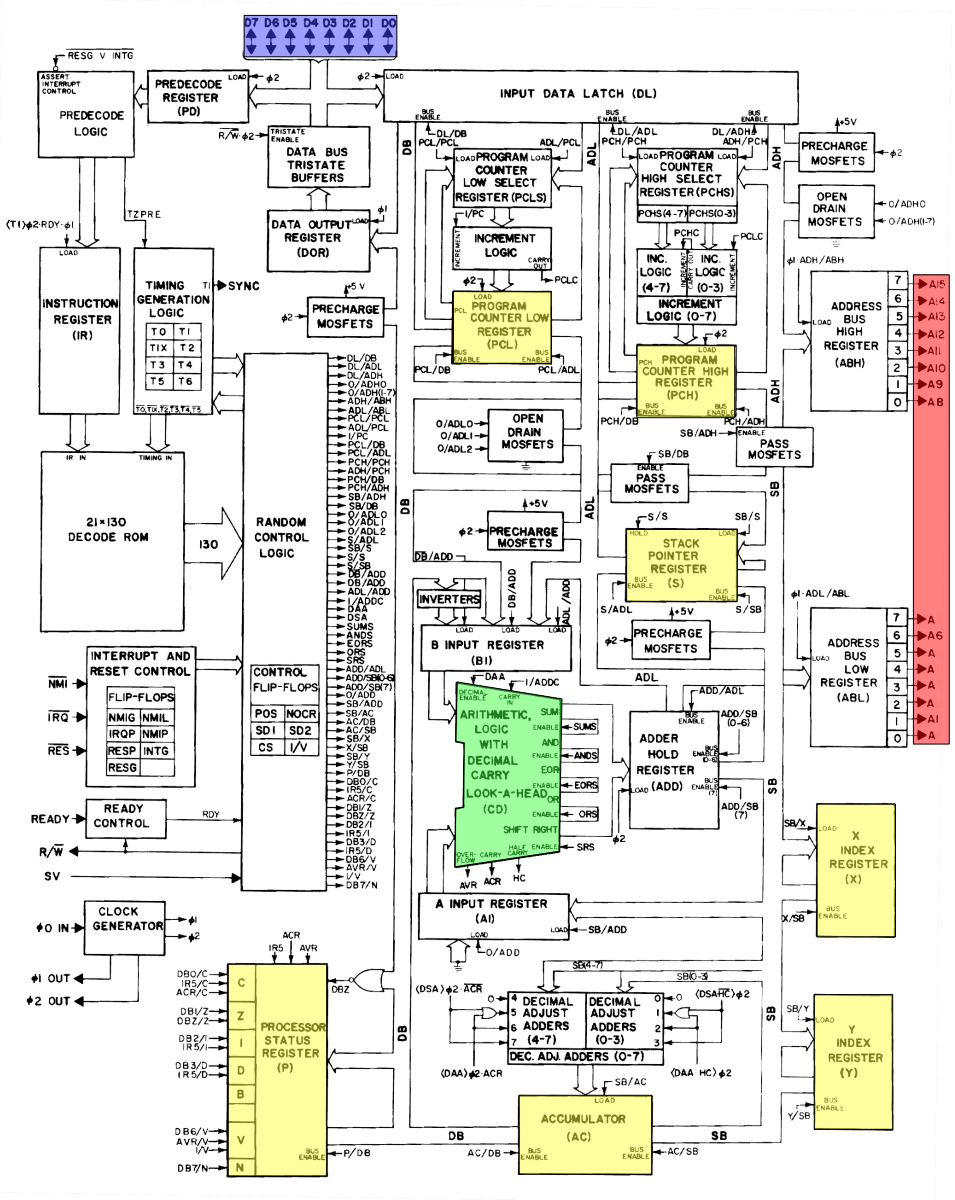
Kolejne 256 bajtów pamięci o adresach od `0x0100` do `0x01FF`, to obszar przeznaczony na stos.

Szczególną rolę odgrywają dwa bajty pamięci pod adresami `0xFFFC` i `0xFFFD`, gdyż zapisany jest w nich adres od którego rozpoczyna się wykonywanie programu.

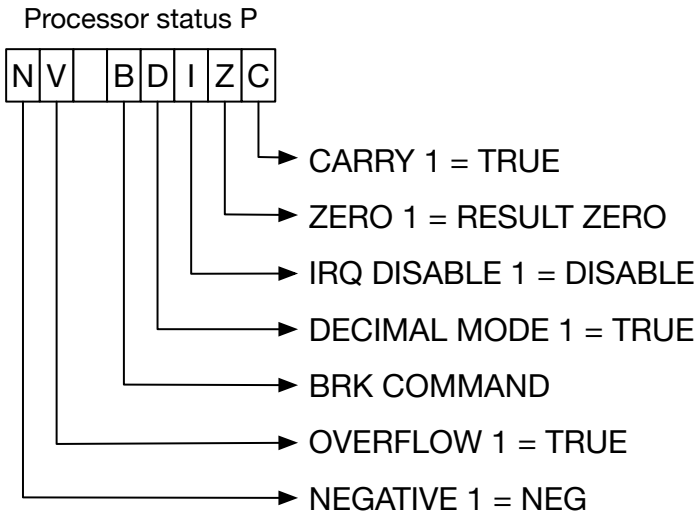
Po włączeniu lub zresetowaniu procesora, zawartość komórki pamięci pod adresem `0xFFFC` zapisywana jest w rejestrze $PCL$, natomiast zawartość pamięci pod adresem `0xFFFD` zapisywana jest w rejestrze $PCH$. Obie te wartości stanowią łącznie szesnastobitowy adres instrukcji, która wykonywana jest jako pierwsza.

Rejestr akumulatora $Acc$ służy do przechowywania jednego z argumentów oraz wyniku operacji.

Za pomocą rejestrów $X$ i $Y$ możliwe jest indeksowanie danych tj. traktowanie kolejnych bajtów pamięci tablicy złożonej z szeregu ośmiobitowych wartości. Jeśli $Adr$ jest adresem początku obszaru zajmowanego przez tablicę $T$ a wartością rejestru $X$ jest liczba $i$, to pod adresem równym $Adr + X$, co zapisujemy w assemblerze jako $Adr, X$, znajduje się element $T[i]$.




Rysunek 3.2: Architektura mikroprocesora 6502: kolorem żółtym oznaczono rejestry, kolorem zielonym ALU, kolorem niebieskim magistralę danych a kolorem czerwonym magistralę adresową (źródło: Apple II Things).




Rysunek 3.3: Bity rejestru P i ich znaczenie.



Rejestr $S$ służy za wskaźnik stosu. Stos znajduje się na pierwszej stronie pamięci RAM o adresach od `0x0000` do `0x00FF`.

Na rysunku 3.3 przedstawiono rejestr statusu procesora $P$. W kolejnych jego bitach przechowywane są znaczniki wyrażające:

- $N$ Czy wynik operacji był ujemny?
- $V$ Czy wynik operacji przekroczył dopuszczalny zakres?
- $B$ Czy wykonano komendę `BRK` przerywającą działanie?
- $D$ Czy procesor działa w trybie arytmetyki dziesiętnej?
- $I$ Czy są zablokowane przerwania?
- $Z$ Czy wynik operacji był równy zero?
- $C$ Czy jest ustawiony bit przeniesienia (pożyczki dla odejmowania)?

## 3.4 Programowanie w języku assemblera

Dla uproszczenia zapisu, przyjmujemy następujące oznaczenia:

- $B(Adr)$ ośmiobitowa wartość zapisana w pamięci RAM pod adresem $Adr$
- $W(Adr)$ szesnastobitowa wartość zapisana w pamięci RAM pod adresem $Adr$ i $Adr + 1$ (równa $B(Adr) + 256 \cdot B(Adr + 1)$).

Po znaku średnika można umieszczać w wierszu komentarz.

Mnemoniki muszą być poprzedzone co najmniej jedną spacją.

Od pierwszej kolumny może rozpoczynać się nazwa etykiety tj. symbolicznej nazwy adresu pamięci, w którym rozpoczyna się instrukcja z danego wiersza.

### 3.4.1 Kopiowanie danych

Instrukcja `LDA` (ang. *LoaD Accumulator*) służy do załadowania ośmiobitowej danej do rejestru akumulatora. Możliwe argumenty, to

- LDA #N ; załadowanie liczby $N$
- LDA Adr ; załadowanie wartości $B(Adr)$ do akumulatora

Instrukcja `STA` (ang. *STore Accumulator*) służy do zapisania wartości z akumulatora do pamięci. Możliwe argumenty, to:

- STA Adr ; zapisanie akumulatora do B(Adr)



**B(A2) = B(A1)**

```
LDA A1
STA A2
```

**W(A2) = W(A1)**

```
LDA A1
STA A2
LDA A1+1
STA A2+1
```

### 3.4.2 Dodawanie

Instrukcja `CLC` (ang. CLear Carry) ustawia bit przeniesienia na 0.

Instrukcja `ADC` (ang. ADd with Carry) dodaje do akumulatora ośmiobitową wartość. Możliwe argumenty, to

- ADC #N ; dodanie do akumulatora bitu przeniesienia i wartości $N$
- ADC Adr ; dodanie do akumulatora bitu przeniesienia i wartości $B(Adr)$

Instrukcja `INC Adr` zwiększa o jeden ośmiobitową wartość pod adresem $Adr$.

Instrukcje `INX` i `INY` zwiększają o jeden, odpowiednio, wartość rejestru $X$ i wartość rejestru $Y$.

Instrukcja `BCC` (ang. *Branch on Carry Clear*) wykonuje skok do instrukcji z podaną etykietą, jeśli przeniesienie Carry jest równe 0.

**B(A3) = B(A1) + B(A2)**

```
LDA A1
CLC
ADC A2
STA A3
```

**W(A3) = W(A1) + W(A2)**

```
LDA A1
CLC
ADC A2
STA A3
LDA A1+1
ADC A2+1
STA A3+1
```

Zwróć uwagę, że bit przeniesienia jest zerowany tylko przed pierwszym dodawaniem.



**B(A4) = B(A1) + B(A2) + B(A3)**

```
LDA A1
CLC
ADC A2
ADC A3
STA A4
```

Zwróć uwagę, że bit przeniesienia jest zerowany tylko przed pierwszym dodawaniem.

**B(A2) = B(A1) + 20**

```
LDA A1
CLC
ADC #20
STA A2
```

**W(A1) = W(A1) + B(A2)**

```
LDA A1
CLC
ADC A2
STA A1
BCC Z4
INC A1+1
Z4
```

### 3.4.3 Odejmowanie

Instrukcja `SEC` ang. *SEt Carry*) ustawia bit przeniesienia na wartość 1. Wymagane jest to do poprawnego wykonania operacji odejmowania. W przypadku odejmowania bit Carry pełni rolę „pożyczki".

**Co to jest pożyczka?** Najprościej wytłumaczyć to na przykładzie odejmowania czerocyfrowych liczb binarnych w kodzie uzupełnieniowym U2.

Różnica cyfr może być dodania, równa zero albo ujemna. W trzecim przypadku, aby było możliwe odjęcie liczby większej od mniejszej, konieczne jest dopisanie na początku odjemnika cyfry 1 (pożyczki). Jeśli po odjęciu wyniki nadal z przodu posiada pożyczkę równą 1, to jest nieujemny a jeśli pożyczka zniknęła, to jest ujemny.

**Przykład 1** *Odejmiemy od* 7 *wartość* 2*:*

| *pożyczka* | | | | | |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 | = 7 |
| | 0 | 0 | 1 | 0 | = 2 |
| 1 | 0 | 1 | 0 | 1 | = 5 *pożyczka pozostała więc wynik nieujemny* |

*Odejmiemy od* 2 *wartość* 7*:*

| *pożyczka* | | | | | |
|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 0 | = 2 |
| | 0 | 1 | 1 | 1 | = 7 |
| 0 | 1 | 0 | 1 | 1 | = −5 *pożyczka zniknęła więc wynik ujemny* |



Instrukcja `SBC` (ang. *SuBtract with Carry*) odejmuje od akumulatora ośmiobitową wartość. Możliwe argumenty, to

- SBC #N ; odjęcie od akumulatora wartości $N$
- SBC Adr ; odjęcie od akumulatora wartości $B(Adr)$

Instrukcja `DEC Adr` zmniejsza o jeden ośmiobitową wartość pod adresem $Adr$.

Instrukcje `DEX` i `DEY` zmniejszają o jeden, odpowiednio, wartość rejestru $X$ i wartość rejestru $Y$.

Instrukcja `BCS` (ang. *Branch on Carry Set*) wykonuje skok do instrukcji z podaną etykietą, jeśli przeniesienie Carry jest równe 1.

**B(A3) = B(A1) - B(A2)**

```
LDA A1
SEC
SBC A2
STA A3
```

**B(A2) = B(A1) - 20**

```
    LDA A1
    SEC
    SBC #20
    STA A2
```

**B(A2) = -B(A1)**

```
    LDA #0
    SEC
    SBC A1
    STA A2
```

**W(A1) = W(A1) - B(A2)**

```
    LDA A1
    SEC
    SBC A2
    STA A1
    BCS Z5
    DEC A1+1
Z5
```



### 3.4.4 Indeksowanie

Rejestry $X$ i $Y$ można załadować danymi za pomocą instrukcji, odpowiednio, `LDX` i `LDY`.

Za pomocą instrukcji `STX` i `STY` można odłożyć wartość z rejestru, odpowiednio, $X$ i $Y$ do pamięci.

Za pomocą następujących instrukcji można dokonywać przesłania wartości między rejestrami *Acc*, $X$ i $Y$:

- `TAX` wykonuje przesłanie z *Acc* do $X$
- `TXA` wykonuje przesłanie z $X$ do *Acc*
- `TAY` wykonuje przesłanie z *Acc* do $Y$
- `TYA` wykonuje przesłanie z $Y$ do *Acc*

Przykład optymalizacji kodu. Zamiast pisać:

```
LDA #$01
LDY #$01
```

lepiej napisać:

```
LDA #$01
TAY
```

Drugi fragment kodu zajmuje w pamięci 3 bajty zamiast 4.

Za pomocą instrukcji `CPX` oraz `CPY` można porównać wartość rejestru, odpowiednio $X$ i $Y$, z podaną wartością.

Jeśli porównano ostatnio dwie równe wartości albo wynik był równy 0, to ustawia się w rejestrze $P$ flaga $Z$ i można wykryć tę sytuację wykonując skok warunkowy `BEQ` (ang. Branch on EQual).

Jeśli porównano ostatnio dwie różne wartości albo wynik był różny od 0, to kasowana jest w rejestrze $P$ flaga $Z$ i można wykryć tę sytuację wykonując kok warunkowy `BNE` (anf Branch on Not Equal).

**LET L = T(J + K)**

```
LDA J
CLC
ADC K
TAX
LDA T,X
STA L
```



**FOR J = 1 TO N: LET T(J) = 0: NEXT J**

```
      LDX #0
      TXA
LOOP: INX
      STA T,X
      CPX N
      BNE LOOP
```

**Skok do etykiety DIFFER gdy T(J) <> U(J)**

```
LDX J
LDA T,X
CMP U,X
BNE DIFFER
```

### 3.4.5 Drukowanie znaków

Poniżej przedstawiono przykład programu drukującego napis `Hello, world!`.

```
POTCHAR =   $1000
        *=  $0600
START   LDX #$00
LOOP    LDA MSG,X
        BEQ STOP
        JSR PUTCHAR ; wydrukowanie znaku o kodzie w Acc
        INX
        JMP LOOP
STOP    BRK
MSG     .ASC "Hello, world!", 10, 0
```

Załóżmy, że jego źródła umieszczono w pliku `hello.asm`.

Pod systemem `Linux` można skompilować go poleceniem `xa hello.asm`.

Aby uruchomić przekład jaki znalazł się w pliku `a.o65` należy wykonać następujące polecenie:

```
$ run6502 -l 600 a.o65 -R 600 -P 1000 -X 0
Hello, world!
$
```



### 3.4.6 Zabawa assemblerem w przeglądarce

Na stronie `https://skilldrick.github.io/easy6502/` można poćwiczyć programowanie w języku assemblera na procesor 6502. Po wpisaniu programu należy kliknąć na przycisk **Assemble** i gdy nie ma błędów, to uruchomić klikając na przycisku **Run**.

Aktualne wartości rejestrów drukowane są w prawym dolnym rogu okienka.

Na rysunku 3.4 przedstawiono przykłady dwóch programów. Pierwszy dodaje w trybie binarnym a drugi w trybie dziesiętnym.

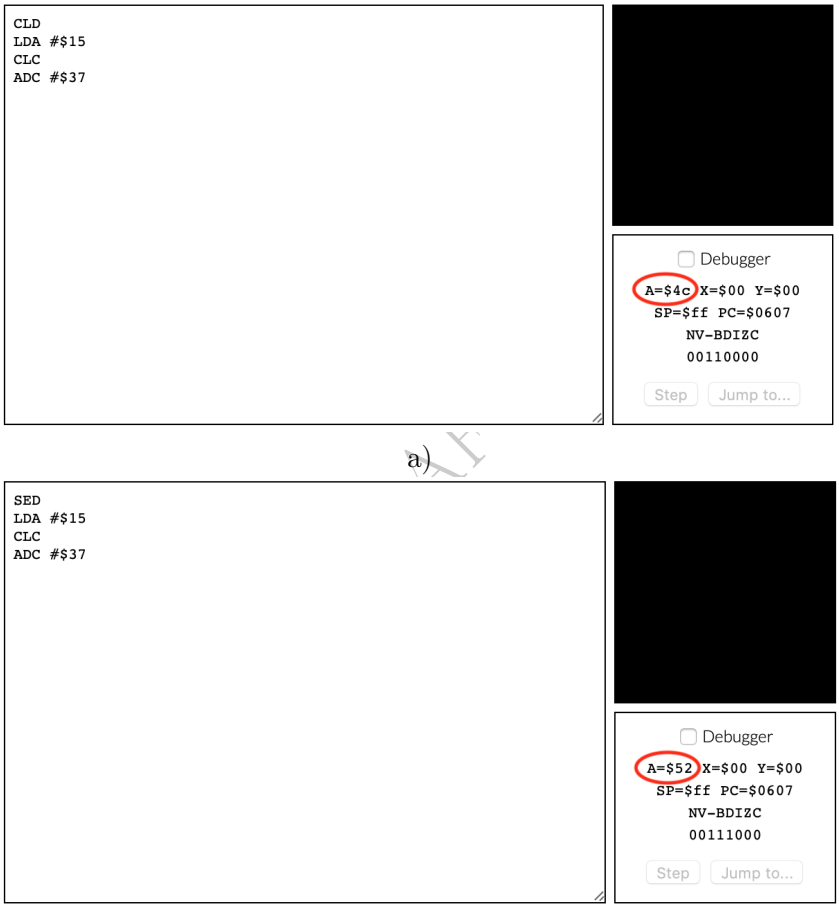
Kiedy wyłączono tryb dziesiętny (instrukcja `CLD`), dodawanie i odejmowanie wykonywane jest binarnie.

Gdy włączono tryb dziesiętny (instrukcja `SED`), argumenty dodawania i odejmowania oraz wynik operacji traktowany jest jako dwucyfrowa liczba dziesiętna. Na mniej znaczących czterech bitach zapisywana jest cyfra 0..9 odpowiadająca jednościom a na bardziej znaczących czterech bitach zapisana jest cyfra 0..9 odpowiadająca dziesiątkom. Zatem na ośmiu bitach zapisane są liczby od 0 do 99.

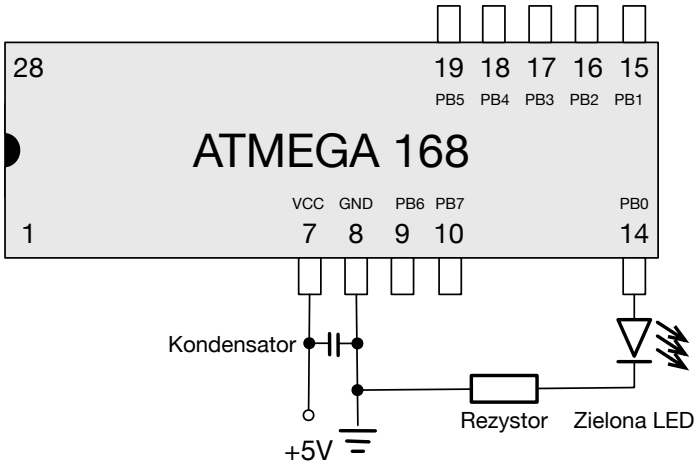
## 3.5 Wejście/wyjście ATMEGA168

Mikrokontroler ATMEGA168 posiada trzy banki pinów $B$, $C$ i $D$,

Z każdym bankiem związany jest:




Rysunek 3.4: Dodawanie w trybie: a) binarnym i b) dziesiętnym.




Rysunek 3.5: Schemat podłączenia mikrokontrolera ATMEGA168 z zasileniem i zieloną LED.



- ośmiobitowy rejestr kierunku danych: $DDRB$ pod adresem `0x04`, $DDRBC$ pod adresem `0x07` i $DDRD$ pod adresem `0x0A`;
- ośmiobitowy rejestr wyjścia: $PORTB$ pod adresem `0x05`, $PORTC$ pod adresem `0x08` i $PORTD$ pod adresem `0x0B`.

Jeśli $i$-ty bit, dla $i = 0, 1, \ldots, 7$, w rejestrze $DDRx$, dla $x = B, C, D$, ma wartość 1 (konfiguracja wyjściowa), to rejestr $PORTx$ może sterować nastawą $i$-tego pinu w porcie $PORTx$. Wpisując na $i$-tą pozycję rejestru $PORTx$ wartość 1 ustawiamy wysoki poziom $+5V$ na nóżce mikrokontrolera odpowiadającej temu bitowi. Natomiast wpisując 0 ustawiamy niski poziomo napięcia $0V$ na tej nóżce.

Na rysunku 3.3 przedstawiono przykładowe podłączenie mikrokontrolera ATMEGA168 do zasilenia oraz podłączenie diody świecącej do nóżki 14, odpowiadającej bitowi $PINB$ na najmniej znaczącej pozycji.

Poniższy program ustawia bit 0-wy portu $B$ jako wyjściowy (instrukcja `sbi DDRB,0`) a następnie w nieskończonej pętli na zmianę zapala i gasi diodę ustawiając wysoki (instrukcja `sbi PINB,0`) i niski (instrukcja `cbi PINB,0`) stan na nóżce 14.

Między wyłączeniem i włączeniem diody wykonuje się pętla opóźniająca (podprogram rozpoczynający się od etykiety `delay`). Czas wykonania tej pętli wynosi ok. jednej sekundy.

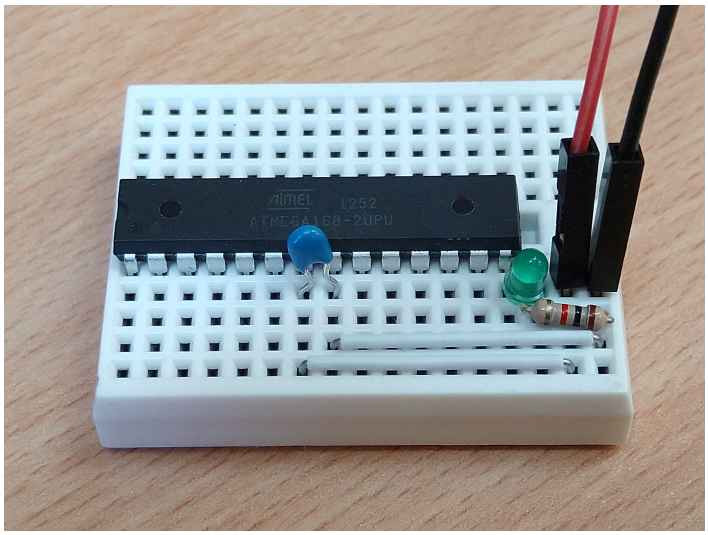
Program do odmierzania czasu korzysta z trzech rejestrów ośmiobitowych `r18`, `r19` i `r20`.

```
DDRB  = 0x04
```

```
PORTB = 0x05

.org 0
        sbi     DDRB,0

loop:
        sbi     PORTB,0
        rcall   delay
        cbi     PORTB,0
        rcall   delay
        rjmp    loop

delay:
        ldi     r18,6
        ldi     r19,19
        ldi     r20,174
L1:
        dec     r20
        brne    L1
        dec     r19
        brne    L1
        dec     r18
        brne    L1
        ret
```



Układ z rysunku 3.3 można zrealizować na prototypowej płytce montażowej w sposób przedstawiony na rysunku 3.6. Czerwonym kabelkiem podłączono napięcie $+5V$ a czarnym uziemienie.




Rysunek 3.6: Realizacja układu z rysunku 3.3.