Systemy mikroprocesorowe

Dariusz Chaberski

Uniwersytet Mikołaja Kopernika w Toruniu
Wydział Fizyki, Astronomii i Informatyki Stosowanej

Regionalne Kółka Fizyczne
Urząd Marszałkowski w Toruniu
Program Operacyjny Kapitał Ludzki

Toruń 2009

1. Wstęp

Zadaniem spotkań pod tytułem Systemy mikroprocesorowe prowadzonych w ramach Regionalnych kółek fizycznych jest zapoznanie uczniów z budową oraz działaniem systemów mikroprocesorowych. Na zajęciach zostaną przedstawione dwa różne pod względem budowy i zastosowania systemy mikroprocesorowe. Pierwszym z nich będzie system zbudowany z wykorzystaniem mikrokontrolera rodziny AVR (ATmega16), drugim systemem będzie komputer osobisty klasy PC.

Zajęcia dotyczące systemu mikroprocesorowego zbudowanego z wykorzystaniem mikrokontrolera AVR będą miały charakter praktyczny. Uczniowie z wykorzystaniem AVR Studio będą mieli możliwość rozbudowy dostarczonych programów, napisania własnych oraz dokonania ich symulacji na modelu lub też sprawdzenia działania w układzie rzeczywistym. Tej tematyce zostanie poświęconych około czterech spotkań (12 godzin lekcyjnych).

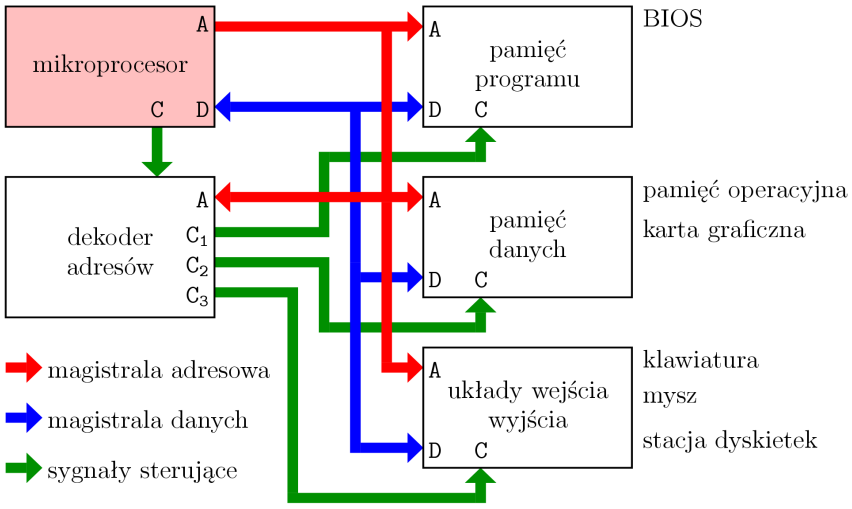
Dwa ostatnie spotkania (6 godzin lekcyjnych) dotyczyć będą budowy i działania komputerów osobistych. Zajęcia będą miały charakter prezentacji, w ramach której pokrótce przedstawiona zostania budowa i działanie procesora x86 oraz wybranych modułów znajdujących się w CHIPSETach płyt głównych komputerów osobistych. Zajęcia te będą wspomagane Asemblerem FLATASM, emulatorem systemu QEMU oraz programem MS DEBUG.

2. System Mikroprocesorowy

W skład każdego systemu mikroprocesorowego wchodzi mikroprocesor, pamięć operacyjna (dane i program), układy wejścia wyjścia oraz oprogramowanie (rys. 1). Mikroprocesor pobiera kolejne instrukcje z pamięci programu, następnie je dekoduje i wykonuje. Adres komórki pamięci, która zostanie odczytana (z której zostanie pobrana instrukcja) przekazywany jest przez magistralę adresową **A**, odczytywana instrukcja wystawiana jest przez pamięć na magistralę danych **D**. Mikroprocesor informuje pomięć o tym, że ma zostać dokonany odczyt pamięci generując odpowiednie przebiegi na magistrali sterującej **C**. Podobnie wygląda odczyt danej z pamięci danych lub odczyt z układów wejścia wyjścia. O tym jaki zasób (pamięć programu, pamięć danych, układy wejścia wyjścia) będzie odczytywany decyduje to co pojawi się na magistrali sterującej **C**.

W przypadku, kiedy mikroprocesor zapisuje do pamięci danych lub układów wejścia wyjścia na magistralę adresową wystawia adres **A**, na magistralę danych wystawia daną **D**, która ma zostać zapisana pod adres **A**, a następnie generuje odpowiednie przebiegi na magistrali sterującej **C**, które spowodują zapis danej **D** pod adres **A**, do zasobu określonego albo bezpośrednio przez stan magistrali sterującej **C** albo określonego przez wartość adresu (dekoder adresów) na magistrali adresowej **A**. Dekoder adresów w zależności od wartości adresu daje dostęp do pamięci programu, albo do pamięci danych. Można się na przykład umówić tak, że program będzie się znajdował w pamięci o adresach od 0 do 64 k - 1, a dane pod adresami od 64 k do 128 kB – 1.

Pamięć danych służy do przechowywania danych, które powstały w trakcie działania programu, mogą to być na przykład dane pomiarowe, wyniki obliczeń lub tekst wprowadzony przez użytkownika. Pamięć programu służy oczywiście do przechowywania programu, ale również można w niej przechowywać stałe liczbowe (zapis do programu jest prawie zawsze niemożliwy) lub napisy, które będą się pojawiać w menu użytkownika.



Rys. 1: Schemat blokowy podstawowego systemu mikroprocesorowego

Układy wejścia wyjścia stanowią interfejs pomiędzy mikroprocesorem a użytkownikiem. Zapewniają one dostęp do większości urządzeń wskazujących takich, jak klawiatura lub mysz. Mikroprocesor odczytując układ wejścia wyjścia otrzymuje informacje o stanie podłączonego do danego portu urządzenia. Przykładowo, gdy do starszych bitów portu podłączona jest cztero przyciskowa klawiaturka stan 0xF0 może oznaczać, że żaden przycisk nie jest wciśnięty a stan 0xB0 może oznaczać, że wciśnięty jest przycisk podłączony do bitu szóstego (numerując od zera) portu.

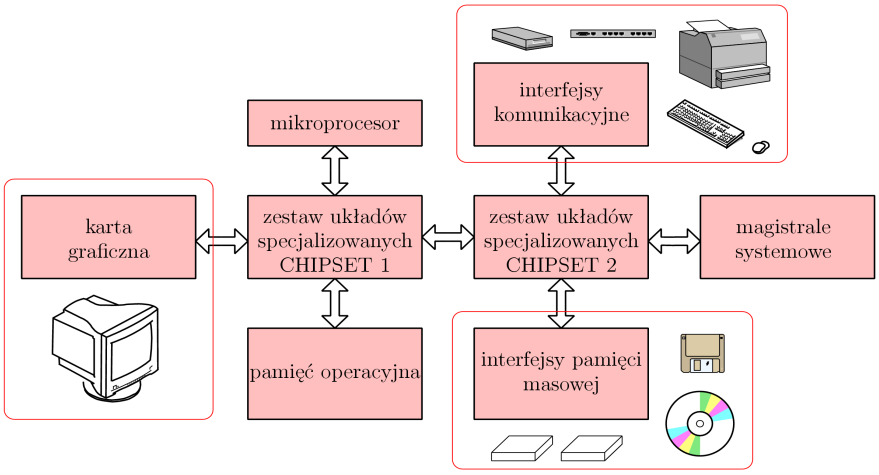
Zapis do układów wejścia wyjścia pozwala zmieniać stan urządzenia, to znaczy pozwala sterować urządzeniem. W przypadku, kiedy do portu podłączona jest linijka ośmiu diod, wysłanie do tego portu wartości 0x18 może spowodować, że będą się świeciły dwie diody środkowe, natomiast w przypadku wysłania do portu wartości 0xe0 mogą się świecić trzy diody podłączone do najbardziej znaczących bitów portu.

2.1 Podział systemów mikroprocesorowych

W zależności od zapotrzebowania stosuje się system mikroprocesorowy, który posiada najbardziej odpowiednie cechy do danego zastosowania. Chcąc napisać artykuł, przeczytać stronę www lub pograć w grę wykorzystujemy komputer osobisty, który posiada dużą moc obliczeniową dużą pamięć operacyjną, ale jednocześnie pobiera stosunkowo dużo energii. Chcą kontrolować stan obiektu (system alarmowy) możemy wykorzystać system kontrolny, którego jedynym zadaniem jest informowanie właściciela o sytuacjach alarmowych. System taki ma również działać jak najdłużej na zasilaniu bateryjnym. Chcąc zbierać duże ilości danych pomiarowych musimy system wyposażyć w dodatkową pamięć danych i dostarczyć wzorce pomiarowe. Projektowanie systemów uniwersalnych nie ma większego sensu. Urządzenia specjalizowane są tańsze i lepiej pełnią funkcję, do której je zaprojektowano.

2.1.1 Komputer osobisty

Komputer osobisty składa się z zestawu układów specjalizowanych (CHIPSETów), które zawierają wszystkie niezbędne do pracy komputera układy pomocnicze. Należą do nich kontrolery przerwań, kontrolery bezpośredniego dostępu do pamięci, liczniki a także moduły realizujące magistrale systemowe, interfejsy szeregowe i inne. CHIPSETy zapewniają komunikację pomiędzy mikroprocesorem, pamięcią oraz układami wejścia wyjścia. Urządzenia o małej szybkości transferu (np. interfejsy pamięci masowej) podłączane są do systemu z wykorzystaniem CHIPSETu 2, który nie ma bezpośredniego dostępu do mikroprocesora i pamięci. Urządzenia, które do swojej pracy wymagają dużej szybkości transmisji (np. karty graficzne) podłączane są do pierwszego zestawu układów specjalizowanych, który bezpośrednio połączony jest z mikroprocesorem i pamięcią.

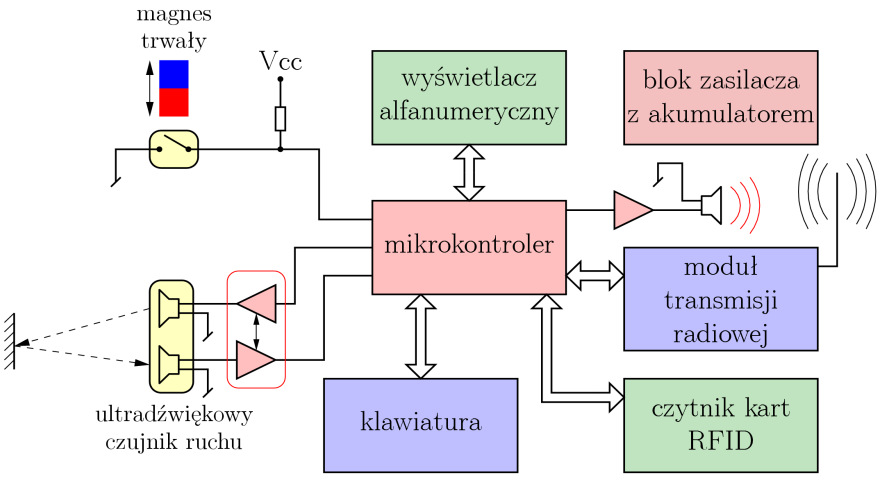


Rys. 2: Schemat blokowy typowego komputera osobistego

2.1.2 System kontrolny

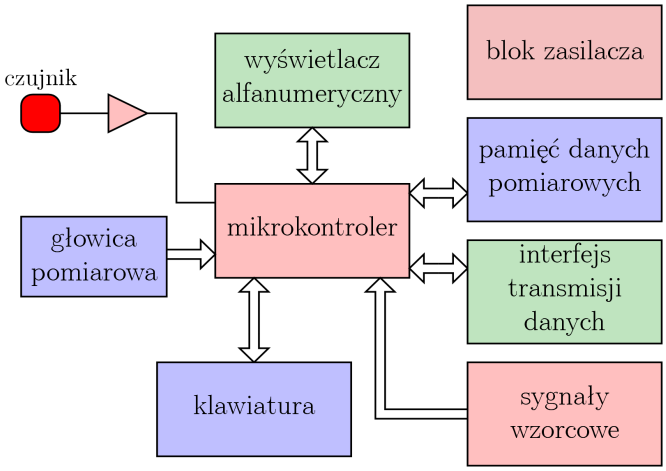
System kontrolny (rys. 3) wykorzystuje w swojej budowie mikrokontroler, który jest scalonym, niemalże kompletnym systemem mikroprocesorowym o stosunkowo małej mocy

obliczeniowej. Mikroprocesor do działania potrzebuje całego szeregu dodatkowych komponentów, natomiast mikrokontroler jest już sam w sobie systemem mikroprocesorowym i aby go uruchomić potrzebujemy tylko zasilania oraz interfejsu użytkownika (wyświetlacz alfanumeryczny, klawiatura) oraz ewentualnych czujników i modułów, których scalenie nie jest możliwe lub nie jest opłacalne. System kontrolny taki, jak system alarmowy powinien posiadać awaryjne zasilanie akumulatorowe tak, aby nie przerywać kontroli obiektu.



Rys. 3: Schemat blokowy przykładowego systemu kontrolnego

2.1.3 System pomiarowy



Rys. 4: Schemat blokowy typowego systemu pomiarowego

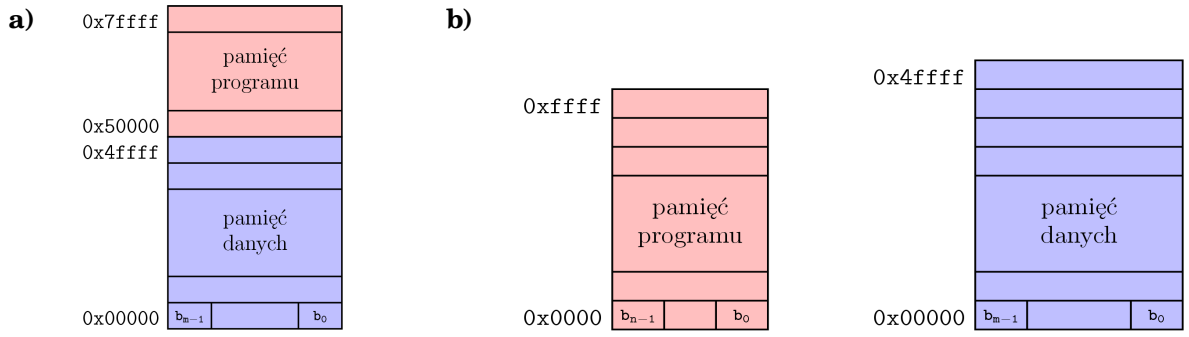
System pomiarowy może zostać zbudowany z wykorzystaniem mikrokontrolera. Oprócz interfejsu użytkownika potrzebne są tutaj również czujniki oraz głowice pomiarowe, które będą dostarczały sygnały elektryczne reprezentujące mierzone wielkości. Wykonane pomiary będą zapisywane w pamięci danych pomiarowych, którą może stanowić pamięć FLASH. Aby dokonać pomiaru należy dowiedzieć się ile jednostek wzorca zawiera mierzona wielkość. Z tego powodu

systemy pomiarowe wyposaża się w sygnały wzorcowe. Do sygnałów wzorcowych najczęściej należą napięcia oraz odcinki czasu lub częstotliwości.

2.2 Porównanie mikroprocesora i mikrokontrolera

Między mikroprocesorem i mikrokontrolerem istnieją zasadnicze różnice, które pojawiły się na drodze ewolucji obu układów. Mikrokontroler jest z reguły stosowany w systemach tanich, objętościowo małych oraz o małym poborze energii. Z tego powodu jego moc obliczeniowa jest mała, dochodzi do około 30 MIPSów dla układów ośmiobitowych. Moc obliczeniowa komputera osobistego wynosi przeciętnie od 100 do 1000 razy więcej. Dodatkowo mikroprocesor może wykonywać obliczenia na liczbach zmiennopozycyjnych, może przetwarzać jednocześnie kilka procesów na jednym rdzeniu lub posiadać zaimplementowanych kilka niezależnych rdzeni. Mikroprocesor umożliwia pracę w trybie chronionym. Oznacza to, że dostęp do zasobów, które nie są własnością danego procesu jest dla tego procesu zabroniony. W ten sposób łatwiej jest chronić system przed wirusami lub błędami programistów.

Mikroprocesor z reguły posiada architekturę Von Neumana, w której dostęp do wszystkich zasobów pamięciowych odbywa się z wykorzystaniem tych samych mechanizmów. Pamięć programu i danych umieszczone są w ciągłym i jednolitym obszarze. Co prawda program może mieścić się pod innymi adresami aniżeli dane, ale oba mają tą samą szerokość słowa i dostęp do obu zasobów odbywa się z wykorzystaniem tych samych instrukcji (rys. 5a). W architekturze harwardzkiej pamięć programu i pamięć danych są dostępne za pomocą innego zestawu instrukcji. Jest to konieczne w celu rozróżnienia dostępu do pamięci danych i pamięci programu. Rozróżnienie na podstawie adresu jest tutaj niemożliwe, gdyż adresy dla pamięci programu i pamięci danych mogą się pokrywać (z reguły oba zaczynają się od zera) (rys. 5b). W architekturze harwardzkiej szerokości słów pamięci danych i programu są różne. Szerokość słów w pamięci danych z reguły jest całkowitą krotnością bajta. Szerokości słów w pamięci programu została dobrana optymalnie do ilości kodów instrukcji mikroprocesora i może wynosić na przykład 14 bitów.



Rys. 5: Mapy pamięci dla architektur Von Neumana (a) i harwardzkiej (b)

Mikrokontroler z reguły będzie posiadał instrukcje bitowe, gdyż w układach kontrolnych i pomiarowych często zachodzi potrzeba ustawiania, zerowania lub czytania pojedynczych bitów. Współczesny mikrokontroler z reguły wyposażony jest w całą gamę peryferii takich, jak przetworniki analogowo cyfrowe i cyfrowo analogowe, liczniki, układy czasowe, interfejsy

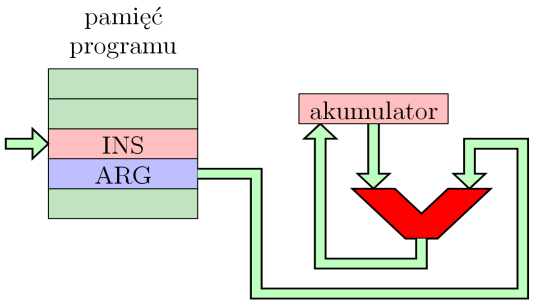
szeregowe, kontrolery przerwań.

2.3 Sposoby adresowania

Mikroprocesor wykonując instrukcje musi pobierać również argumenty dla tych instrukcji oraz zapisywać wyniki działania tych instrukcji. Przykładowo wykonując instrukcję mnożenia mikroprocesor potrzebuje takich informacji jak mnożna, mnożnik oraz miejsce dokąd ma być zapisany iloczyn.

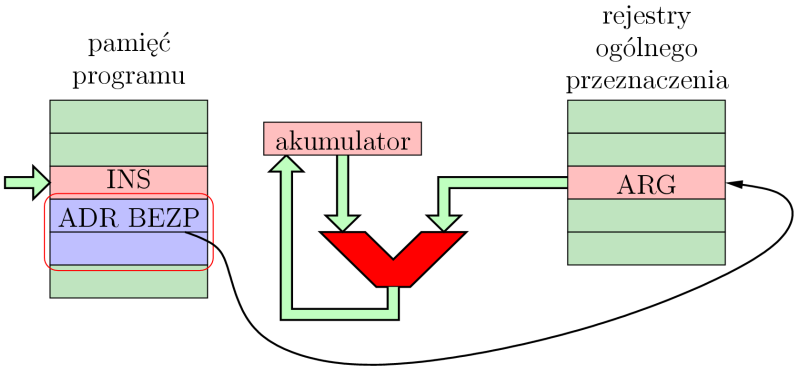
2.3.1 Instrukcja z wartością natychmiastową

Jeżeli argument jest dostarczany wraz z instrukcją, wówczas mówimy o argumencie natychmiastowym lub instrukcji z wartością natychmiastową (rys. 6). W tym przypadku argument ARG stanowi nierozłączną część instrukcji, natomiast INS stanowi rdzeń instrukcji. Do układu wykonawczego jako jeden z argumentów zostanie dostarczona wartość natychmiastowa ARG, drugim argumentem jest natomiast w tym przypadku zawartość rejestru specjalnego zwanego akumulatorem, w którym również zostanie zapisany wynik operacji.



Rys. 6: Idea instrukcji z wartością natychmiastową

2.3.2 Adresowanie bezpośrednie



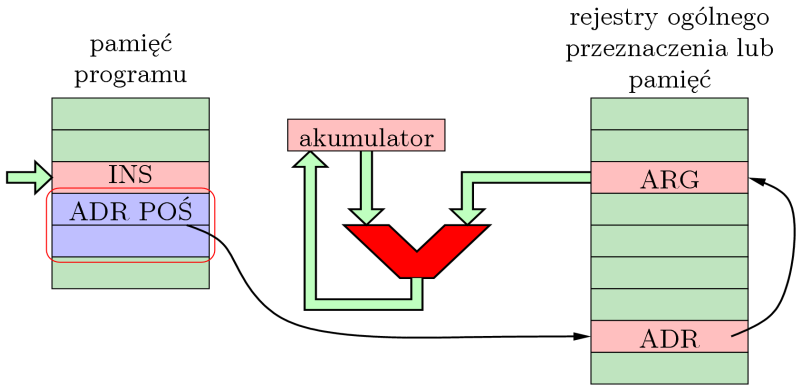
Rys. 7: Idea adresowania bezpośredniego

W tym przypadku w kodzie instrukcji nie ma argumentu tak, jak to było w poprzednim przypadku, ale znajduje się tutaj adres ADR BEZP bezpośrednio wskazujący na ten argument i

dopiero pod tym adresem znajduje się właściwy argument ARG, który zostanie wykorzystany na potrzeby wykonania instrukcji INS (rys. 7). Adres ADR BEZP przekazywany jest bezpośrednio w kodzie instrukcji, więc mówimy o adresowaniu bezpośrednim.

2.3.3 Adresowanie pośrednie

Przy adresowaniu pośrednim bezpośredni (właściwy) adres argumentu nie jest przekazywany w kodzie instrukcji. W kodzie instrukcji znajduje się jedynie informacja, która pozwala uzyskać adres argumentu. Na rysunku 8 przedstawiono diagram ilustrujący ten przypadek. W kodzie instrukcji INS zawarte jest pole ADR POŚ, które wskazuje na miejsce w pamięci lub rejestrach ogólnego przeznaczenia gdzie zawarty jest adres ADR wskazujący na argument, który ma zostać wykorzystany w kodzie instrukcji. Ilość bitów (wielkość słowa ADR POŚ) potrzebna na przekazanie informacji na temat tego w jakim rejestrze znajduje się adres ADR jest często wielokrotnie mniejsza aniżeli sam adres ADR. Do przechowywania instrukcji wykorzystujących adresowanie pośrednie potrzeba więc znacznie mniej bitów aniżeli w przypadku instrukcji, w których adres ten przekazywany jest bezpośrednio.



Rys. 8: Idea adresowania pośredniego

2.4 Podział mikroprocesorów pod względem listy instrukcji

Mikroprocesor lub mikrokontroler może posiadać tak zwaną kompletną listę instrukcji (CISC). Instrukcje w liście CISC są wysoko specjalizowane i przygotowane na każdą ewentualność. W liście tej na przykład może znaleźć się instrukcja zwiększająca o jeden zawartość komórki pamięci danych lub instrukcja przesyłania komórki pamięci do przestrzeni wejścia wyjścia.

Przeciwieństwem mikroprocesorów z kompletną listą instrukcji są mikroprocesory ze zredukowaną listą instrukcji (RISC). Mikroprocesory takie posiadają tak zwaną ortogonalną listę instrukcji. Ortogonalość w tym przypadku oznacza to, że ciężko (lub jest to niemożliwe) jest znaleźć instrukcje, które w konsekwencji będą robić to samo co instrukcja zastępowana. Z reguły mikroprocesory ze zredukowaną listą instrukcji będą posiadały tych instrukcji mniej aniżeli mikroprocesory typu CISC. W liście instrukcji nie znajdziemy na przykład instrukcji zwiększającej zawartość komórki pamięci o jeden, ale taka operacja będzie możliwa poprzez zastosowanie trzech innych instrukcji robiących łącznie to samo. W mikroprocesorze RISC taka

operacja może zostać rozbita na trzy instrukcje. Pierwsza instrukcja odczyta (do rejestru) komórkę pamięci, która ma zostać zwiększona, druga instrukcja dokona zwiększenia o jeden zawartości rejestru i w końcu trzecia operacja zapisze zawartość rejestru na miejsce zwiększanej komórki pamięci. W pseudokodzie można to zapisać następująco

```
rejestr=pamięć[adres]
rejestr=rejestr+1
pamięć[adres]=rejestr,
```

w przypadku mikroprocesora z kompletną listą instrukcji będzie to oczywiście tylko

```
pamięć[adres]=pamięć[adres]+1,
```

ale mikroprocesor będzie oczywiście miał w swoim repertuarze również instrukcje zwiększające rejestr o jeden lub odczytujące i zapisujące pamięć.

3. Podstawowe kody liczb w zapisie dwójkowym oraz operacje

Mikroprocesor wykonuje instrukcje na liczbach zapisanych w kodzie dwójkowym. Znajomość podstawowych kodów liczbowych, operacji na liczbach dwójkowych oraz ograniczeń z tym związanych jest niezbędna do efektywnego programowania zarówno w języku Asembler jak i w języku C. Przykładowo znając maksymalną wartość jaką możemy zapisać w rejestrze ośmiobitowym możemy ustalić, że ośmiobitowa zmienna (char – język C) będzie wystarczająca do implementacji licznika, który ma się zmieniać w zakresie od 0 do 32, nie potrzebujemy używać do tego od razu zmiennej 32 bitowej (long int – język C) lub 16 bitowej (short int – język C). Wpłynie to na zwiększenie szybkości działania programu oraz jednocześnie może przyczynić się do zmniejszenia kodu programu.

3.1 Kod naturalny binarny

W kodzie naturalnym binarnym (NB) cyfry otrzymują wagi 2^i, gdzie i jest indeksem bitu na miejscu którego znajduje się cyfra. Przykładowo czterobitowa liczba 1011 w zapisie dwójkowym będzie miała wartość $(1011)_2=1\cdot2^0+1\cdot2^1+0\cdot2^2+1\cdot2^3=1+2+0+8=(11)_{10}$ w systemie dziesiętnym. Minimalna wartość jaką liczba przyjmuje wynosi zero, natomiast wartość maksymalna wynosi 2^n-1, gdzie n jest liczbą bitów rejestru, w którym przechowywana jest liczba. W przypadku liczb czterobitowych zakres wynosi więc od od 0 do 15, w przypadku liczb ośmiobitowych od 0 do 255, a w przypadku liczb 32 bitowych wartość maksymalna wynosi aż 4294967295. Jak widać kod NB nadaje się do przechowywania liczb bez znaku. Z wykorzystaniem tego kodu implementuje się wszelkiego rodzaju zmienne licznikowe, kody ASCII znaków, rzadziej pomiary, gdyż te mogą przyjmować wartości ujemne.

W kodzie NB możemy zapisywać również liczby rzeczywiste stałoprzecinkowe. Jeżeli umówimy się, że mamy przecinek na pozycji 4 (pomiędzy 3 i 4 bitem numerując od zera), wówczas bit 4 będzie miał wagę 2^0, bit 5 będzie miał wagę 2^1 i tak dalej kierunku bitów o większym indeksie. Bit na pozycji 3 będzie miał jednak wagę 2^{-1}, aż wreszcie bit na pozycji 0 będzie miał wagę 2^{-4}. Wagę bitu na pozycji i można zapisać jako 2^{i-p}, gdzie p jest pozycją przecinka. W tym

przypadku oczywiście p=4. Przykładowo $(1001.0011)_2=8+1+0.125+0.0625=(9.1875)_{10}$. Oczywiście przecinek ten jest umowny i nie jest zapisywany w rejestrach mikroprocesora. Mikroprocesor nie zostaje w żaden sposób poinformowany o fakcie na którym miejscu znajduje się przecinek.

3.2 Kod uzupełnienia do 2

Istnieje wiele sposobów zapisywania liczb ze znakiem. W systemach mikroprocesorowych praktyczne zastosowanie ma jednak tylko kod uzupełnienia do 2 (U2). W kodzie tym dla n bitowej liczby poszczególne cyfry o indeksach od 0 do n-2 posiadają takie same wagi jak w kodzie NB, bit najstarszy posiada jednak wagę -2^{n-1}. Przykładowo czterobitowa liczba 1101 w systemie dwójkowym w kodzie U2 będzie miała wartość $(1101)_{U2}=1\cdot2^0+0\cdot2^1+1\cdot2^2-1\cdot2^3=1+0+4-8=(-3)_{10}$ w systemie dziesiętnym.

Dla liczb w kodzie U2 istnieje ciągłość stanów. Czterobitowe zero wynosi 0000, czterobitowa jedynka wynosi 0001, natomiast czterobitowa minus jedynka to 1111. Czterobitowa minus jedynka powstaje przez odjęcie jedynki od zera. Dzięki temu mikroprocesor może wykonywać operacje na liczbach w kodzie U2 w taki sam sposób jak operacje na liczbach w kodzie NB. Jeżeli mikroprocesor wykonuje operacje na liczbach w kodzie U2 nie jest o tym fakcie w żaden sposób informowany.

Podobnie jak liczby w kodzie NB liczby w kodzie U2 mogą przechowywać wartości niecałkowite. Wszystkie bity oprócz najstarszego otrzymują takie same wagi jak bity w kodzie stałoprzecinkowym NB. Bit najstarszy posiada wagę -2^{n-p}, gdzie p jest pozycją przecinka.

Liczby całkowite zapisane w kodzie U2 na n bitach mogą przyjmować wartości od -2^{n-1} do $2^{n-1}-1$. Przykładowo dla liczby ośmiobitowej zakres ten wynosi od -128 do 127. W przypadku stałoprzecinkowej liczby w kodzie U2, w której przecinek znajduje się na pozycji p, maksymalna wartość wynosi $2^{n-1-p}-1^{-p}$, a minimalna jest równa -2^{n-1-p}. Dla liczby ośmiobitowej, w której przecinek znajduje się na pozycji 4 liczba maksymalna wynosi więc $(0111.1111)_{U2}=(7.9375)_{10}$, a liczba minimalna to $(1000.0000)_{U2}=(-8)_{10}$.

3.3 Operacje na liczbach w kodzie NB

Podstawowymi operacjami na liczbach w kodzie NB są dodawanie i odejmowanie. Aby dodać dwie liczby binarne należy mieć na uwadze następujące reguły $0_2+0_2=0_2$, $0_2+1_2=1_2$, $1_2+0_2=1_2$ oraz $1_2+1_2=10_2$. W ostatniej regule następuje przeniesienie na bit o wadze dwukrotnie większej, gdyż nie jesteśmy w stanie w systemie dwójkowym zapisać wartości 2_{10} na pojedynczej cyfrze.

Jako przykład prześledźmy sumę dwóch liczb

```
    10011101
  + 01100111
= 1 00000100.
```

Widać w tym przykładzie, że pomimo że oba argumenty były ośmiobitowe wynik nie zmieścił się w rejestrze ośmiobitowym, nastąpiło przeniesienie. W rejestrach mikroprocesora znajduje się jeden rejestr specjalny zwany flagowym lub statusowym, który przechowuje informację o tego typu zdarzeniach. W tym konkretnym przypadku w rejestrze statusowym zostanie ustawiony znacznik przeniesienia **C**. W przypadku gdybyśmy byli w stanie zapisać (zmieścić) wynik w rejestrze

wyjściowym, wówczas znacznik **C** przyjąłby wartość 0.

Podobnie wygląda odejmowania liczb. Zasady są następujące 1-1=0, 0-0=0, 1-0=1 oraz 0-1=1 z jednoczesnym zapożyczeniem ze starszej pozycji.

Rozważmy następujący przykład

```
   10010101
 - 01101111
=  00100110.
```

W tym przykładzie odjemnik był mniejszy od odjemnej i zapożyczenie nie było konieczne. Gdyby jednak zaszła taka potrzeba mikroprocesor zanotowałby to poprzez ustawienie tego samego znacznika **C**, który ustawia w przypadku przeniesienia. W tym przypadku oczywiście **C**=0.

Jeżeli wynik operacji jest równy zero, wówczas mikroprocesor odnotowuje to poprzez ustawienie znacznika zera **Z**. Informacja o tym, że jakiś rejestr przyjął wartość zero jest użyteczna w przypadku realizacji pętli programowych.

Pojedynczy bajt może zostać podzielony na dwie liczby czterobitowe, z których każda może przedstawiać cyfrę dziesiętną zakodowaną dwójkowo (BCD). W takim przypadku istotna jest informacja o tym, czy nie nastąpiło przeniesienie z cyfry mniej znaczącej (bity 3-0) na cyfrę bardziej znaczącą (bity 7-4) lub zapożyczenie z pozycji bardziej znaczącej. Oba te fakty mikroprocesor odnotowuje poprzez ustawienie znacznik przeniesienia połówkowego **H**.

3.4 Operacje na liczbach w kodzie U2

Mikroprocesor operacje na liczbach w kodzie U2 wykonuje dokładnie w taki sam sposób jak operacje na liczbach w kodzie NB. To programisty musi zadbać o właściwą interpretację wyniku. Mikroprocesor wspomaga pracę programisty poprzez ustawianie dodatkowych znaczników zgodnie z regułami dodawania w kodzie U2.

Znacznik przepełnienia V ustawiany jest wówczas, kiedy wynik operacji nie może zostać zapisany w słowie wyjściowym gdyż nie mieści się w zakresie liczb w kodzie U2. Na przykład w arytmometrze ośmiobitowym

```
   01000000₂     =64₁₀,U2     =64₁₀,NB
 + 01000001₂     =65₁₀,U2     =65₁₀,NB
=  10000001₂     =-127₁₀,U2   =129₁₀,NB,
```

stąd **C**=0, **V**=1, gdyż maksymalna wartość jaką możemy zapisać w kodzie U2 na ośmiu bitach wynosi 127. Indeks **10,U2** oznacza wartość dziesiętną jeżeli liczba binarna traktowana jest jako liczba w kodzie U2, natomiast **10,NB** oznacza wartość dziesiętną jeżeli liczba binarna traktowana jest jako liczba w kodzie NB.

Dla potrzeby operacji na liczbach w kodzie U2 mikroprocesor posiada jeszcze jeden znacznik, tak zwany znacznik znaku **S**. Znacznik ten zostaje ustawiony jeżeli poprawny (prawdziwy) wynik operacji jest ujemny. Przykładowo operacja

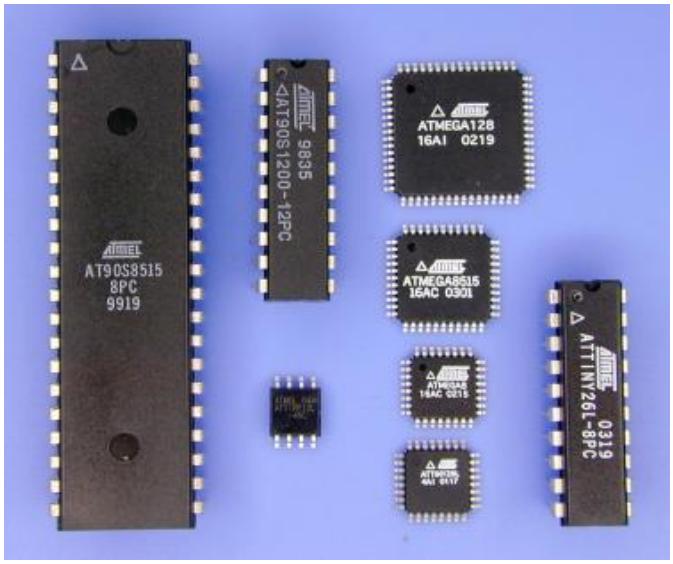
```
   11000000₂     =-64₁₀,U2    =192₁₀,NB
 - 01111111₂     =127₁₀,U2    =127₁₀,NB
=  01000001₂     =65₁₀,U2     =65₁₀,NB
```

ustawi znacznik **S**, gdyż prawdziwy wynik operacji jest ujemny, wynosi -64-(+127)=-191. W tym przykładzie poprawny wynik nie zmieścił się w rejestrze wyjściowym, w związku z czym dodatkowo zostanie ustawiony znacznik **V**.

Oprócz dwóch wymienionych znaczników, które maja wspomagać operacje na liczbach ze znakiem jest jeszcze jeden, który jest kopią najstarszego bitu wyniku zamieszczonego w rejestrze. Znacznik ten oznacza się najczęściej jako **N** i nazywa się go znacznikiem wartości ujemnej. Jeżeli jest ustawiony oznacza to, że w rejestrze znajduje się liczba ujemna. Wspomaga on operacje na liczbach w innych systemach kodowania znaku aniżeli U2. W ostatnim przykładzie bit ten zostałby wyzerowany – informację o znaku poprawnego wyniku (nawet takiego, który nie mieści się w rejestrze) niesie znacznik **S**.

4. Mikrokontrolery rodziny AVR

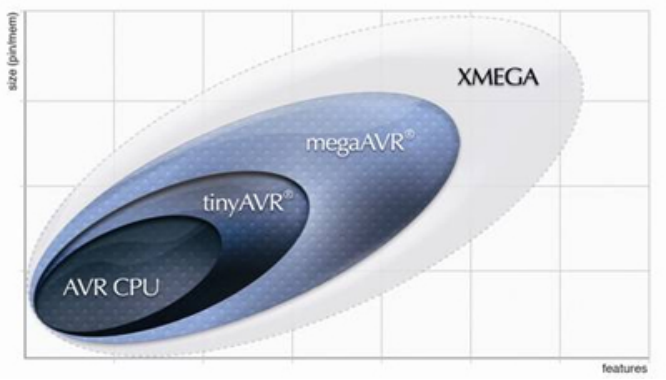
Mikrokontrolery firmy Atmel rodziny AVR są obecnie jednymi z najbardziej powszechnych i najbardziej wydajnych mikrokontrolerów ośmio bitowych. Moc obliczeniowa typowego układu jest rzędu kilkunastu MIPSów. Cena typowego mikrokontrolera tej rodziny jest rzędu kilku złotych za sztukę przy zakupie jednostkowym. Firma Atmel udostępnia bezpłatnie wszystkie niezbędne narzędzia programistyczne, które umożliwiają asemblowanie i programowanie, dodatkowo, bezpłatnie dostępny pakiet WinAVR umożliwia programowanie mikrokontrolera w języku C.



Rys. 9: Typowe obudowy mikrokontrolerów rodziny AVR

Mikrokontrolery tej rodziny dostępne są w szerokiej gamie obudów (rys. 9) oraz w wielu wariantach różniących się ilością zaimplementowanych zasobów pamięciowych oraz peryferii (rys. 10). Najmniejszymi przedstawicielami tej rodziny są układy ATtiny. posiadają one stosunkowo mało modułów oraz pamięć operacyjną liczoną w setkach bajtów. Z drugiej strony znajdują się układy ATXmega, które zostały zaprojektowane z myślą o bardzo wydajnych i zaawansowanych systemach kontrolno pomiarowych. Cechą wspólną wszystkich mikrokontrolerów rodziny AVR jest rdzeń AVR CPU (rys. 10). Układ, który zostanie wykorzystany na zajęciach (ATmega) jest

układem pośrednim. Umiejętność korzystania z jakiegokolwiek układu rodziny AVR pozwala na szybkie zapoznanie się z innym dowolnym (również bardziej zaawansowanym) mikrokontrolerem tej rodziny.



Rys. 10: Rodzina mikrokontrolerów AVR

Koszt elementów elektronicznych niezbędnych do budowy kompletnego systemu mikroprocesorowego z wykorzystaniem układu ATmega16 może nie przekraczać 20 zł. System oprogramowany na potrzeby konkretnego zadania może kosztować już kilkadziesiąt razy więcej. Programowanie mikrokontrolerów może więc nie tylko być zajęciem przyjemnym, ale również może być źródłem stałego i solidnego dochodu.

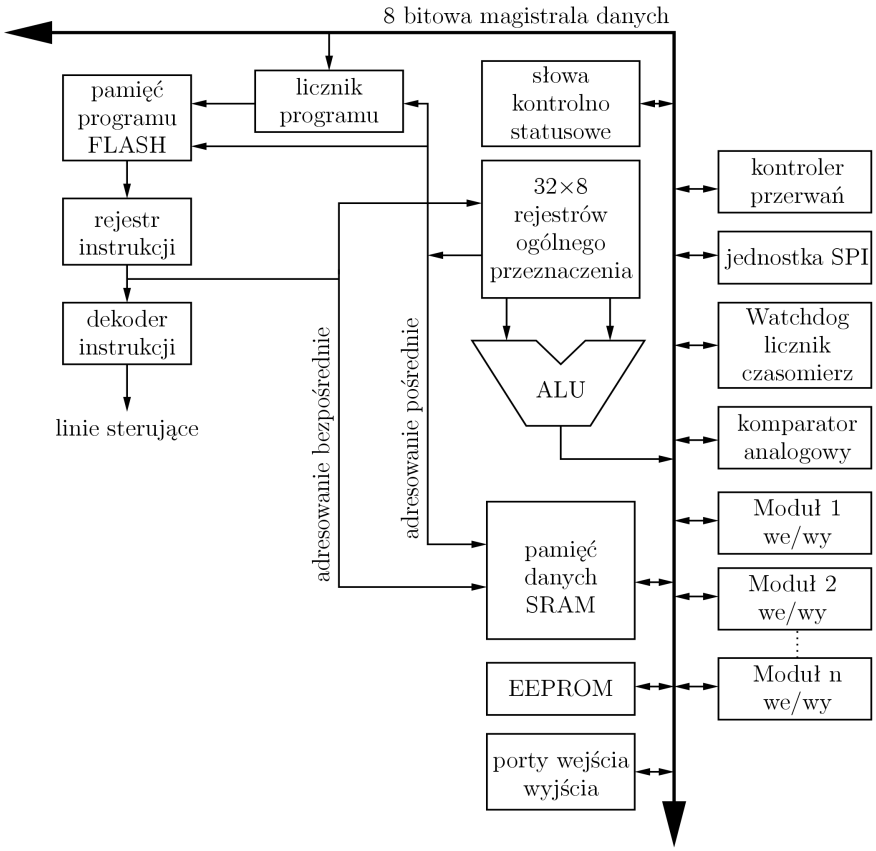
4.1 Rdzeń mikrokontrolera rodziny AVR [1,2]

Poprzez rdzeń (rys. 11) mikrokontrolerów rodziny AVR rozumie się najczęściej te zasoby i mechanizmy, które występują w każdym układzie tej rodziny (aczkolwiek nie zawsze w tych samych ilościach) lub też są niezbędne do właściwego wykonywania instrukcji. Należą do nich 32 8-bitowe rejestry ogólnego przeznaczenia **`R0`** – **`R31`**, mechanizm korzystania z przestrzeni wejścia wyjścia, pamięć danych SRAM, jednostka arytmetyczno logiczna (ALU), pamięć programu, licznik programu, mechanizm stosu, wskaźnik stosu, rejestr i dekoder instrukcji, pamięć EEPROM oraz inne. Do rdzenia nie będzie zaliczał się przykładowo moduł transmisji szeregowej (USART), komparator analogowy lub też przetwornik analogowo cyfrowy.

Aby zaznajomić się z architekturą mikrokontrolera AVR najlepiej jest prześledzić i zrozumieć działanie tych instrukcji Asemblera, których zadaniem jest dostęp do poszczególnych zasobów rdzenia AVR. Współcześnie mikrokontrolerów nie programuje się tylko wyłącznie w języku Asemblera – znajomość architektury (poprzez poznanie i zrozumienie kilku instrukcji) jest niezbędna do tego aby pisane programy w sposób optymalny wykorzystywały dostępne zasoby.

Rejestry ogólnego przeznaczenia mają za zadanie przechowywać zmienne na potrzeby wykonywania instrukcji. Zmienne przechowywane są w sposób trwały (długoczasowy, docelowy) w pamięci SRAM i tylko na potrzeby wykonania instrukcji, w których dana zmienna stanowi argument, są kopiowane do rejestrów ogólnego przeznaczenia (nie jest to proces samoczynny, pisząc w Asemblerze programista musi użyć odpowiedniej instrukcji lub grupy instrukcji aby dokonać kopiowania). Mechanizm korzystania z rejestrów wejścia wyjścia ma za zadanie zapewnić komunikację pomiędzy rdzeniem i modułami wejścia wyjścia. Rysunek 12 przedstawia mapę

pamięci danych. Od adresu 0 do adresu 31 znajdują się rejestry ogólnego przeznaczenia, od adresu 32 do adresu 95 znajduje się przestrzeń wejścia wyjścia (64 rejestry/porty wejścia wyjścia), poniżej od adresu 96 (szesnastkowo 0x60) znajduje się pamięć SRAM, gdzie docelowo przechowywane są zmienne.

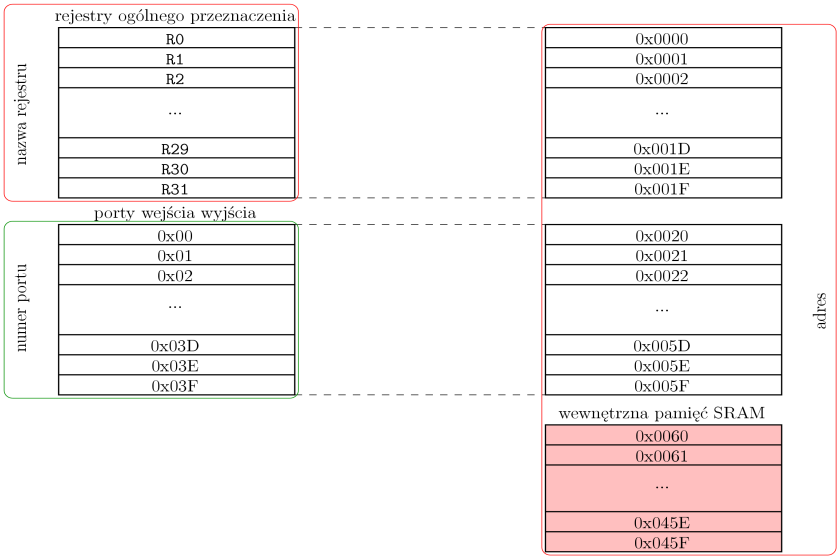


Rys. 11: Schemat blokowy mikrokontrolera rodziny AVR

4.1.1 Adresowanie bezpośrednie

Instrukcja **`MOV Rd, Rs`** pozwala na skopiowanie wartości z rejestru **`Rs`** do rejestru **`Rd`**, zawartość rejestru **`Rs`** po wykonaniu instrukcji nie ulegnie zmianie. Instrukcja **`STS K, Rs`**, pozwala na zapis do komórki pamięci danych o adresie **`K`** zawartości rejestru **`Rs`**. Jest to przykład adresowania bezpośredniego – bezpośrednio w kodzie instrukcji przekazany został adres pod który należy dokonać zapisu. Przy okazji należy wspomnieć, że adresowanie rozpoczyna się od rejestru **`R0`**, czyli adres 0 (**`K`**=0) odpowiada rejestrowi **`R0`**, adres 32 (**`K`**=32) odpowiada pierwszemu (zerowemu) rejestrowi wejścia wyjścia, natomiast zerowa komórka pamięci SRAM będzie miała adres 96 (szesnastkowo 0x60). Instrukcja ta (**`STS`**) ma więc możliwość przesłania pomiędzy rejestrami ogólnego przeznaczenia podobnie jak instrukcja **`MOV`**, ale takie jej zastosowanie jest nieoptymalne ze względu na czas wykonywania. Przykładowo instrukcje **`MOV R5, R3`** oraz **`STS 5, R3`** są równoważne pod względem logicznym, natomiast druga instrukcja wykona się w czasie dwukrotnie dłuższym aniżeli pierwsza. Tak więc instrukcję **`STS`** stosujemy wyłącznie w celu dokonania zapisu do pamięci SRAM. Instrukcją odwrotną do **`STS`** jest instrukcja **`LDS`**, która

powstała z myślą o kopiowaniu danych z pamięci do rejestrów ogólnego przeznaczenia. Podobnie jak w przypadku instrukcji **STS** adres dla instrukcji **LDS** podawany jest bezpośrednio w kodzie instrukcji. Przykładowo **LDS R5, 3** skutkuje tym samym co **MOV R5, R3** oraz **STS 5, R3**, natomiast **LDS R0, 0x6F** powoduje zapisanie do rejestru **R0** komórki pamięci SRAM o adresie 0x0F=15.



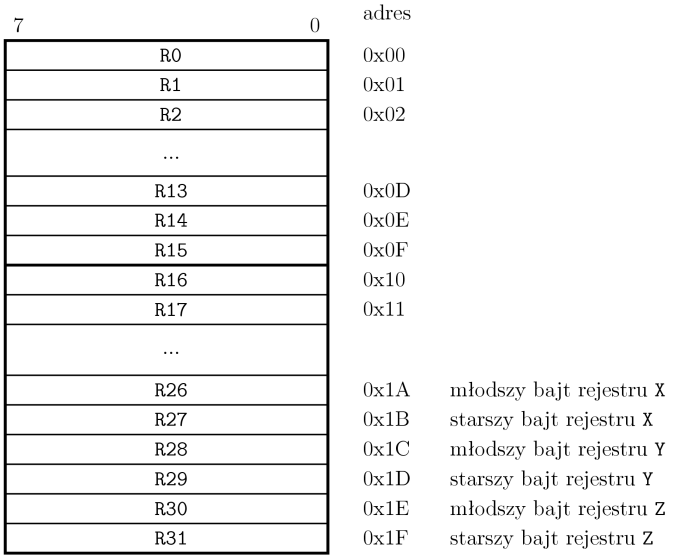
Rys. 12: Mapa pamięci danych mikrokontrolera AVR

4.1.2 Adresowanie pośrednie

Rejestry od **R26** do **R31** mogą zostać wykorzystane jako rejestry indeksowe (wskaźnikowe) w instrukcjach z adresowaniem pośrednim, czyli takim, w którym w kodzie instrukcji przekazuje się tylko informację o rejestrze, który przechowuje adres operandu, a nie sam operand lub jego adres. W mikrokontrolerach AVR zaimplementowano 3 16-bitowe rejestry wskaźnikowe o nazwach **X**, **Y** raz **Z**, które pokrywają się (zostały zmapowane) częściowo z rejestrami ogólnego przeznaczenia. 16 bitowy rejestr indeksowy **Z** stanowią rejestry **R31** oraz **R30**, z czego **R31** przechowuje bity bardziej znaczące, natomiast **R30** bity mniej znaczące (rys. 13). Podobnie zaimplementowano rejestry indeksowe **Y=R29:R28** oraz **X=R27:R26**. W celu uproszczenia dostępu do rejestrów indeksowych stosuje się następujące aliasy: **ZL** odpowiada rejestrowi **R31**, **ZH** odpowiada rejestrowi **R30**, czyli **Z=ZH:ZL** i analogicznie **Y=YH,YL** oraz **X=XH:XL**. Adresowanie pośrednie jest niezwykle wydajnym mechanizmem przesyłania tablic danych. Aby odczytać/zapisać (wskazać) kolejną komórkę pamięci wystarczy tylko zwiększyć o jeden adres zawarty w rejestrze indeksowym.

Przykładem instrukcji z adresowaniem pośrednim jest instrukcja **ST**. W instrukcji tej adres komórki pamięci do której wykonywany jest zapis przekazywany jest w jednym z trzech rejestrów indeksowych **X**, **Y** lub **Z**, natomiast sama instrukcja niesie informację który to jest rejestr (indeksowy) oraz który rejestr ogólnego przeznaczenia (**R0** – **R31**) stanowi źródło. Przykładowo **ST X, R0** powoduje zapisanie do komórki pamięci o adresie zawartym w rejestrze **X** zawartości rejestru **R0**. Programista sam musi zadbać o to aby rejestr **X** miał odpowiednią wartość, poprzez wcześniejszy zapis do **X**. Instrukcją która ma pełnić funkcję odwrotną jest

instrukcja **LD**. Przykładowo instrukcja **LD R20, Y** powoduje uzupełnienie rejestru **R20** zawartością komórki pamięci o adresie zawartym w rejestrze **Y**. Jak łatwo zauważyć, w przypadku, kiedy zawartości rejestrów indeksowych są mniejsze aniżeli 32 instrukcje **ST** oraz **LD** mogą pełnić tą samą funkcję co instrukcja **MOV**, podobnie jak to było z instrukcjami (**STS** oraz **LDS**) w przypadku kiedy adres bezpośredni **K**<32.



Rys. 13: Idea implementacji rejestrów indeksowych

4.1.3 Adresowanie pośrednie z postinkrementacją i predekrementacją

W celu kopiowania tablic danych zaimplementowano instrukcje, które albo po wykonaniu dostępu do pamięci zwiększają zawartość rejestru indeksowego, który przechowywał adres komórki pamięci albo przed dostępem zmniejszają zawartość rejestru indeksowego i następnie nowej zawartości tego rejestru używają jako adresu. Instrukcja **LD Rd, X+**, powoduje odczyt komórki pamięci o adresie zawartym w rejestrze **X** do rejestru **Rd** i następną inkrementację rejestru **X**. Instrukcja **ST X+, Rd** dokonuje kopiowania *w drugą stronę*. Jak widać jednokrotne wpisanie do rejestru **X** adresu początku tablicy danych a potem sukcesywne wykonywanie instrukcji z argumentem **X+** umożliwia odczyt/zapis całej tablicy danych.

Instrukcja **LD -Z, Rd** wykonuje wcześniejszą dekrementację rejestru **Z**, a następnie kopiowanie z rejestru **Rd** do komórki pamięci, której adres zawarty jest w zinkrementowanym już rejestrze **Z**. Uzupełnieniem tej instrukcji jest instrukcja **ST Rd, -Z**.

4.1.4 Adresowanie pośrednie z przesunięciem

Wzajemnie uzupełniające się instrukcje **LDD**, **STD** pozwalają na dostęp do argumentów zawartych w pamięci SRAM w sposób pośredni z przesunięciem. Oznacza to, że adres zapisywanej/odczytywanej komórki pamięci wyznaczany jest jako suma zawartości rejestru indeksowego oraz wartości przesunięcia podanego w kodzie instrukcji. Dla tych instrukcji możliwe do wykorzystania są rejestry indeksowe **Y** oraz **Z**. Instrukcja **LDD R0, Z+5** zapisze do rejestru **R0**

zawartość komórki pamięci, która znajduje się pod adresem równym sumie zawartości rejestru indeksowego **Z** i wartości 5. Jeśli przed wykonaniem tej instrukcji **Z**=2, to instrukcja ta będzie równoważna instrukcji **MOV R0, R7**.

4.1.5 Instrukcje z wartością natychmiastową

Instrukcje te zostały zaimplementowane na potrzeby uzyskiwania wartości stałych w programie. Jak można zauważyć na rysunku 11 pamięć programu stanowi całkowicie odrębny zasób w stosunku do pamięci danych. Inny jest mechanizm adresowania tej pamięci oraz jej przeznaczenie. Z pamięci programu pobierane są instrukcje, które następnie są dekodowane i wykonywane. Jeżeli programista chce przekazać do programu wartość stałą umieszcza ją właśnie w tej pamięci. Instrukcja **LDI Rh, K** pozwala na zapisanie wartości **K** z zakresu od 0 do 255 do jednego z 16 górnych (**h**=[16,31]) rejestrów ogólnego przeznaczenia. Dla tej instrukcji wartość stała przechowywana jest bezpośrednio w kodzie instrukcji, wartość ta dostępna jest w sposób natychmiastowy.

4.1.6 Adresowanie pośrednie w pamięci programu

Instrukcja **LDI** jest wygodna w przypadku kiedy wartość stała stanowi pojedynczy bajt. W przypadku, kiedy programista zdefiniował w pamięci tablice danych (np. napisy do wyświetlenia) wygodniej byłoby mieć instrukcję podobną do instrukcji **LD** z tym, że kopiującą z pamięci programu, a nie tak jak instrukcja **LD** z pamięci danych. Instrukcją taką jest instrukcja **LPM**, która z pamięci programu kopiuje wskazany przez rejestr indeksowy bajt do jednego z rejestrów ogólnego przeznaczenia. Przykładowo sekwencja **LPM R0, Z** spowoduje odczyt komórki pamięci programu spod adresu zawartego w rejestrze **Z** i umieszczenie zawartości tej komórki w rejestrze **R0**. Dla uproszczenia i przyspieszenia kopiowania tablic danych zaimplementowano instrukcję **LPM** z postinkrementacją (drugi argument przybiera postać **Z+**) adresu, podobnie jak to było w przypadku instrukcji **LD**.

4.1.7 Dostęp do przestrzeni wejścia wyjścia

Jak już wspomniano wcześniej komunikację pomiędzy modułami a rdzeniem procesora zapewniają rejestry umieszczone w przestrzeni wejścia wyjścia (porty). Dostęp do nich zapewniony jest poprzez instrukcje **IN** oraz **OUT**. Instrukcja **IN** pozwala odczytać dowolny rejestr z przestrzeni wejścia wyjścia i zapisać jego zawartość do rejestru ogólnego przeznaczenia. Instrukcja **OUT** pozwala zapisać dowolny rejestr w przestrzeni wejścia wyjścia zawartością rejestru ogólnego przeznaczenia. Przykładowo instrukcja **OUT 8, R5** spowoduje zapis do portu o numerze 8 zawartości rejestru **R5**, natomiast instrukcja **IN R10, 20** odczyta port (rejestr wejścia wyjścia) o numerze 20 i jego zawartość wpisze do rejestru **R10**. Podobnie jak instrukcja **MOV** instrukcje **IN** oraz **OUT** mogą zostać zastąpione instrukcjami **STS**, **ST**, **STD** oraz **LDS**, **LD** oraz **LDD**, ale nie jest to optymalne.

4.1.8 Podsumowanie

Przedstawione do tej pory instrukcje stanowią jedynie wycinek wszystkich instrukcji

mikrokontrolera rodziny AVR, są to tak zwane instrukcje przeniesień chociaż wykonują kopiowanie. Zrozumienie ich działania pozwala zrozumieć architekturę AVR, co jest niezbędne zarówno do programowania w języku Asemblera jak i wydajnego programowania w języku wysokiego poziomu.

4.2 Instrukcje warunkowe

W przypadku kiedy chcemy aby konkretny fragment kodu został wykonany w konkretnym przypadku a inny fragment kodu w pozostałych przypadkach stosujemy instrukcje warunkowe (rozgałęziające warunkowo). Przykładowo jeżeli w momencie przekroczenie zadanej temperatury ma być generowany sygnał dźwiękowy wówczas instrukcja warunkowa sprawdzi relację jaka zachodzi dla temperatury zadanej i odczytywanej i jeżeli temperatura odczytywana jest większa od zadanej wykona fragment kodu odpowiedzialny za generowanie sygnału dźwiękowego w przeciwnym przypadku zwróci sterowanie do programu głównego. Instrukcje warunkowe mogą zostać zastosowane również do wykonywania danego fragmentu kodu zadaną ilość razy. Wielokrotne wykonywanie tego samego fragmentu kodu może przykładowo zostać wykorzystane do kopiowania tablic danych z pamięci programu do pamięci danych z wykorzystaniem rejestrów indeksowych, lub też chociażby do zaimplementowania kodu, którego zadaniem jest migać diodą świecącą LED.

Instrukcje warunkowe w mikrokontrolerach AVR dzielimy zasadniczo na dwa rodzaje. Pierwszym rodzajem są instrukcje typu **SKIP,** które przeskakują (nie wykonują) instrukcję znajdującą się bezpośrednio za instrukcją warunkową typu **SKIP** jeżeli jest spełniony konkretny warunek. Drugim rodzajem instrukcji warunkowych są typowe instrukcje rozgałęzień warunkowych typu **BRANCH**. Instrukcje te skaczą (przekazując sterowanie) pod wskazany adres jeżeli jest spełniony przypisany dla instrukcji warunek.

4.2.1 Instrukcje typu BRANCH

Instrukcja **BRNE** rozgałęzia program (zmienia przebieg programu) skacząc do miejsca wskazanego etykietą jeżeli w wyniku poprzedniej operacji arytmetycznej powstał wynik różny od zera, w przypadku instrukcji testującej testowany rejestr posiadał wartość różną od zera lub w przypadku instrukcji porównującej gdy podczas porównania oba argumenty były różne. Przykładowo kod

```
LDI R16, 5
PĘTLA:
      DEC R16
      BRNE PĘTLA
kolejna_instrukcja
```

wykona się 5 razy. Etykietę **PĘTLA** należy rozumieć jako adres w pamięci programu. Nie znamy jego wartości pisząc program, ale nie przeszkadza to abyśmy posługiwali się etykietą **PĘTLA** wskazując na miejsce gdzie umieszczona została instrukcja **DEC R16**. Konkretna wartość liczbowa zostanie przypisana etykiecie w momencie asemblacji programu. Instrukcja arytmetyczna **DEC** zmniejsza zawartość rejestru o jeden. Tuż przed pierwszym wykonaniem instrukcji **DEC R16**, **R16**=5, a po wykonaniu tej instrukcji **R16**=4, instrukcja **BRNE PĘTLA** wykona w tym przebiegu

pętli rozgałęzienie do miejsca **PĘTLA**, gdyż w poprzedniej operacji arytmetycznej (**DEC R16**) nie osiągnięto wyniku równego zero, co spowoduje, że dekrementacja wykona się ponownie. Po piątym wykonaniu instrukcji **DEC R16, R16**=0 i **BRNE PĘTLA** nie wykona skoku do **PĘTLA** ale wykona **kolejną_instrukcję**.

Kolejna instrukcja **BREQ** rozgałęzia w przypadkach przeciwnych do tych, dla których rozgałęzia instrukcja **BRNE**. Instrukcje **BRCS**, **BRLO** rozgałęziają w przypadku, kiedy, na przykład, w wyniku poprzedniej operacji arytmetycznej nastąpiło przeniesienie lub zapożyczenie, natomiast instrukcje **BRCC**, **BRSH** rozgałęziają w przypadkach przeciwnych. Instrukcja **BRCS** robi dokładnie to samo co instrukcja **BRLO**, tak samo jest dla instrukcji pary instrukcji **BRCC**, **BRSH**. Utworzone zostały przez producenta (Atmel) dwa mnemoniki na potrzeby stosowania w zależności od kontekstu programu. Mnemonik instrukcji **BRCS** jest tłumaczony skocz jeśli bit **C** jest ustawiony, natomiast mnemonik instrukcji **BRLO** tłumaczy się jako skocz jeśli mniejszy (pierwszy argument operacji odejmowania – odjemna). Tak więc instrukcje **BRCS** oraz **BRCC** łatwiej jest stosować w przypadku kiedy wykonujemy dodawanie i chcemy sprawdzić czy nastąpiło przeniesienie (**C**=1), natomiast instrukcje **BRLO** i **BRSH** wykonujemy w przypadku odejmowania.

Ogólnie większość instrukcji typu **BRANCH** sprawdza znaczniki rejestru znaczników **SREG** mikrokontrolera AVR. Jak już powiedziano instrukcje **BRCC**, **BRSH**, **BRCS**, **BRLO** rozgałęziają ze względu na znacznik przeniesienia **C**. Kolejne poznane już instrukcje **BREQ** oraz **BRNE** sprawdzają znacznik **Z**, który wskazuje, że w wyniku operacji otrzymano wynik zerowy. Poniżej przedstawione zostały kolejne instrukcje typu **BRANCH** z zaznaczeniem znacznika i jego wartości dla którego następuje rozgałęzienie.

BRPL (**N**=0), **BRMI** (**N**=1)
BRGE (**S**=0), **BRLT** (**S**=1)
BRHC (**H**=0), **BRHS** (**H**=1)
BRTC (**T**=0), **BRTS** (**T**=1)
BRVC (**V**=0), **BRVS** (**V**=1)
BRID (**I**=0), **BRIE** (**I**=1)

Aby w pełni zrozumieć ich działanie należy zapoznać się z podrozdziałami 3.3, 3.4 oraz 4.2.2, zapraszam również do noty katalogowej producenta [1].

Instrukcja **BRBS** rozgałęzia program jeżeli bit znajdujący się w rejestrze **SREG**, którego indeks wskazany został drugim argumentem jest ustawiony – przeciwnie działa instrukcja **BRBC**.

4.2.2 Rejestr znaczników operacji arytmetycznych

Do pełnego zrozumienia instrukcji warunkowych wypada zapoznać się z rejestrem znaczników SREG mikrokontrolera AVR, zwany również rejestrem statusowym (rys. 14). Bity **H**, **S**, **V**, **N** **Z**, **C** ustawiane są zgodnie z wynikiem wykonanej operacji arytmetycznej (podrozdziały 3.3 i 3.4), testującej lub porównującej.

Bit **I** pozwala na blokadę przerwań. Jeżeli jest wyzerowany wówczas system przerwań nie będzie reagował na żadne zdarzenia mające miejsce w modułach mikrokontrolera. Jeżeli bit ten jest ustawiony oraz będzie miało zdarzenie w module, dla którego został zarejestrowany podprogram obsługi przerwania, wówczas podprogram ten zostanie wykonany. System przerwań umożliwia szybką odpowiedź (wywołanie właściwego podprogramu) na konkretne zdarzenia. Bez

jego obecności mikrokontroler musiałby zapytywać w sposób ciągły poszczególne moduły o to czy nie miało miejsce konkretne zdarzenie.

Bit **T** jest znacznikiem kopii i może zostać dowolnie wykorzystany – jest to po prostu jeden bit pamięci danych.

numer bitu	7							0
nazwa	I	T	H	S	V	N	Z	C
dostęp	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
wartość początkowa	0	0	0	0	0	0	0	0

Rys. 14: Rejestr statusowy mikrokontrolera rodziny AVR

4.2.3 Instrukcje typu SKIP

Instrukcja **SBIC** przeskoczy, anuluje wykonanie kolejnej instrukcji jeżeli bit we skazanym rejestrze wejścia wyjścia jest wyzerowany, przeciwnie działa instrukcja **SBIS**. Instrukcje te mogą zostać wykorzystane do implementacji oczekiwania na konkretne zdarzenia. Przykładowo w przestrzeni wejścia wyjścia w rejestrze kontrolnym **EECR** pamięci EEPROM znajduje się bit **EEWE**, który wskazuje stanem wysokim, że zapis do pamięci EEPPROM trwa, natomiast stanem niskim wskazuje, że zapis się już zakończył. Kod programu

```
CZEKAJ:
      SBIC EECR, EEWE
      RJMP CZEKAJ
kolejna_instrukcja
```

będzie wykonywany w sposób ciągły tak długo jak długo bit **EEWE** jest ustawiony, czyli do momentu zakończenia zapisu. Instrukcja **RJMP CZEKAJ** powoduje bezwarunkowy skok do miejsca wskazanego etykietą **CZEKAJ**. Jeżeli zapis się zakończy instrukcja **RJMP CZEKAJ** zostaje pominięta i wykonana zostaje **kolejna_instrukcja**.

Para instrukcji **SBRC** oraz **SBRC** działa analogicznie do instrukcji **SBIC** oraz **SBIC** z tą różnicą, że jej wykonywanie jest uzależnione od bitu znajdującego się we wskazanym rejestrze ogólnego przeznaczenia, a nie rejestrze przestrzeni wejścia wyjścia.

4.2.4 Podstawy Asemblera dla mikrokontrolerów AVR [3]

Asembler jest językiem programowania mikrokontrolerów i mikroprocesorów. Wykorzystuje się go pisząc programy na niskim poziomie, czyli na poziomie pojedynczych instrukcji mikroprocesora. Pisząc program w języku C lub Pascal mówimy o programowaniu wysokopoziomowym. Programowanie niskopoziomowe daje większe możliwości, ale wymaga również dużo większego nakładu czasu programisty na napisanie tak samo funkcjonalnego programu jak w języku wysokiego poziomu. Umiejętność programowania w Asemblerze pozwala jednak lepiej zrozumieć architekturę i lepiej wykorzystać mikroprocesor pisząc już później w języku wysokiego poziomu. Asembler oprócz możliwości wpisywania instrukcji za pomocą mnemoników dostarcza dodatkowo zestaw dyrektyw, funkcji i definicji oraz daje możliwość tworzenia etykiet i makr. Dzięki Asemblerowi nie musimy na przykład pamiętać i kontrolować wartości argumentów instrukcji (na przykład adres skoku) ale możemy posługiwać się etykietą,

którą Asembler zamieni na wartość liczbową w momencie asemblacji.

Dyrektywy **.DSEG**, **.ESEG** oraz **.CSEG** przełączają zawartość pliku na definicję segmentu pamięci, odpowiednio, danych, EEPROM oraz kodu. Przykładowo kod Asemblera

```
RJMP START:                              ;1

.DSEG                                    ;2
DMEM: .BYTE 6                            ;3

.CSEG                                    ;4
CMEM: .DB 0xAA, 0xBB, 0xCC, 0xDD, 0x0 ;5

START:                                   ;6

LDI ZL, LOW(CMEM<<1)                     ;7
LDI ZL, HIGH(CMEM<<1)                    ;8
LPM R0, Z                                ;9

LDI XL, LOW(DMEM)                        ;10
LDI XH, HIGH(DMEM)                       ;11
ST X, R0                                 ;12

END:                                     ;13
RJMP END                                 ;14 to już koniec
```

w linii 2 przełącza plik na definicję segmentu danych, a w linii 4 z powrotem na definicję segmentu programu. Standardowo plik z kodem źródłowym Asemblera, bez zastosowania jakiejkolwiek dyrektywy przełączającej, jest przełączony na definicję pamięci kodu, czyli na miejsce, gdzie standardowo mają znaleźć się instrukcje.

Dyrektywa **.BYTE** (linia 3) rezerwuje w pamięci miejsce na liczbę bajtów, która zostaje podana jako drugi argument. W tym przypadku miejsce to zostało zarezerwowane w pamięci danych, gdyż byliśmy przełączeni na definicję jej zawartości (**.DSEG** – linia 2). Natomiast etykieta **DMEM:** (linia 3) zawiera adres do pierwszego elementu znajdującego się tuż za nią. Troszeczkę inne działanie ma dyrektywa **.DB**, która nie tylko rezerwuje miejsce w pamięci, ale dodatkowo określa wartość zarezerwowanych bajtów. W tym przypadku (linia 5) pierwszy bajt tuż za etykietą **cmem:** będzie miał wartość 0xAA, drugi 0xBB itd. Bajty zostaną zapisane w pamięci kodu programu gdyż w linii 4 zastosowano dyrektywę **.CSEG**.

Instrukcją skoku względnego **RJMP** przeskakujemy do kodu, który zostanie wygenerowany przez instrukcje znajdujące się za etykietą **START:** (linia 6). W linii 7 wpiszemy część młodszą (**LOW**) adresu do pierwszej komórki pamięci programu znajdującej się za etykietą **CMEM:** do rejestru **ZL** (**R30**), który stanowi osiem młodszych bitów szesnastobitowego rejestru **Z**. W linii 8 do rejestru **ZH** wpiszemy część starszą (**HIGH**) adresu do pierwszej komórki pamięci programu znajdującej się za etykietą **CMEM:**. Operator **<<** dokonuje przesunięcia w lewo, co jest równoważne z mnożeniem przez 2 wartości etykiety **CMEM**. Operacja mnożenia jest konieczna, gdyż etykieta **CMEM:** znajduje się w pamięci programu FLASH, do której domyślny dostęp (pobranie kodu instrukcji) jest wykonywany w sposób 16 bitowy. Etykieta zwracając adres *nie wie*, że jest czytana na potrzeby dostarczania adresu dla ośmiobitowej instrukcji **LPM**. Aby to lepiej zrozumieć należy sobie uzmysłowić, że umieszczona w pamięci programu etykieta pod adresem **CMEM:** *widzi* dwa

bajty młodszy 0xAA oraz starszy 0xBB, pod adresem **CMEM+1:** *widzi* kolejne dwa bajty młodszy 0xCC oraz starszy 0xDD, natomiast instrukcja **LPM** pod adresem **CMEM·2:** *widzi* bajt 0xAA, pod adresem **CMEM·2+1:** *widzi* bajt 0xBB, a pod adresem **CMEM·2+2:** *widzi* bajt 0xCC itd. Podsumowując w liniach 7 i 8 do rejestru **Z** zostaje zapisany adres pierwszej komórki pamięci, która znajduje się tuż za etykietą **CMEM:**.

W linii 9 zawartość komórki pamięci, której adres został umieszczony w rejestrze **Z** zostaje zapisana do rejestru **R0**. W rejestrze **Z** znajduje się oczywiście wskaźnik (adres) na wartość 0xAA, a więc po wykonaniu instrukcji z linii 9, rejestr **R0** będzie miał wartość 0xAA.

Kod w liniach 10 i 11 działa analogicznie do kodu umieszczonego w liniach 7 i 8 z tą różnicą, że nie musimy mnożyć wartości zwracanej przez etykietę przez dwa, gdyż etykieta ta wskazuje na miejsce w pamięci danych, która jest 8 bitowa tak, jak instrukcja **ST**. Instrukcja z linii 12 zapisze do komórki pamięci danych o adresie zawartym w rejestrze **X** wartość znajdującą się w rejestrze **R0**. Ogólnie instrukcje od linii 7 do 12 włącznie dokonują przesłania bajtu z komórki pamięci programu do komórki pamięci danych. Bajt ten ma wartość 0xAA, w pamięci programu zawiera się pod adresem **DMEM:**, natomiast w pamięci programu zawiera się pod adresem **CMEM·2:**.

W linii 13 znajduje się etykieta a tuż za nią instrukcja skoku względnego i bezwarunkowego do instrukcji znajdującej się tuż za tą etykietą (linia 14), czyli instrukcja skoku na samą siebie, co powoduje zapętlenie programu na końcu. Średnik **;** umożliwia dodawanie komentarzy w kodzie programu. Wszystko co się znajdzie za średnikiem jest pomijane w procesie asemblacji.

4.3 Przykładowe programy w Asemblerze

Na zajęciach zostanie zademonstrowanych kilka prostych programów napisanych w Asemblerze na mikrokontroler ATmega16. Wszystkie prezentowane programy są dostępne na stronie www, wszystkie są w pełni funkcjonale. Niektóre programy w czasie zajęć będzie należało zmodyfikować na potrzeby uzyskania dodatkowego efektu. W czasie zajęć każdy z programów zostanie omówiony tak, aby nie pozostawić wątpliwości co do jego działania.

4.3.1 Migająca dioda

Program zawarty jest w pliku **led_twinkle.asm**. Program ten naprzemiennie zapala i gasi diodę świecącą LED. Dioda została podłączona do wyprowadzenia zerowego portu **PORTA**, który został skonfigurowany jako wyjście. Opóźnienie wykonywane jest w sposób programowy i w dostarczonym pliku jest ono na tyle duże, że miganie jest ledwo dostrzegalne (dioda miga bardzo szybko – gdybyśmy podłączyli generowany sygnał nie do diody ale do głośniczka usłyszelibyśmy buczenie). Na zajęciach należy, poprzez analogię do zastosowanego rozwiązania, zwiększyć opóźnienie generowane przez program tak, aby dioda migała z częstotliwością równą około 1 Hz.

4.3.2 Modulacja szerokości impulsu

Modulacja szerokości impulsu (PWM) wykorzystywana jest powszechnie w przemyśle do sterowaniu elementami wykonawczymi takimi, jak żarówki, silniki, grzejniki oporowe i inne. Podczas sterowania z wykorzystaniem PWM do odbiornika (na przykład żarówki), którego mocą

chcemy sterować (jasność świecenia żarówki) nie dostarcza się napięcia analogowego o wartości proporcjonalnej do żądanej jasności, ale napięcie prostokątne, którego wypełnienie jest tym większe im do odbiornika ma zostać dostarczona większa moc. Wypełnienie definiuje się jako stosunek czasu, kiedy odbiornik jest załączony (230 V) do sumy obu czasów, to jest czasu wyłączenia (0V) i wspomnianego wcześniej włączenia odbiornika. Program pwm.asm steruje diodą świecącą LED wykorzystując licznik czasomierz 0 mikrokontrolera AVR skonfigurowany do pracy w trybie PWM. Dostarczony program stopniowo rozjaśnia diodę świecącą LED (zmieniając wypełnienie sygnału na wyjściu OC0), a następnie ją gasi i cały proces powtarza ponownie. Zadaniem uczniów będzie taka zmiana programu, aby proces zachodził w przeciwnym kierunku, czyli aby dioda stopniowo zmniejszała intensywność świecenia.

4.3.3 Przerwania sprzętowe

Mechanizm przerwań upraszcza implementację obsługi zdarzeń, które wystąpiły poza rdzeniem mikrokontrolera, a należy na nie zareagować z jak najmniejszym opóźnieniem. W mikrokontrolerze AVR istnieje wiele źródeł przerwań. Na zajęciach jako przykład zostanie pokazana obsługa przerwań zewnętrznych (zewnętrzne wyprowadzenia mikrokontrolera) oraz przerwań pochodzących od licznika czasomierza zerowego. Program **`interrupt_ext.asm`** reaguje na zdarzenia zewnętrzne, które przejawiają się zboczem opadającym na wyprowadzeniu **`INT1`** (bit trzeci portu **`PORTD`**) mikrokontrolera. Każde zbocze opadające powoduje przerwanie aktualnie wykonywanego programu, zapisanie adresu powrotu na stosie i rozpoczęcie wykonywania podprogramu obsługi przerwania. Po wykonaniu podprogramu obsługi przerwania mikrokontroler, wykorzystując adres powrotu, powraca do przerwanego zajęcia. Drugi program **`interrupt_tc0.asm`** demonstruje obsługę dwóch rodzajów zdarzeń. Pierwsze przerwanie występuje w momencie, kiedy licznik osiągnie wartość zapisaną w rejestrze z nim skojarzonym (0x10=16). Następuje wówczas wykonanie odpowiedniego podprogramu, ale licznik liczy dalej. W momencie, kiedy się przepełni (255->0) następuje wywołanie drugiego podprogramu obsługi przerwania. Zadaniem uczniów jest taka modyfikacja drugiego programu, aby znacząco zmniejszyć czas pomiędzy wywołaniem przerwania na zdarzenie, kiedy licznik osiągnie wartość zapisaną w rejestrze a zdarzeniem, kiedy się przepełni.

4.3.4 Program odtwarzający dźwięk

Program **`sound.asm`** odtwarza dźwięk zapisany w postaci cyfrowej w pliku **`sound.txt`**, który (plik) z użyciem odpowiednich dyrektyw Asemblera zostaje w momencie asemblacji umieszczony w sekcji danych pamięci programu. Podczas działania programu pobierane są kolejne bajty z sekcji danych pamięci programu, które określają wypełnienie na wyjściu PWM. Do wyjścia PWM podłączony jest głośniczek i w efekcie słyszany jest sygnał dźwiękowy zapisany w pliku. Dostarczony w pliku tekstowym sygnał dźwiękowy powstał poprzez modyfikację oryginału **`sound_22050_Hz_16_bit.wav`**, który został spróbkowany z częstotliwością 22050 Hz oraz rozdzielczością 16 bitów. Aby możliwe było zapisanie tego dźwięku do mikrokontrolera (ograniczona pamięć) sygnał ten został poddany procesowi zmniejszania rozdzielczości do 8 bitów oraz procesowi zmniejszania częstotliwości próbkowania do 3906 Hz. Plik **`sound_3906_Hz_8_bit.wav`** zawiera tą samą informację co dostarczony plik tekstowy, ale w formacie, który można bezpośrednio odtworzyć na komputerze. Zadaniem uczniów będzie taka

modyfikacja programu, aby poprawnie odtworzyć kolejne nagranie **sound1.txt**, w którym próbki umieszczone zostały w odwrotnej kolejności.

4.3.5 Kopiowanie tablic danych

Program **strcopy.asm** dokonuje kopiowania do pamięci EEPROM tablicy danych, która (tablica) pierwotnie znajduje się w pamięci programu FLASH, a następnie tablica ta jest kopiowana do pamięci danych SRAM. Śledząc działający na modelu program można dostrzec istniejące w rodzinie AVR mechanizmy adresowania. Program demonstruje zapis i odczyt pamięci EEPROM, który jest nieco odmienny od zapisu i odczytu innych zasobów pamięciowych.

4.3.6 Edycja pamięci EEPROM

Program **eeprom_edit.asm** umożliwia edycję (zapis oraz odczyt) czterech pierwszych komórek pamięci EEPROM. Interfejs użytkownika stanowią cztery przyciski (młodsze bity portu **PORTB**) oraz osiem diod LED (**PORTA**). Przycisk pierwszy umożliwia wybór edytowanego bitu, każde jego wciśnięcie powoduje wybór kolejnego bitu. Przycisk drugi umożliwia edycję bitu, każde jego wciśnięcie powoduje zanegowanie wybranego do edycji bitu. Przycisk trzeci powoduje zapis do komórki pamięci EEPROM na którą wskazuje wskaźnik zapisu oraz następnie inkrementację modulo 4 wskaźnika zapisu. Przycisk czwarty umożliwia odczyt i inkrementację modulo 4 wskaźnika odczytu. Oba te wskaźniki są niezależne od siebie.

Po zapisaniu kilku przykładowych wartości do pamięci EEPROM, można układ wyłączyć z prądu i ponownie włączyć, a wartości, które zostały zapisane nie ulegną skasowaniu.

4.4 Programowanie mikrokontrolerów w języku C

Programowanie mikrokontrolerów w języku C nie różni się znacząco od programowania komputera osobistego w tym języku, lub innym języku wysokiego poziomu. Należy jednak pamiętać o tym, że wszystkie zasoby są tutaj mniej liczne. Do dyspozycji mamy nie setki megabajtów ale pojedyncze kilobajty pamięci operacyjnej. Moc obliczeniowa jest rzędu pojedynczych MIPSów, a większość operacji arytmetycznych jest wykonywana programowo.

Aby pisanie programów w języku wysokiego poziomu było w ogóle możliwe należy znać architekturę programowanego mikrokontrolera. Oprócz ogólnej znajomości języka dobrze jest poznać te jego elementy, które pozwalają na programowanie współbieżne. Aby wykorzystać optymalnie programowany mikrokontroler należy zaznajomić się z biblioteką producenta (avr-libc [5]), która dostarcza szeregu pomocnych definicji, makr oraz funkcji. Poniższe przykłady mają na celu pokazanie jak proste może być programowania mikrokontrolerów. Przy tej okazji zademonstrowane stosowaną podstawowe konstrukcje, ponad standard języka, stosowane przy programowaniu mikrokontrolerów AVR.

4.4.1 Eliminacja wpływu drgań styków

Każde naciśnięcie lub zwolnienie przycisku w początkowej fazie powoduje szereg zwarć i rozwarć, które spowodowane są drganiami mechanicznymi elementów zwierających. Efekt ten często nazywa się drganiami styków i jest on oczywiście niepożądany z punktu widzenia systemu

mikroprocesorowego. Nie jesteśmy w stanie tego efektu wyeliminować, jedyne co, to mamy możliwość wyeliminować jego wpływ na stan układu elektronicznego. Algorytm eliminacji jest bardzo prosty. Jeżeli nastąpiło przyciśnięcie przycisku, należy odczytać go ponownie po czasie dłuższym aniżeli okres drgań styków (10 ms) i sprawdzić czy nadal przycisk jest wciśnięty. Jeżeli jest wciśnięty, to należy podjąć związaną z tym przyciskiem akcję, w przeciwnym wypadku należy wrócić do pętli głównej programu.

Program **key_cnt.c** powoduje zwiększenie stanu diod podłączonych do portu **PORTA** przy każdym wciśnięciu jakiegokolwiek z czterech przycisków podłączonych do młodszych bitów portu **PORTB**. Zadaniem ucznia jest taka modyfikacja programu aby jedno przyciśnięcie powodowało pojedynczą inkrementację stanu diod (należy zastosować algorytm eliminacji opisany powyżej). Do uzyskania opóźnień można wykorzystać funkcję **_delay_ms(t)** (**util/delay.h**), która pozwala uzyskiwać opóźnienia o wartości **t** wyrażonej w milisekundach. Następnie należy napisać program, który będzie rozróżniał przyciski (dla każdego przycisku zostanie przypisana inna akcja). Naciśnięcie pierwszego przycisku ma zwiększać stan diod o 1, drugiego zwiększać o 10, trzeciego ma zmniejszać o 10, a czwartego zmniejszać o 1.

4.4.2 Sterowanie multipleksowe zespołem wyświetlaczy siedmio-segmentowych

Sterowanie multipleksowe wykorzystywane jest w sytuacjach, kiedy istnieje konieczność sterowania stosunkowo licznym zbiorem elementów. Idea sterowania multipleksowego polega na tym, że w danym przedziale czasu sterowany jest jeden podzbiór elementów, podczas gdy inne pozostają wyłączone. W następnym przedziale czasu sterowany jest kolejny podzbiór itd. W danej chwili tylko elementy jednego podzbioru są sterowane, podczas gdy wszystkie inne są wyłączone. Przełączanie podzbiorów następuje stosunkowo szybko (1 ms), tak więc obserwator ma wrażenie, że wszystkie elementy są sterowane jednocześnie.

W programie **7_seg_mux_edit_dec.c** występuje sterowanie multipleksowe czterema 7 segmentowymi wyświetlaczami LED. Wyprowadzenia segmentów o tych samych oznaczeniach (oraz znaków dziesiętnych) wszystkich czterech wyświetlaczy zostały podłączone (równolegle) do ośmiu bitów portu **PORTA** mikrokontrolera. Cztery sygnały wspólne (wspólne katody) podłączono do czterech młodszych bitów portu **PORTB**. Dostarczony program umożliwia edycję dowolnej cyfry przy użyciu prostego interfejsu zbudowanego z wykorzystaniem czterech przycisków. Przycisk pierwszy wybiera kolejną cyfrę do edycji, natomiast przycisk drugi wybiera poprzednią. Przycisk trzeci powoduje zwiększenie stanu wybranej cyfry, natomiast przycisk czwarty powoduje zmniejszenie stanu tej cyfry.

Definicje wyglądu poszczególnych cyfr zawarte są w pamięci programu, do której na poziomie danych otrzymujemy dostęp poprzez stosowanie przy deklaracji atrybutu **__attribute__((progmem))** (**avr/pgmspace.h**) oraz funkcji **pgm_read_byte()** przy odczycie. Makro **ISR()** pozwala zdefiniować podprogram obsługi przerwania, w tym przypadku (**TIMER0_COMP_vect**) jest to przerwanie wykonywane w sytuacji, kiedy licznik osiągnął wartość zapisaną w rejestrze z nim skojarzonym. Przerwanie to wykorzystywane jest do przełączania podzbiorów elementów sterowanych multipleksowo.

Dostarczony program należy zmodyfikować w taki sposób, aby umożliwiał on edycję liczb szesnastkowych (na pojedynczej cyfrze wystąpią wartości 0-9 oraz abcdef). Dodatkowo należy zaimplementować migający punkt dziesiętny na wyświetlaczu, na którym znajduje się edytowana cyfra.

4.4.3 Obsługa przetwornika analogowo-cyfrowego

Program **7_seg_mux_calc_adc.c** stanowi demonstrację wykorzystania przetwornika analogowo-cyfrowego. Wartość podawana na wejście przetwornika AC ustalana jest za pomocą potencjometru znajdującego się na zestawie laboratoryjnym. Każde naciśnięcie przycisku drugiego spowoduje dodanie aktualnego stanu przetwornika AC do zmiennej **wynik**, naciśnięcie przycisku trzeciego spowoduje odjęcie od zmiennej **wynik** aktualnego stanu przetwornika, natomiast naciśnięcie przycisku czwartego spowoduje wyzerowanie zmiennej **wynik**. Na czterech wyświetlaczach 7 segmentowych (**PORTB[0-7]**, **PORTC[0-3]**) może pojawiać się albo aktualny stan przetwornika AC albo wartość zmiennej **wynik**. Przełączanie pomiędzy wartością a przetwornikiem AC odbywa się z wykorzystaniem przycisku pierwszego. Przyciski podłączono do starszych bitów portu **PORTC**.

Program należy rozbudować tak, aby edycja każdej cyfry możliwa była z wykorzystaniem potencjometru. Dwa przyciski należy wykorzystać do wyboru edytowanej cyfry. Maksymalna wartość odczytana z przetwornika wynosi 1023, stąd wartość cyfry na pozycji **pos** można wyznaczać w programie w następujący sposób **data[pos]=10*ADC/1024**, gdzie **ADC** jest wskazaniem przetwornika AC.

4.4.4 Obsługa klawiatury matrycowej

Podobnie jak sterowanie multipleksowe wyświetlaczami 7 segmentowymi odbywa się odczyt klawiatury matrycowej. W danej chwili czasowej tylko jedna kolumna przycisków jest aktywna, to znaczy, że przy odczycie otrzymamy stan aktywny na którymkolwiek wierszu, jeżeli zostanie wciśnięty przycisk znajdujący się w tej kolumnie. Jeżeli jest wciśnięty tylko przycisk znajdujący się w nieaktywnej kolumnie, wówczas wszystkie wiersze będą nieaktywne w tej chwili czasowej. Oczywiście przełączanie następuje bardzo szybko (1 ms) i nawet najkrótsze wciśnięcie przycisku (30 ms) nie zostanie przegapione.

Dostarczony program **7_seg_mux_key_mux_adder.c** demonstruje rozróżnianie przycisków oraz eliminację wpływu drgań styków. Każde naciśnięcie przycisku powoduje zwiększenie stanu wyświetlanego na czterech wyświetlaczach 7 segmentowych o wartość równą indeksowi przycisku (0 – 15). Wyświetlacze podłączono tak samo jak w poprzednim przykładzie. Klawiaturę natomiast podłączono w taki sposób, że sygnały sterujące (kolumny) podłączono do starszych bitów portu **PORTC**, natomiast wiersze (odczyt) podłączono do starszych bitów portu **PORTA**.

Z wykorzystaniem dostarczonego szablonu należy napisać program realizujący kalkulator. Przyciski o indeksach od 0 do 9 mogą stanowić wpisywane cyfry. Przyciski o indeksach 10, 11, 12 i 13 mogą być odpowiadać za poszczególne operacje **+**, **-**, ***** oraz **/**. Przycisk o indeksie 15 może stanowić odpowiadać za wykonanie działania (**=**), a przycisk o indeksie 14 może być przyciskiem zerującym wynik.

4.4.5 Interfejs szeregowy RS232

Program **rs_echo.c** (AVR) odbiera znaki pochodzące z komputera (Hyper Terminal) gromadzi je w buforze i z częstotliwością równą około 1 Hz odsyła z powrotem. Jeżeli użytkownik

wpisze kilka znaków w krótkim przedziale czasu, wówczas znaki zostaną zgromadzone w buforze i będą odsyłane sukcesywnie (jeden znak na sekundę). Indeks (położenie w buforze) dla znaków odebranych i wysłanych jest raportowany na porcie **PORTA**.

Zadaniem ucznia jest tak zmodyfikować program aby odebrane wielkie litery wysyłał z powrotem jako małe i vice versa, inne znaki mają nie ulegać zmianie.

5. Komputer osobisty

Wszystkie komputery osobiste klasy PC w swojej budowie wykorzystują mikroprocesory, które są w pełni kompatybilne wstecz z mikroprocesorem 8086. Mikroprocesor 8086 był wykorzystywany w komputerach klasy IBM XT, które to są pierwowzorem współczesnych komputerów osobistych. W rozdziale tym omówione zostanie działanie mikroprocesorów rodziny x86. Na zajęciach do zademonstrowania działania mikroprocesora w trybie rzeczywistym wykorzystany zostanie program MS DEBUG, natomiast do demonstracji pracy w trybie chronionym wykorzystany zostanie emulator systemu QEMU oraz Assembler FLATASM.

5.1 Procesor 8086 [6]

W podrozdziale tym opisany zostanie mikroprocesor 8086. Przedstawiony zostanie mechanizm segmentacji pamięci, rejestry mikroprocesora oraz mapa pamięci. Omówiony zostanie sposób wyznaczania adresu argumentu, adresu instrukcji oraz adresu podprogramu obsługi przerwania w trybie rzeczywistym.

5.1.1 Rejestry mikroprocesora

Mikroprocesor 8086 posiada kilkanaście rejestrów 16 bitowych (rys. 15). Niektórym z tych rejestrów przypisano funkcje, inne można traktować jako rejestry ogólnego przeznaczenia. Rejestr **AX** zwany jest akumulatorem stanowi on przeznaczenie (miejsce zapisu wyniku) dla większości instrukcji mikroprocesora. Rejestr **BX** może zostać wykorzystany jako rejestr indeksowy, lub dodatkowy rejestr danych. Rejestr **CX** głównie służy do implementacji pętli programowych, gdzie stanowi licznik iteracji. Rejestr **DX** jest rejestrem danych wykorzystywanym dla operacji wejścia wyjścia oraz operacji arytmetycznych. Wszystkie te cztery rejestry mogą być dostępne w całości (16 bitów) lub w 8 bitowych kawałkach. Wówczas rejestr **AX** składa się z dwóch rejestrów **AH** oraz **AL**. W tym przypadku rejestr **AH** stanowi część bardziej znaczącą, natomiast rejestr **AL** część mniej znaczącą. Podobnie jest z rejestrami **BX**, **CX** oraz **DX**.

Rejestr **SP** jest wskaźnikiem stosu. Razem z rejestrem **SS** określają miejsce w pamięci gdzie znajduje się wierzchołek stosu. Rejestr **BP** może być wykorzystany jako rejestr indeksowy w segmencie stosu. Przy operacjach przesłań blokowych stosuje się wskaźniki źródła (**SI**) – miejsce odczytu danych oraz przeznaczenia (**DI**) – miejsce zapisu danych. Rejestry **SI** oraz **DI** są rejestrami indeksowymi (w odróżnieniu od **BP**) w segmencie danych.

Rejestr **IP** jest wskaźnikiem instrukcje, razem z rejestrem **CS** wskazuje na miejsce skąd pobierane są instrukcje. Rejestr **FLAGS** jest rejestrem zawierającym znaczniki dla operacji arytmetycznych oraz inne bity kontrolne i statusowe.

Cztery rejestry segmentowe **CS**, **DS**, **SS** oraz **ES** służą jako rejestry określające początki segmentów (szczegóły w podrozdziale 5.1.2).

Rys. 15: Rejestry procesora 8086

5.1.2 Segmentacja pamięci

Mikroprocesor 8086 umożliwia pracę wyłącznie w trybie rzeczywistym, w którym możliwe jest zaadresowanie tylko 1 MB pamięci. Rejestry procesora 8086 są 16-bitowe i adresowanie powyżej 64 kB jest możliwe poprzez zastosowanie mechanizmu segmentacji pamięci. Znajomość mechanizmu segmentacji jest niezbędna do zrozumienia pracy w trybie chronionym. Każdy współczesny procesor po uruchomieniu pracuje w trybie rzeczywistym i dopiero system operacyjny, przełącza go na pracę w trybie chronionym.

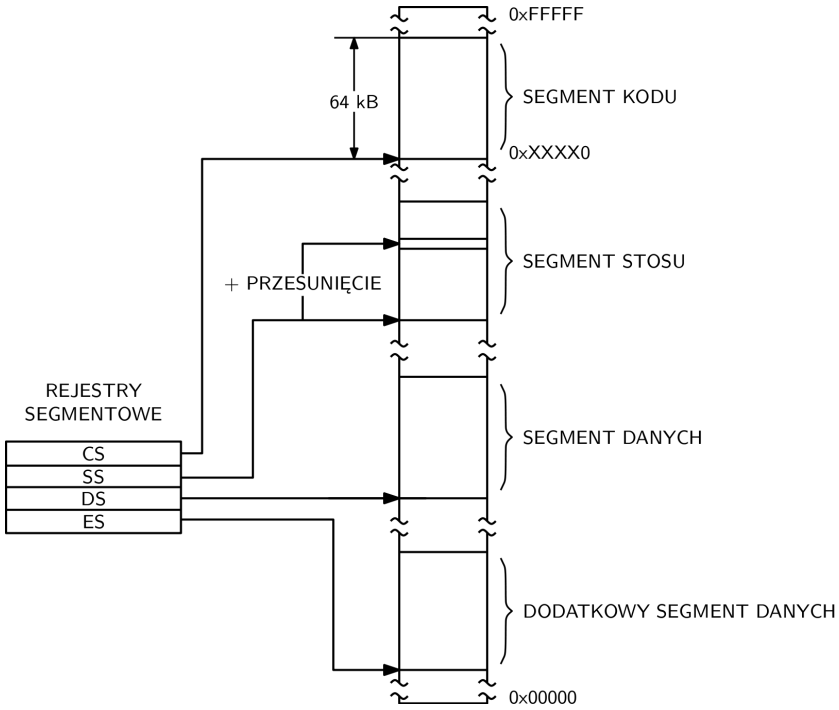
Rysunek 16 przedstawia mechanizm segmentacji pamięci. Czterem rejestrom procesora (**CS**, **SS**, **DS**, **ES**) zostały przypisane funkcje rejestrów segmentowych. Wskazują one na początek segmentu pamięci, którego funkcja jest ściśle określona.

Rejestr segmentowy kodu wskazuje na miejsce w pamięci gdzie znajduje się segment kodu, (pamięć, gdzie zapisane są instrukcje/program mikroprocesora). Sposób wyznaczania adresu fizycznego w trybie rzeczywistym jest stosunkowo prosty (rys. 17). Na przykład, w celu wyznaczenia adresu instrukcji, mikroprocesor mnoży zawartość rejestru segmentowego programu/kodu **CS** przez 16 (przesuwana w lewo o 4 pozycje) i uzyskaną wartość zwiększa o zawartość wskaźnika instrukcji. Innymi słowy, adres fizyczny dla instrukcji wynosi **CS***16+**IP**.

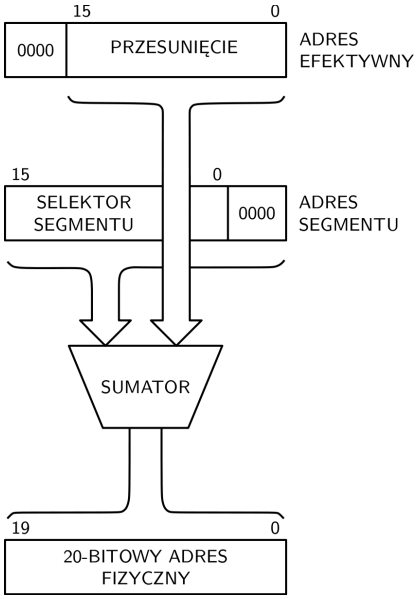
Podobnie odbywa się wyznaczanie adresu wierzchołka stosu, adres fizyczny w tym przypadku wynosi **SS***16+**SP**. Adres fizyczny dla danych wyznaczany jest z wykorzystaniem rejestru **DS** lub **ES**. Jeżeli programista nie wskaże, który rejestr ma zostać wykorzystany użyty zostanie domyślnie rejestr **DS**.

Dzięki zastosowaniu mechanizmu segmentacji możliwe jest zaadresowanie 1 MB pamięci, a w nowszych procesorach (mających bit adresowy **A20**) do 1088 kB. Przesunięcie w rejestrze segmentowym nazywany jest adresem efektywnym. Adres efektywny jest 16 bitowy, stąd maksymalna wielkość segmentu wynosi 64 kB. Segmenty mogą nawzajem się pokrywać, mogą też stanowić obszary rozłączne lub sąsiednie. Ze względu na segmentację danych programiście łatwiej jest zachować przejrzystość kodu, a adresy, którymi musi się posługiwać (szczególnie istotne przy

adresowanie bezpośrednim) mogą być stosunkowo krótkie (16 bitowe).



Rys. 16: Segmentacja pamięci w procesorach x86



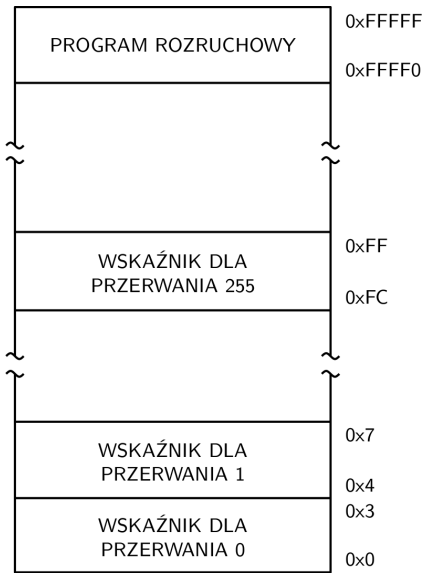
Rys. 17: Wyznaczanie adresu fizycznego w trybie rzeczywistym

Początkowy obszar pamięci (pierwszy 1 kB) mikroprocesora został przewidziany na wektory przerwań (rys. 18). W procesorze znajduje się 256 wektorów przerwań, każdy wektor zajmuje 4 bajty. Pierwsze 2 bajty określają segment, a kolejne dwa bajty określają adres

efektywny miejsca w pamięci, gdzie znajduje się podprogram obsługi danego przerwania.

Aby uzyskać adres podprogramu obsługi przerwania (segment i przesunięcie) należy wektor tego przerwania pomnożyć przez 4 i wykorzystać 4 kolejne bajty. Dla przerwań sprzętowych, pochodzących od programowalnego kontrolera przerwań (8259) wektor przerwania przekazywany jest w momencie potwierdzania przerwania.

Po uruchomieniu mikroprocesor rozpoczyna wykonywanie instrukcji od adresu 0xFFFF0. W tym miejscu należy umieścić instrukcję skoku do miejsca w pamięci gdzie znajduje się program inicjalizujący system.



Rys. 18: Mapa pamięci procesora x86

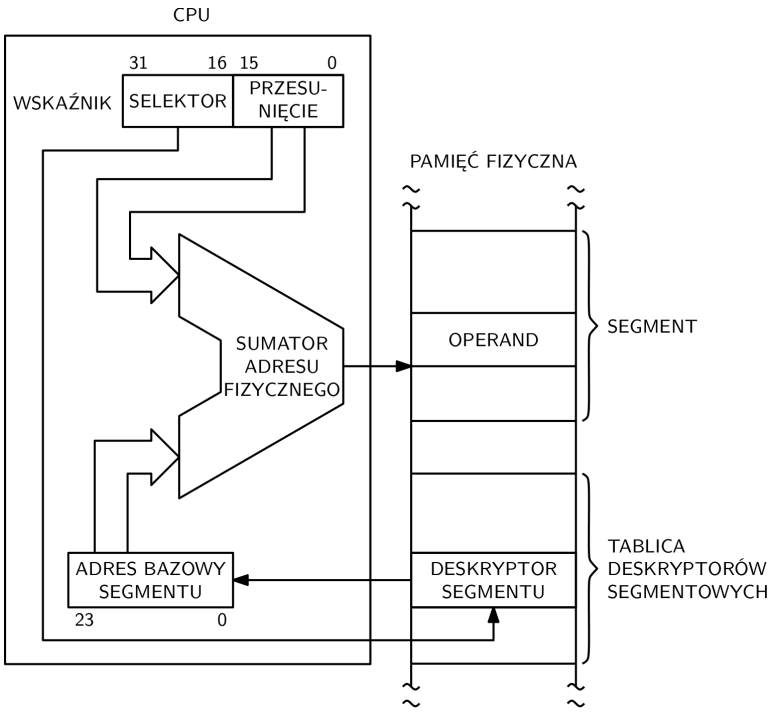
5.2 Praca w trybie chronionym [7]

W trybie chronionym rejestry segmentowe pełnią rolę selektorów. Wyznaczają one miejsce w tablicy deskryptorów segmentowych, gdzie znajduje się deskryptor segmentu (rys. 19). Dopiero deskryptor segmentu zawiera adres bazowy segmentu. Adres fizyczny wyznaczany jest w ten sposób, że do adresu bazowego dodawany jest adres efektywny (przesunięcie). Operacja ta wykonywana jest w sumatorze adresu fizycznego, który wykonuje zwykłe dodawania (bez przesunięcia segmentu tak, jak miało to miejsce przy segmentacji w trybie rzeczywistym).

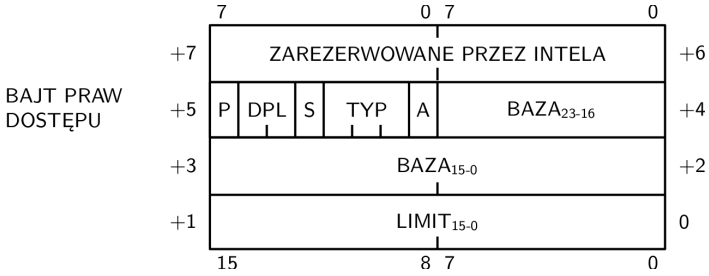
Rysunek 20 przedstawia budowę deskryptora segmentowego dla systemu 16 bitowego (80286). Widać tutaj, że adres bazowy (**BAZA**) jest 24 bitowa, stąd wynika, że możemy zaadresować aż 16 MB pamięci, wykorzystując segmenty o wielkości do 64 kB każdy (pole **LIMIT** jest 16 bitowe). W procesorach 32 bitowych (80386 i wyższych) adres bazowy został rozszerzony do 32 bitów (z miejsca zarezerwowanym przez Intela wykorzystano na ten cel 8 bitów) i dzięki temu możliwe stało się adresowanie w obszarze do 4 GB pamięci. 8 bitów, które pozostały w obszarze wcześniej zarezerwowanym przeznaczono na zwiększenie pola **LIMIT** do 24 bitów, co pozwala na uzyskanie pojedynczego segmentu o wielkości do 16 MB.

W deskryptorze występuje tak zwany bajt praw dostępu. Określa on między innymi (pole **DPL**) jaki minimalny poziom uprzywilejowania powinien posiadać proces, który otrzyma dostęp do

zasobów wskazywanych przez deskryptor. Pole **TYP** określa rodzaj wskazywanego segmentu (danych, kodu, stosu – dla deskryptorów segmentu, bramki lub inny – dla deskryptorów systemowych). Bit **S** określa czy deskryptor jest deskryptorem segmentowym, czy też deskryptorem systemowym. Bit **P** określa czy dane zasoby rezydują w pamięci, czy też zostały zrzucone na dysk twardy. Bit **A** wskazuje, że deskryptor był w ostatnim czasie użyty.



Rys. 19: Wyznaczanie adresu fizycznego w trybie chronionym

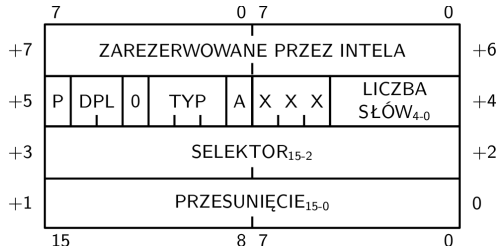


Rys. 20: Budowa deskryptora segmentowego

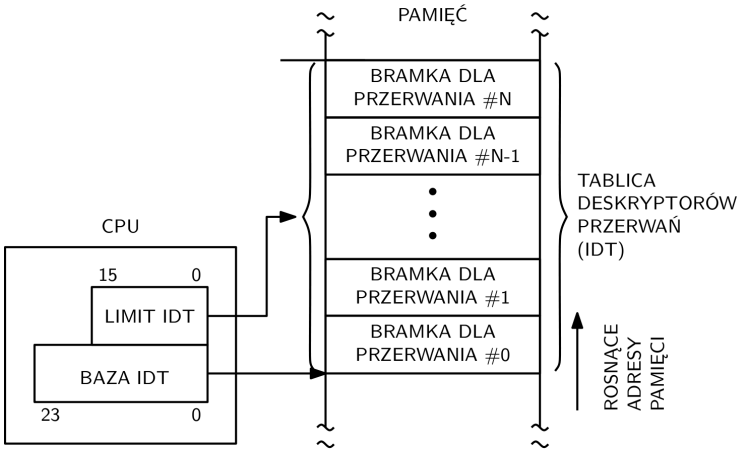
Wyznaczanie adresu podprogramu obsługi przerwania wymaga użycia deskryptora bramki, który nie przechowuje adresu bazowego, ale **SELEKTOR** (rys. 21), który wskazuje na konkretny deskryptor w tablicy deskryptorów (rys. 19), z którego dopiero pobierany jest adres bazowy.

Podobnie jak to było w trybie rzeczywistym również i w trybie chronionym przerwania mają swoje wektory. Wektor przerwania w trybie chronionym określa miejsce w tablicy deskryptorów przerwań (rys. 22), skąd pobierany jest deskryptor bramki (bramka dla przerwania). W trybie chronionym można jednak określić liczbę przerwań (**LIMIT IDT**) oraz ich

położenie w pamięci (**BAZA IDT**).



Rys. 21: Budowa deskryptora bramki



Rys. 22: Wyznaczanie adresu podprogramu obsługi przerwania w trybie chronionym

Literatura:

[1] 8-bit Microcontroller with 16K Bytes In-System Programmable Flash – Atmega16 ATmega16L. Atmel 2006.

[2] Mikrokontrolery AVR w praktyce. Jarosław Doliński. BTC. Warszawa 2003.

[3] AVR Assembler User Guide. Atmel 1998.

[4] Uniwersalny zestaw uruchomieniowy dla mikrokontrolerów AVR. KAMAMI, BTC.

[5] AVR-libc Reference Manual. Savannah. Free Software. GNU.

[6] 8086 16-bit HMOS Microprocessor. Intel 1990.

[7] 80286 High Performance Microprocessor with Memory Management and Protection. Intel 1988.