

1. POJĘCIE SYSTEMU WBUDOWANEGO. PRZYKŁADY. HISTORIA. OBSZAR ZASTOSOWAŃ

W ostatnich latach, komputery i różne urządzenia cyfrowe bardzo głęboko przenikają do wszystkich sfer działalności człowieka: od statków kosmicznych i aparatów głębinowych po proste gry elektroniczne i systemy sterowania sztucznym sercem.

Aby objąć całą różnorodność urządzeń elektronicznych otaczających człowieka, został wprowadzony termin systemy wbudowane (embedded systems).

System wbudowany jest specjalizowanym systemem komputerowym, który rozwiązuje jedno lub kilka specyficznych zadań i jest bezpośrednio wbudowany w sterowany obiekt.

Przeciwieństwem systemu wbudowanego jest uniwersalny komputer, na przykład komputer osobisty, który jest przeznaczony do rozwiązywania różnorodnych zadań dla szerokiego kręgu użytkowników.

Obecnie, systemy wbudowane i wszystko, co jest z nimi związane – jest najbardziej gwałtownie rozwijającą się dziedziną informatyki.

Ważność systemów wbudowanych w życiu codziennym współczesnego człowieka potwierdza się w części tym, że od niedawna przedmiot „Systemy wbudowane” został przedmiotem obowiązkowym na specjalności „Informatyka” na studiach w uczelniach technicznych.

PRZYKŁADY:

- urządzenia przenośne PDA (personal digital assistant),
- odtwarzacze mp3 i DVD,
- telefony komórkowe,
- kontrolery gier video,
- aparaty fotograficzne i kamery cyfrowe,
- kuchenki mikrofalowe, pralki, lodówki,
- klimatyzatory, termostaty i inne;
- urządzenia komputerowe i biurowe:
- drukarki, skanery, faksy, modemy,
- dyski twardye,
- klawiatury i inne;
- systemy transportowe:
- różne sterowniki w samochodach, układy sterujące silnikiem,
- zawieszeniem,
- wspomaganiem parkowania,
- systemami ABS, ESP/ESC, nawigacją GPS;
- systemy sterowania sygnalizacją świetlną;
- systemy sterowania ruchem pojazdów, lotem samolotów i inne;
- systemy telekomunikacyjne:
- centrale telefoniczne,
- układy sterujące ruchem w sieciach,
- modemy, hosty;
- aparatura medyczna:
- systemy monitorowania życia,
- elektrokardiografy, stetoskopy elektroniczne,
- tomografy komputerowe, aparaty słuchowe,
- systemy korekcji wzroku, różna aparatura diagnostyczna;
- robotyka,

- zwłaszcza zautomatyzowane i automatyczne urządzenia wykorzystywane w środowiskach agresywnych:
 - przy pożarach,
 - w warunkach podwyższonej radioaktywności;
 - podwodne aparaty głębinowe,
 - stacje kosmiczne,
 - aparatura do badania kraterów wulkanów;
 - zautomatyzowana aparatura w fabrykach chemicznych,
 - w reaktorach jądrowych, elektrowniach atomowych i inne

HISTORIA:

Historycznie, za pierwszy system wbudowany uważa się komputer pokładowy Autonetics D-17 rakiety kosmicznej Minuteman, zbudowanej w 1961 roku w ramach projektu kosmicznego Apollo, który, jak wiadomo, zakończył się udanym lotem na Księżyc. Masowe wdrożenie przemysłowe komputera D-17 odbyło się w 1966 roku.

2. FUNKCJE SYSTEMÓW WBUDOWANYCH. CECHY WSPÓŁCZESNYCH SYSTEMÓW WBUDOWANYCH

FUNKCJE:

- Sterowanie różnymi urządzeniami peryferyjnymi.
- Pobieranie sygnałów i ich przetwarzanie (od przycisków, przełączników, czujników i innych).
- Formowanie sygnałów sterowania logicznego.
- Realizacja interfejsów peryferyjnych (szeregowego, równoległego, z klawiaturą, wyświetlaczem LED, LCD, VGA).

Oprócz tego współczesne systemy wbudowane charakteryzują się następującymi cechami:

- konieczność obróbki potoków informacji o dużej objętości z wysoką szybkością,
- możliwość operacyjnej zmiany algorytmów przetwarzania informacji,
- możliwość dołączenia nowych funkcji przetwarzania informacji.

Ostatnio największą popularnością cieszą się miniaturowe urządzenia bezprzewodowe (*motes*).

Dla systemów wbudowanych często stawia się osobne **wymagania** określone specyfiką realizowanych zadań i obszarem zastosowania. Niektóre z **wymagań przedstawiono poniżej**:

- praca w trybie czasu rzeczywistego, gdy mocno ograniczony jest czas reakcji systemu wbudowanego na określone zdarzenia;
- minimalny pobór mocy;
- minimalne gabaryty i ciężar;
- możliwość pracy w agresywnym środowisku

3. BAZA TECHNOLOGICZNA SYSTEMÓW WBUDOWANYCH. INNE TECHNOLOGIE

Baza technologiczna systemów wbudowanych jest ścisłe związana z ewolucją minikomputerów i mikrokomputerów.

Pierwsze systemy wbudowane budowano na bazie tranzystorów i układów małej skali integracji.

Jednak szerokie rozprzestrzenienie się systemów wbudowanych rozpoczęło się w momencie pojawienia się mikroprocesorów.

Następnym etapem było zastosowanie do budowy systemów wbudowanych komputerów jednopłytowych i mikrokontrolerów.

UKŁADY PROGRAMOWALNE:

Ostatnio systemy wbudowane coraz częściej budowane są na bazie układów logiki programowalnej:

- Field Programmable Gate Array – FPGA i
- Complex Programmable Logic Device – CPLD oraz
- systemów na jednym układzie scalonym (System on Chip – SoC i System on Programmable Chip – SoPC)

W przypadku produkcji masowej, na przykład telefonów komórkowych, systemy wbudowane mogą być budowane na bazie zamawianych specjalizowanych układów scalonych: *application specific integrated circuits - ASICs*.

INNE TECHNOLOGIE:

W niektórych wypadkach do budowy systemów wbudowanych można wykorzystywać płyty główne komputerów osobistych. W tym przypadku można zastosować systemy operacyjne ogólnego wykorzystania oraz popularne języki programowania.

Ostatnio do budowy systemów wbudowanych coraz częściej wykorzystuje się gotowe do wykorzystania płyty komputerowe (*ready made computer boards*)

4. MIKROKONTROLERY. MIKROPROCESORY. KOMPUTERY JEDNOPŁYTOWE. WADY MIKROKONTROLERÓW I MIKROPROCESORÓW

Mikrokontroler – mikrokomputer zrealizowany na jednym układzie scalonym. Architektura mikrokontrolera zawiera główne komponenty komputera:

- procesor,
- szyny,
- pamięć
- zbiór urządzeń peryferyjnych najczęściej wykorzystywanych w systemach wbudowanych:
 - układy czasowe i liczniki,
 - porty szeregowe i równoległe,
 - przetworniki analogowo-cyfrowe i cyfrowo-analogowe,
 - kontrolery popularnych interfejsów i inne.

Główną wadą mikrokontrolerów jest mała moc obliczeniowa (która jednak w wielu systemach wbudowanych nie jest potrzebna).

Mikroprocesory, w odróżnieniu od mikrokontrolerów, dysponują większą mocą obliczeniową, jednak mają niewielką liczbę układów peryferyjnych. Ważną zaletą mikroprocesorów jest możliwość wykorzystywania systemów operacyjnych DOS, Windows, Unix, Linux itp. i języków programowania wysokiego poziomu.

Komputery jednopłytowe kompensują wady mikroprocesorów w odniesieniu do małej liczby urządzeń peryferyjnych, przy czym pozostaje możliwość wykorzystywania systemów operacyjnych ogólnego przeznaczenia.

Jednak komputery jednopłytowe w większości przypadków nie zapewniają wymagań systemów wbudowanych na pobór mocy, ciężar i gabaryty.

Jeszcze jedną ważną wadą mikrokontrolerów i mikroprocesorów jest niewielka liczba wyprowadzeń zewnętrznych, do przesyłania danych. Rozwiązaniem tego problemu jest multipleksowanie danych, a do przesyłania większych ilości informacji często wykorzystuje się interfejsy szeregowe. Jednak wszystkie podobne podejścia prowadzą do obniżenia szybkości działania systemu wbudowanego, co może być niedopuszczalne dla systemów czasu rzeczywistego.

5. UKŁADY LOGIKI PROGRAMOWALNEJ. ZALETY LOGIKI PROGRAMOWALNEJ

Ostatnio do budowy szybkich systemów wbudowanych najczęściej wykorzystuje się układy logiki programowalnej: FPGA, CPLD i SoPC. Dzięki dużej mocy funkcjonalnej i dużej liczbie wyprowadzeń zewnętrznych, układy te pozwalają, drogą konfiguracji swojej struktury wewnętrznej, realizować na jednym układzie scalonym całe systemy mikroprocesorowe z dużą liczbą urządzeń peryferyjnych.

Inną właściwością układów programowalnych jest to, że pozwalają one rozwiązywać specjalne zadania systemów wbudowanych nie programowo, za pomocą wykonywania szeregu rozkazów, a sprzętowo. Inaczej mówiąc, specjalne zadania przedstawiane są w postaci układu logicznego, który jest realizowany sprzętowo w układzie programowalnym.

Przy czym, **szynkość** rozwiązania zadania, w porównaniu z mikroprocesorami i mikrokontrolerami, jest **kilka rzędów wyższa**.

ZALETY:

Inną ważną cechą układów logiki programowalnej jest **duża liczba wyprowadzeń** zewnętrznych ogólnego przeznaczenia. Dzięki temu w wielu przypadkach można zrezygnować z multipleksowania sygnałów i wykorzystywania różnych standardowych interfejsów.

Oprócz tego, układy logiki programowalnej wyróżniają się **niskim poborem mocy, małymi gabarytami i ciężarem**. Ostatnio pojawiła się tendencja do realizacji w układzie scalonym, razem z logiką programowalną, **różnych urządzeń peryferyjnych**, często wykorzystywanych przy budowie systemów wbudowanych: bloków pętli sprzężenia fazowego (PLL), przetworników analogowo-cyfrowych (A/D) i cyfrowo-analogowych (D/A), komparatorów sygnałów analogowych, bloków mnożących itp.

6. PŁYTY KOMPUTEROWE. GOTOWE DO WYKORZYSTANIA PŁYTY DLA SYSTEMÓW WBUDOWANYCH

Typowym przykładem gotowych do wykorzystania płyt komputerowych, które wykorzystuje się do budowy niewielkich systemów wbudowanych są płyty PC/104 i PC/104+. Płyty te bazują na mikroprocesorach Intel x86 i są fizycznie znacznie mniejsze niż standardowe płyty główne komputerów osobistych.

Na takich płytach można wykorzystywać systemy operacyjne MSDOS, Linux, NetBSD, a także systemy operacyjne czasu rzeczywistego MicroC/OS-II, QNX i VxWorks.

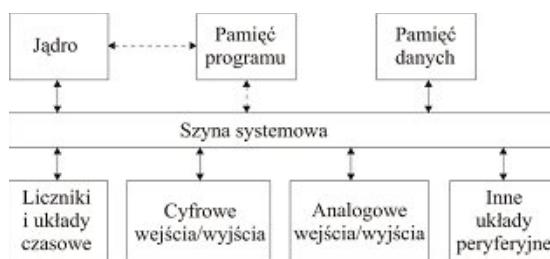
Czasami na takich płytach stosuje się mikroprocesory inne niż Intel x86.

GOTOWE DO WYKORZYSTANIA PŁYTY DLA SYSTEMÓW WBUDOWANYCH:

Można określić niektóre ogólne właściwości gotowych do wykorzystania płyt dla systemów wbudowanych: istnienie układu SoC lub SoPC wykonanego w korpusie BGA; układy zewnętrznej pamięci FLASH i DRAM; standardowe kontrolery interfejsów itp.

Na układach SoC i SoPC realizowany jest wysokowydajny procesor ARM i różnorodna logika systemu wbudowanego. Układy pamięci są wykorzystywane do przechowywania stałych i tymczasowych danych. Takie płyty mogą być wykonywane w dużych ilościach i charakteryzują się niskim kosztem.

7. OGÓLNA STRUKTURA MIKROKONTROLERA. RODZINY MIKROKONTROLERÓW



Podstawą każdej rodziny jest jądro mikrokontrolera, które może być 8, 16 lub 32 bitowe. W ramach **jednej rodziny** mikrokontrolery różnią się **wielkością pamięci i zestawem układów peryferyjnych**. W ten sposób, w jednej rodzinie można spotkać bardzo proste układy z niewielką ilością pamięci i małą liczbą wyprowadzeń zewnętrznych, jak i dostatecznie złożone układy z odpowiednio większą ilością pamięci i bogatym zestawem układów peryferyjnych.

Rodzina mikrokontrolerów

Jądro mikrokontrolera jest takie samo dla wszystkich układów w tej samej rodzinie. Dlatego każda **rodzina** mikrokontrolerów **posiada jeden zestaw instrukcji**. Ta cecha znacznie upraszcza programowanie mikrokontrolerów jednej rodziny. Można także zauważyc, że mikrokontrolery mogą być produkowane w **różnych obudowach**, które z kolei charakteryzują się **dużą różnorodnością**.

8. MIKROKONTROLERY PIC FIRMY MICROCHIP. GŁÓWNE PARAMETRY MIKROKONTROLERÓW PIC ŚREDNIEJ KLASY

Pierwsze mikrokontrolery PIC (Peripheral Interface Controller) zostały opracowane w latach siedemdziesiątych dwudziestego wieku w firmie General Instruments.

Cechami charakterystycznymi mikrokontrolerów PIC są:

- architektura RISC;
- prostota procesora;
- mała liczba rozkazów;
- autonomiczność pracy;
- wysoka szybkość działania;
- niski koszt.

Rozpatrywane są tylko **8-bitowe** mikrokontrolery PIC, które firma Microchip zalicza do mikrokontrolerów **średniej klasy** (Mid-Range MCU).

Mikrokontrolery **tej klasy** charakteryzują się następującymi właściwościami:

- są zbudowane na bazie architektury harwardzkiej, tzn. jądro procesora (Central Processing Unit – CPU) ma oddzielne szyny do połączenia z pamięcią programu i pamięcią danych;
- realizują skrócony zestaw rozkazów (Reduced Instruction Set Computer – RISC);
- dopuszczają pracę w trybie autonomicznym;
- wykorzystują potokowe przetwarzanie rozkazów;
- zawierają specjalny rejestr roboczy W nazywany akumulatorem;
- mają stałe adresy w pamięci programu dla wektora zerowania i wektora przerwania.

GŁÓWNE PARAMETRY MIKROKONTROLERÓW PIC ŚREDNIEJ KLASY:

Mikrokontrolery każdej rodziny mają jednakową architekturę jądra i co za tym idzie – jednakowy zbiór rozkazów. **Symbol „S”** w oznaczeniu mikrokontrolerów PIC maja następujące znaczenie:

- „C” – wytworzony w technologii CMOS;
- „F” – w mikrokontrolerze stosuje się pamięć FLASH;
- „XX” lub „XXX” – cyfry oznaczające model mikrokontrolera.

Symbol „A” po grupie cyfr oznacza, że było przeprowadzone **technologiczne uaktualnienie** układu z wcześniejszej produkcji.

| Rodzina | Rozmiar stosu (w słowach) | Liczba bitów słowa rozkazowego | Liczba rozkazów | Liczba wektorów przerwań |
|---------------|---------------------------|--------------------------------|-----------------|--------------------------|
| 12CXXX/12FXXX | 2 | 12/14 | 33 | - |
| 16C5XX/16F5XX | 2 | 12 | 33 | - |
| 16CXXX/16FXXX | 8 | 14 | 35 | 1 |
| 17CXXX | 16 | 16 | 58 | 4 |
| 18CXXX/18FXXX | 32 | 16 | 75 | 2 |

9. JĘZYK ASSEMBLERA MIKROKONTROLERÓW PIC16. ETYKIETY. MNEMONIKI. DYREKTYWY. MAKRODEFINICJE. OPERANDY. KOMENTARZE

Asembler dla mikrokontrolerów PIC ma nazwę Microchip's Universal Assembler (MPASM). Dla trybu wiersza poleceń (środowiska MSDOS) jest przeznaczony program mpasm.exe, dla środowiska Windows -mpasmwin.exe.

Plik wejściowy z tekstem programu musi być z **rozszerzeniem „.asm”**. Asembler produkuje dwa pliki wyjściowe:

1. plik obiektowy z rozszerzeniem „.o”,
2. plik absolutnego kodu z rozszerzeniem „.hex”.

Plik absolutnego kodu z rozszerzeniem „.hex” jest przeznaczony dla **programatora**, tj. urządzenia, które zapisuje program do programowej pamięci mikrokontrolera. Pliki obiektowe z rozszerzeniem „.o” zawierają **kod przemieszczalny**, który można dołączyć do innych podobnych plików w złożonym projekcie.

9.1. Format tekstu programu

Plik wejściowy z tekstem programu musi być z rozszerzeniem „.asm”. W tekście asemblerowym zezwolono **korzystać tylko ze znaków ASCII** (kody od 0 do 127).

Tekst programu w języku asemblera MPASM ma sztywny format.

Każda instrukcja powinna zajmować **odrębny wiersz** z liczbą znaków nie więcej niż 255. Wiersz składa się z części nazywanych zwykle kolumnami, które są oddzielone separatorami. Separatorem może być **spacja, dwukropka lub znak tabulacji**.

- Kolumna **pierwsza** jest przeznaczona na **etykiety**.
- Od kolumny **drugiej** mogą znajdować się **mnemoniki i operandy** instrukcji asemblera.
- Dyrektywy asemblera mogą znajdować się w każdej z kolumn.

9.2. Etykiety

Po etykiecie można użyć każdego separatora, nie tylko tradycyjnego dwukropka. Etykieta nie powinna mieć dwóch podkreśleń lub podkreślenia i cyfry z przodu. Jasne, że etykietą nie może być słowem kluczowym języka asemblera.

Długość etykiety nie może być większa niż 32 znaki. Etykiety są czujne na wysokość liter.

Etykiety pisane są w pierwszej kolumnie (label), służą do nazywania konkretnych adresów pamięci programu, istnieją jedynie na poziomie kodu asemblerowego.

9.3. Mnemoniki

Mnemoniki to skróty zastosowane do oznaczeń rozkazów procesora. Mnemoniki nie są czujne na wysokość liter.

Mnemonik – słowa reprezentujące instrukcje, np. **add, sub**

9.4. Operandy

Operand zawiera informację o tym, gdzie znajduje się dana. Operand musi być odseparowany od mnemonika spacją lub tabulacją.

Operand – wartość (argument) instrukcji, np. zmienna, stała, rejestr.

9.5. Dyrektywy

Dyrektywy to instrukcje skierowane do asemblera. Dyrektywy nie są czujne na wysokość liter.

Dyrektywy – komendy języka, które nie są bezpośrednio interpretowane przez mikroprocesor tylko przez kompilator dla ułatwienia programowania, np.

- PROCESSOR,
- BANKSEL (do wyboru aktywnego banku pamięci)

9.6. Marodefinicje (makra)

Makrodefinicje to zdefiniowane przez programistę grupy instrukcji asemblera. Asembler wpisuje tą grupę do kodu programu.

9.7. Komentarze

Komentarz powinien być zapisany poza średnikiem i zajmować koniec wiersza. Komentarz może być napisany w języku narodowym.

Zaczynają się od: ; nie są komplikowane, są to notatki ułatwiające zrozumienie danego fragmentu kodu.

10. STAŁE LICZBOWE I ZNAKOWE. PRZYKŁADY STAŁYCH. CIĄGI ZNAKÓW. SŁOWA ZAREZERWOWANE

Stałe liczbowe mogą być określone w formacie dziesiętnym, szesnastkowym, ósemkowym lub binarnym. Co uwarunkowują odpowiednie przedrostki:

- B – binarne (np. B'00111001', b'00111001'),
- D – dziesiętne (np. D'100', d'100' lub .100),
- H – szesnastkowe (np. H'9f', h'9f' lub 0x9f) ,
- O – ósemkowe (np. O'777', o'777').

Ciągiem znaków (stałą łańcuchową) jest wszystko pomiędzy apostrofami lub cudzysłowami, co zawiera **więcej niż 4 znaki**. Stosuje się je w rozkazach takich jak DB/DW, INCLUDE i FNAME. Stałych łańcuchowych nie można stosować w zwykłych wyrażeniach.

Przykład deklaracji stałych znakowych w pamięci programu:

db "łańcuch w 'cudzysłowach' może zawierać apostrofy"

db 'a łańcuch w "apostrofach" może zawierać cudzysłowy'

Stałe znakowe podlegają takim samym regułom jak stałe łańcuchowe, lecz można je stosować w wyrażeniach. Mogą mieć **długość do 4 znaków** i są przechowywane w kolejności od najmłodszego bajtu.

Zatem stała 'A' jest traktowana jako 41h, 'AB' jako 4241h (lub 41h, 42h), 'ABC' jako 434241h (41h,42h,43h) i 'ABCD' jako 44434241h.

W DB/DW stałe znakowe są zawsze traktowane jako stałe łańcuchowe, chyba że znajdują się wewnątrz wyrażenia.

Przykład:

```
db  'abcd'      ; daje 'a', 'b', 'c', 'd'.
```

```
db  +"abcd",1+'a'  ; daje 'a', 'b'.
```

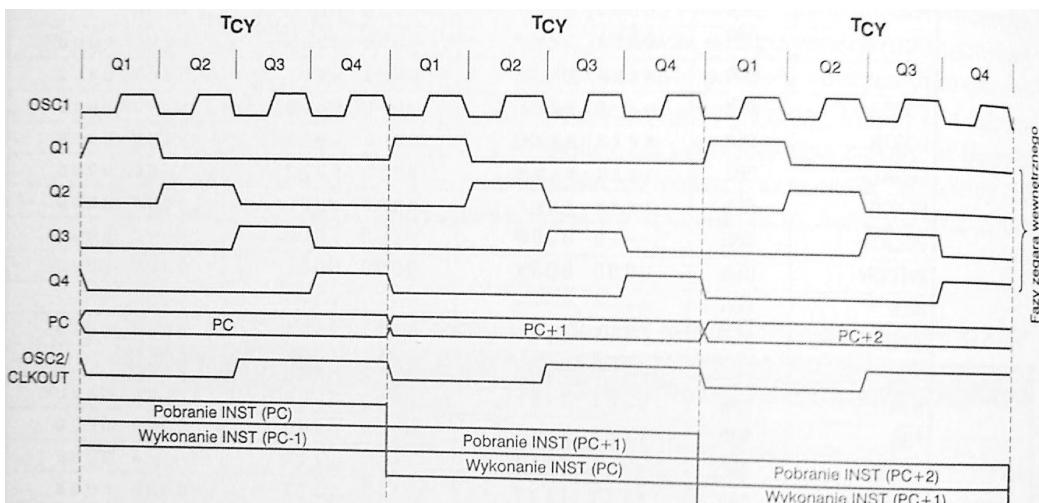
Ponieważ oba łańcuchy są wewnątrz wyrażenia, traktuje się je jako stałe znakowe.

```
db  ('abcd' >> 8)  ; daje 'b'
```

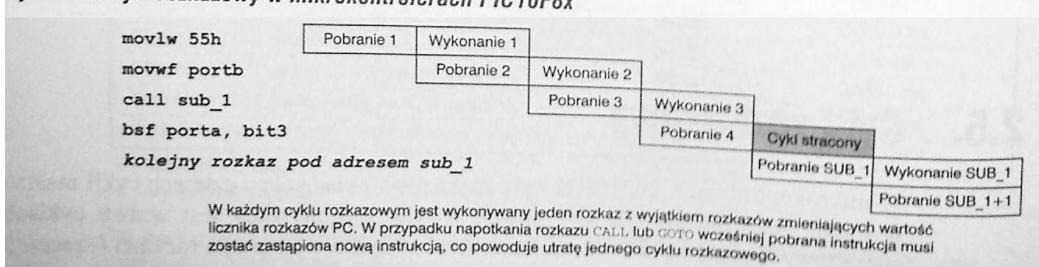
Słowo zarezerwowane to słowo (najczęściej pochodzenia angielskiego) lub jego fragment o znaczeniu określonym w definicji języka programowania, któremu nie można w programie nadać innego sensu.

Przykłady słów zarezerwowanych: ADDRESS, AND, CALL, NOP, NOT, ... (m.in. wszystkie mnemoniki i istniejące dyrektywy preprocesora)

11. INSTRUKCJE. STRUKTURA INSTRUKCJI. POTOKOWE WYKONYWANIE ROZKAZÓW (RYSUNEK)



Rys. 2.18. Cykl rozkazowy w mikrokontrolerach PIC16F8x



Rys. 2.19. Potokowe wykonywanie rozkazów

Instrukcja-zdefiniowana przez producenta, Składnia: nazwa arg1 (operand), arg2 (operand),

Cykl potokowego wykonywania rozkazów dzieli się na dwa etapy:

1. pobieranie instrukcji z pamięci,
2. wykonanie instrukcji.

Potokowość polega na jednoczesnym wykonaniu danej instrukcji i pobieraniu z pamięci instrukcji następnej. Dzięki temu w każdym cyklu rozkazowym kończy się wykonanie poprzedniego rozkazu.

W przypadku instrukcji sterujących przebiegiem programu wyjątkowo potrzebne są już **2 cykle instrukcji** (8 zegarowych)

Wszystkie mikrokontrolery mają taką samą listę 35 instrukcji. Jeden rozkaz zajmuje zawsze jedno słowo 14-bitowe.

Wykonanie instrukcji przebiega w cyklu złożonym z czterech taktów zegara:

1. dekodowanie instrukcji lub warunkowe pominięcie operacji,
2. odczyt danych lub brak operacji,
3. przetwarzanie danych (w ALU),
4. zapis wyniku lub brak operacji

Struktura instrukcji

| Rozkazy operujące na bajtach | | | |
|---|-----------|------------|-----|
| 13 | 8 | 7 | 6 |
| Kod rozkazu | d | f (FILE #) | 0 |
| d=0 rejestr przeznaczenia W d=1 rejestr przeznaczenia f f – 7-bitowy adres rejestru | | | |
| Rozkazy operujące na bitach | | | |
| 13 | 10 | 9 | 7 6 |
| Kod rozkazu | b (BIT #) | f (FILE #) | 0 |
| b – 3-bitowy numer bitu w rejestrze f – 7-bitowy adres rejestru | | | |
| Rozkazy sterujące i operujące na stałych | | | |
| 13 | 8 | 7 | 0 |
| Kod rozkazu | | k (stała) | |
| k – 8-bitowa stała | | | |
| Rozkazy CALL i GOTO | | | |
| 13 | 11 | 10 | 0 |
| Kod rozkazu | | k (stała) | |
| k – 11-bitowy adres | | | |

s. 6.1. Budowa słowa rozkazowego

12. INSTRUKCJE PRZESYŁANIA DANYCH MOVF, MOVLW, MOVWF I SWAPF

12.1. MOVF f, d

| | |
|----------------------|--|
| Funkcja: | Prześlij zawartość rejestru o adresie f |
| Argumenty: | 0 <= f <= 127; d = 0 lub 1 |
| Działanie: | W (f) -> rejestr przeznaczenia określony bitem d |
| Zmieniane znaczniki: | Z |
| Kod rozkazu: | 00 1000 dfff ffff |
| Opis: | Do rejestrów przeznaczenia przesyłana jest zawartość rejestrów o adresie f. Jeśli d = 0 to wynik jest przesyłany do rejestrów W. Jeśli d = 1 to wynik jest przesyłany do rejestrów o adresie f |

Przykład:

```
movf TMR0, 0      ; zapisz w rejestrze W zawartość rejestrów TMR0  
movf FSR, 1       ; przepisz rejestr FSR do siebie samego  
btfs STATUS, Z    ; wartość w FSR różna od zera  
goto FSR_not_0    ; wartość w FSR równa zero  
goto FSR_is_0     ; wartość w FSR równa zero
```

12.2. MOVWF f

| | |
|----------------------|--|
| Funkcja: | Prześlij zawartość rejestrów W do rejestrów o adresie f |
| Argumenty: | 0 <= f <= 127 |
| Działanie: | W -> (f) |
| Zmieniane znaczniki: | brak |
| Kod rozkazu: | 00 0000 1fff ffff |
| Opis: | Zawartość rejestrów W jest przesyłana do rejestrów o adresie f |

12.3. MOVLW k

| | |
|----------------------|---|
| Funkcja: | Prześlij stałą k do rejestrów W |
| Argumenty: | 0 <= k <= 255 |
| Działanie: | k -> W |
| Zmieniane znaczniki: | brak |
| Kod rozkazu: | 11 00xx kkkk kkkk |
| Opis: | Do rejestrów W wpisywana jest 8-bitowa stała k. |

Przykład:

```
ST_ADDR equ 0x20  
movlw .123      ; W := .123 (dziesiątkowo)  
movwf TMR0      ; TMR0 := W  
movlw ST_ADDR   ; W := 0x20 (szesnastkowo)  
movwf FSR       ; FSR := W
```

12.4. SWAPF f, d

| | |
|----------------------|--|
| Funkcja: | Zamień połówkami zawartość rejestru o adresie f |
| Argumenty: | $0 \leq f \leq 127$; $d = 0$ lub 1 |
| Działanie: | $(f)_{<3:0>}$ -> rejestr przeznaczenia $<7:4>$ określony bitem d $(f)_{<7:4>}$ -> rejestr przeznaczenia $<3:0>$ określony bitem d |
| Zmieniane znaczniki: | brak |
| Kod rozkazu: | 00 1110 dfff ffff |
| Opis: | W rejestrze o adresie f cztery mniej znaczące bity zamieniane są miejscami z czterema bardziej znaczącymi bitami. Jeśli $d = 0$ to wynik jest przesyłany do rejestr W. Jeśli $d = 1$ to wynik jest przesyłany do rejestr o adresie f . |

Przykłady:

```
REG    equ 0x20
movlw  0xAB      ; W := 0xAB
movwf  REG       ; REG := W = 0xAB
swapf  REG, f    ; REG := swapf (REG) = 0xBA
swapf  REG, w    ; W := swapf (REG) = 0xAB
```

13. INSTRUKCJE ARYTMETYCZNE ADDLW, ADDWF, INCF, SUBLW, SUBWF I DECF

13.1. ADDLW k

| | |
|----------------------|---|
| Funkcja: | Dodaj stałą k i zawartość rejestru W |
| Argumenty: | 0 <= k <= 255 |
| Działanie: | W + k -> W |
| Zmieniane znaczniki: | C, DC, Z |
| Kod rozkazu: | 11 111x kkkk kkkk |
| Opis: | Zawartość rejestru W jest dodawana do 8-bitowej stałej k. Wynik operacji jest przesyłany do W. |

Przykłady:

```
movlw    1      ; W := 1  
addlw    0xFF   ; W := 0xFF + W
```

; Po wykonaniu rozkazów: W = 0x00, C = 1, DC = 1, Z = 1

```
movlw    0x81   ; W := 0x81  
addlw    0x80   ; W := 0x80 + W
```

; Po wykonaniu rozkazów: W = 0x01, C = 1, DC = 0, Z = 0

13.2. ADDWF f, d

| | |
|----------------------|--|
| Funkcja: | Dodaj zawartość rejestru W i zawartość rejestru o adresie f |
| Argumenty: | 0 <= f <= 127; d = 0 lub 1 |
| Działanie: | W + (f) -> rejestr przeznaczenia określony bitem d |
| Zmieniane znaczniki: | C, DC, Z |
| Kod rozkazu: | 00 0111 dfff ffff |
| Opis: | Zawartość rejestru W jest dodawana do zawartości rejestru o adresie f. Jeśli d = 0 to wynik jest przesyłany do rejestru W. Jeśli d = 1 to wynik jest przesyłany do rejestru o adresie f. |

Przykłady:

```
movlw    0x17   ; W := 0x17  
movwf    FSR    ; FSR := W = 0x17  
movlw    0xC2    ; W := 0xC2  
addwf    FSR, 0 ; W := FSR + W = 0x17 + 0xC2
```

; Po wykonaniu rozkazów: FSR = 0x17, W = 0xD9, C = 0, DC = 0, Z = 0

13.3. INCF f, d

| | |
|----------------------|---|
| Funkcja: | Zwiększa o 1 zawartość rejestru o adresie f |
| Argumenty: | $0 \leq f \leq 127$; $d = 0$ lub 1 |
| Działanie: | $(f) + 1 \rightarrow$ rejestr przeznaczenia określony bitem d |
| Zmieniane znaczniki: | Z |
| Kod rozkazu: | 00 1010 dfff ffff |
| Opis: | Zawartość rejestru o adresie f jest zwiększana o 1. Jeśli $d = 0$ to wynik jest przesyłany do rejestru W. Jeśli $d = 1$ to wynik jest przesyłany do rejestru o adresie f. |

13.4. DECF f, d

| | |
|----------------------|--|
| Funkcja: | Zmniejsza o 1 zawartość rejestru o adresie f |
| Argumenty: | $0 \leq f \leq 127$; $d = 0$ lub 1 |
| Działanie: | $(f) - 1 \rightarrow$ rejestr przeznaczenia określony bitem d |
| Zmieniane znaczniki: | Z |
| Kod rozkazu: | 00 0011 dfff ffff |
| Opis: | Zawartość rejestru o adresie f jest zmniejszona o 1. Jeśli $d = 0$ to wynik jest przesyłany do rejestru W. Jeśli $d = 1$ to wynik jest przesyłany do rejestru o adresie f. |

Przykłady:

```
REG equ 0x20
movlw .254      ; W := 254
movwf REG       ; REG := W
incf REG, 1     ; REG := REG + 1 = .255 ( Z = 0 )
incf REG, 0     ; W = REG + 1 = 0 ( Z = 1 )
decf REG, 0     ; W = REG - 1 = .254 ( Z = 0 )
```

13.5. SUBLW k

| | |
|----------------------|--|
| Funkcja: | Odejmij zawartość rejestru W od stałej k |
| Argumenty: | $0 \leq k \leq 255$ |
| Działanie: | $k - W \rightarrow W$ |
| Zmieniane znaczniki: | C, DC, Z |
| Kod rozkazu: | 11 110x kkkk kkkk |
| Opis: | Od 8-bitowej stałej k odejmowana jest zawartość rejestru W. Wynik tej operacji przesyłany jest do rejestru W. Odejmowanie jest wykonywane na liczbach zapisanych w kodzie uzupełnienia do 2. |

Przykłady:

```
movlw    0x01 ; W := 0x01  
sublw    0x02 ; W := 0x02 - W
```

; Po wykonaniu rozkazów: W = 0x01, DC = 1, C = 1, Z = 0 (wynik dodatni)

```
movlw    0x02 ; W := 0x02  
sublw    0x02 ; W := 0x02 - W
```

; Po wykonaniu rozkazów: W = 0x00, DC = 1, C = 1, Z = 1 (wynik równy zero)

```
movlw    0x03 ; W := 0x03  
sublw    0x02 ; W := 0x02 - W
```

; Po wykonaniu rozkazów: W = 0xFF, DC = 0, C = 0, Z = 0 (wynik ujemny)

13.6. SUBWF f, d

| | |
|----------------------|--|
| Funkcja: | Odejmij zawartość rejestru W i zawartość rejestru o adresie f |
| Argumenty: | 0 <= f <= 127; d = 0 lub 1 |
| Działanie: | (f) - W -> rejestr przeznaczenia określony bitem d |
| Zmieniane znaczniki: | C, DC, Z |
| Kod rozkazu: | 00 0010 dfff ffff |
| Opis: | Od zawartości rejestru o adresie f odejmowana jest zawartość rejestru W. Jeśli d = 0 to wynik jest przesyłany do rejestru W. Jeśli d = 1 to wynik jest przesyłany do rejestru o adresie f. |

Przykłady:

```
; Wykonanie porównania arytmetycznego rejestru ze stałą  
REG    equ 0x20  
ST     equ .123  
. . .           ; dowolny kod zapisujący wartość rejestru REG  
movlw   ST          ; W := ST = .123  
subwf   REG, 0       ; W := ST - REG  
btfs s STATUS, Z  
goto    reg_not_equ_st ; Z = 0 ( W różny od zera → REG różny od ST)  
reg_equ_st:           ; Z = 1 ( W równy zero → REG równy ST)  
. . .  
goto    on_end  
  
reg_not_equ_st:       ; Z = 0 ( W różny od zera → REG różny od ST)  
btfs s STATUS, C  
goto    reg_above_st ; C = 0 ( W ujemny → REG większy od ST)  
reg_below_st:         ; C = 1 ( W dodatni → REG mniejszy od ST)  
. . .  
goto    on_end  
  
reg_above_st:         ; C = 0 ( W ujemny → REG większy od ST)  
. . .  
on_end:  
. . .
```

14. INSTRUKCJE LOGICZNE ANDLW, ANDWF, IORLW, IORWF, XORLW, XORWF I COMF

14.1. ANDLW k

| | |
|----------------------|--|
| Funkcja: | iiloczyn logiczny stałej k i zawartości rejestru W |
| Argumenty: | 0 <= k <= 255 |
| Działanie: | k & W -> W |
| Zmieniane znaczniki: | Z |
| Kod rozkazu: | 11 1001 kkkk kkkk |
| Opis: | Do rejestru W przesyłany jest wynik iloczynu logicznego rejestru W i 8-bitowej stałej k. |

Przykłady:

```
movlw b'10101101'; W := 10101101 (binarnie)
andlw b'00001111'; W := 00001111 & W
; Po wykonaniu rozkazów: W = b'00001101', Z = 0

movlw b'00001010'; W := 00001010 (binarnie)
andlw b'11110000'; W := 11110000 & W
; Po wykonaniu rozkazów: W = b'00000000', Z = 1
```

14.2. ANDWF f, d

| | |
|----------------------|---|
| Funkcja: | iiloczyn logiczny zawartości rejestru W i zawartości rejestru o adresie f |
| Argumenty: | 0 <= f <= 127; d = 0 lub 1 |
| Działanie: | (f) & W -> rejestr przeznaczenia określony bitem d |
| Zmieniane znaczniki: | Z |
| Kod rozkazu: | 00 0101 dfff ffff |
| Opis: | Do rejestrów przeznaczenia przesyłany jest wynik iloczynu logicznego rejestru W i rejestru o adresie f. Jeśli d = 0 to wynik jest przesyłany do rejestru W. Jeśli d = 1 to wynik jest przesyłany do rejestru o adresie f. |

Przykłady:

```
REG equ 0x20
VAL equ b'11011010'
movlw VAL ; W := VAL = b'11011010'
movwf REG ; REG := W = b'11011010'
swapf REG, f ; REG := b'10101101'
movlw b'00001111'; W := b'00001111'
andwf REG, 0 ; W := b'00001101' (starsza tetrada z VAL)
swapf REG, f ; REG := b'11011010'
movlw b'00001111'; W := b'00001111'
andwf REG, 0 ; W := b'00001010' (młodsza tetrada z VAL)
```

14.3. IORLW k

| | |
|----------------------|---|
| Funkcja: | Suma logiczna stałej k i zawartości rejestru W |
| Argumenty: | 0 <= k <= 255 |
| Działanie: | k W -> W |
| Zmieniane znaczniki: | Z |
| Kod rozkazu: | 11 1000 kkkk kkkk |
| Opis: | Do rejestru W przesyłany jest wynik sumy logicznej rejestru W i 8-bitowej stałej k. |

Przykłady:

```
movlw    b'10101101' ; W := 10101101 (binarnie)
iorlw    b'00001111' ; W := 00001111 | W
; Po wykonaniu rozkazów: W = b'10101111', Z = 0
```

```
movlw    b'00000000' ; W := 00000000 (binarnie)
iorlw    b'00000000' ; W := 00000000 | W
; Po wykonaniu rozkazów: W = b'00000000', Z = 1
```

14.4. IORWF f, d

| | |
|----------------------|--|
| Funkcja: | Suma logiczna zawartości rejestru W i zawartości rejestru o adresie f |
| Argumenty: | 0 <= f <= 127; d = 0 lub 1 |
| Działanie: | (f) W -> rejestr przeznaczenia określony bitem d |
| Zmieniane znaczniki: | Z |
| Kod rozkazu: | 00 0100 dfff ffff |
| Opis: | Do rejestrów przeznaczenia przesyłany jest wynik sumy logicznej rejestru W i rejestrów o adresie f. Jeśli d = 0 to wynik jest przesyłany do rejestru W. Jeśli d = 1 to wynik jest przesyłany do rejestrów o adresie f. |

Przykład: **IORWF RESULT, 0**

Przed wykonaniem instrukcji :

```
RESULT = 0x13
W      = 0x91
```

Po wykonaniu instrukcji :

```
RESULT = 0x13
W      = 0x93
```

14.5. COMF f, d

| | |
|----------------------|---|
| Funkcja: | Zaneguj zawartość rejestru o adresie f |
| Argumenty: | 0 <= f <= 127; d = 0 lub 1 |
| Działanie: | $\sim(f) \rightarrow$ rejestr przeznaczenia określony bitem d |
| Zmieniane znaczniki: | Z |
| Kod rozkazu: | 00 1001 dfff ffff |
| Opis: | Zawartość rejestru o adresie f jest negowana bitowo. Jeśli d = 0 to wynik jest przesyłany do rejestru W. Jeśli d = 1 to wynik jest przesyłany do rejestru o adresie f. |

Przykład:

```
movlw  b'10101101' ; W := b'10101101'  
movwf  PORTA        ; PORTA := W = b'10101101'  
call   delay_2s      ;  
comf   PORTA        ; PORTA := ~PORTA = b'01010010'
```

14.6. XORLW k

| | |
|----------------------|--|
| Funkcja: | Suma logiczna modulo 2 stałej k i zawartości rejestru W |
| Argumenty: | 0 <= k <= 255 |
| Działanie: | $W \wedge k \rightarrow W$ |
| Zmieniane znaczniki: | Z |
| Kod rozkazu: | 11 1010 kkkk kkkk |
| Opis: | Do rejestru W przesyłany jest wynik sumy logicznej modulo 2 rejestru W i 8-bitowej stałej k. |

Przykłady:

```
movlw  b'10101101' ; W := 10101101 (binarnie)  
xorlw  b'00001111' ; W := 00001111 ^ W  
; Po wykonaniu rozkazów: W = b'10100010', Z = 0  
  
movlw  b'11001110' ; W := 11001110 (binarnie)  
xorlw  b'11001110' ; W := 11001110 ^ W  
; Po wykonaniu rozkazów: W = b'00000000', Z = 1
```

14.7. XORWF f, d

| | |
|----------------------|--|
| Funkcja: | Suma logiczna modulo 2 zawartości rejestru W i zawartości rejestru o adresie f |
| Argumenty: | 0 <= f <= 127; d = 0 lub 1 |
| Działanie: | (f) ^ W -> rejestr przeznaczenia określony bitem d |
| Zmieniane znaczniki: | Z |
| Kod rozkazu: | 00 0110 dfff ffff |
| Opis: | Do rejestru przeznaczenia przesyłany jest wynik sumy logicznej modulo 2 rejestru W i rejestru o adresie f. Jeśli d = 0 to wynik jest przesyłany do rejestru W. Jeśli d = 1 to wynik jest przesyłany do rejestru o adresie f. |

Przykład: **XORWF REG, 1**

Przed wykonaniem instrukcji :

REG= 0xAF, W = 0xB5

Po wykonaniu instrukcji :

REG= 0x1A, W = 0xB5

15. INSTRUKCJE ZEROVANIA REJESTRÓW (CLRF, CLRW I CLRWDT), INSTRUKCJE PRACUJĄCE NA POJEDYNCZYCH BITACH (BCF I BSF), INSTRUKCJE PRZESUNIĘCIA (RLF I RRF)

15.1. CLRF f

| | |
|----------------------|--|
| Funkcja: | Wyzeruj zawartość rejestru o adresie f |
| Argumenty: | 0 <= f <= 127 |
| Działanie: | 00h -> (f) ; 1 -> Z |
| Zmieniane znaczniki: | Z |
| Kod rozkazu: | 00 0001 1fff ffff |
| Opis: | Wykonanie tej instrukcji spowoduje wyzerowanie zawartości rejestru o adresie f oraz ustawienie znacznika Z |

15.2. CLRW

| | |
|----------------------|--|
| Funkcja: | Wyzeruj zawartość rejestru W |
| Argumenty: | brak |
| Działanie: | 00h -> W ; 1 -> Z |
| Zmieniane znaczniki: | Z |
| Kod rozkazu: | 00 0001 0xxx xxxx |
| Opis: | Wykonanie tej instrukcji spowoduje wyzerowanie zawartości rejestru W oraz ustawienie znacznika Z |

15.3. BCF f, b

| | |
|----------------------|--|
| Funkcja: | Wyzeruj bit b rejestru o adresie f |
| Argumenty: | 0 <= f <= 127, 0 <= b <= 7 |
| Działanie: | 0 → (f) |
| Zmieniane znaczniki: | brak |
| Kod rozkazu: | 01 00bb bfff ffff |
| Opis: | Instrukcja powoduje wyzerowanie bitu b w rejestrze o adresie f |

Przykład: **BCF FLAG_REG, 7**

Przed wykonaniem instrukcji :

FLAG_REG = 0xC7

Po wykonaniu instrukcji :

FLAG_REG = 0x47

15.4. BSF f, b

| | |
|----------------------|---|
| Funkcja: | Ustaw bit b rejestru o adresie f |
| Argumenty: | 0 <= f <= 127, 0 <= b <= 7 |
| Działanie: | 1 → (f) |
| Zmieniane znaczniki: | brak |
| Kod rozkazu: | 01 01bb bfff ffff |
| Opis: | Instrukcja powoduje ustawienie bitu b w rejestrze o adresie f |

Przykład: **BSF FLAG_REG, 7**

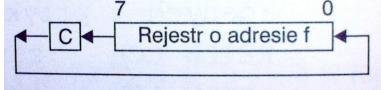
Przed wykonaniem instrukcji :

FLAG_REG = 0x0A

Po wykonaniu instrukcji :

FLAG_REG = 0x8A

15.5. RLF f, d

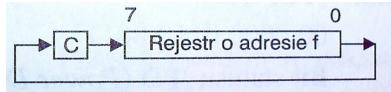
| | |
|----------------------|---|
| Funkcja: | Przesuń cyklicznie w lewo zawartość rejestru o adresie f ze znacznikiem C |
| Argumenty: | 0 <= f <= 127, d = 0 lub 1 |
| Działanie: |  |
| Zmieniane znaczniki: | C |
| Kod rozkazu: | 00 1101 dfff ffff |
| Opis: | Zawartość rejestru o adresie f jest przesuwana cyklicznie o jedną pozycję w lewo z uwzględnieniem znacznika przeniesienia C. Jeśli d = 0 to wynik jest przesyłany do rejestru W. Jeśli d = 1 to wynik jest przesyłany do rejestru o adresie f |

15.6. RRF f, d

Funkcja: Przesuń cyklicznie w prawo zawartość rejestru o adresie f ze znacznikiem C

Argumenty: $0 \leq f \leq 127$, d = 0 lub 1

Działanie:



Zmieniane znaczniki: C

Kod rozkazu: 00 1100 dfff ffff

Opis: Zawartość rejestru o adresie f jest przesuwana cyklicznie o jedną pozycję w prawo z uwzględnieniem znacznika przeniesienia C. Jeśli d = 0 to wynik jest przesyłany do rejestru W. Jeśli d = 1 to wynik jest przesyłany do rejestru o adresie f

Przesunięcie w lewo liczby 16-bitowej

```
;; Wejście
;; | TMP_B | TMP_A | - liczba 16-bitowa
;; W - wartość o jaką przesunąć (W > 0)
;; Wyjście:
;; TMP_B:TMP_A = TMP_B:TMP_A << W
;; Zmienia: LEN
rlf_16bit:
    movwf    LEN
rlf_16bit_loop:
    bcf      STATUS, C
    rlf      TMP_A, f
    rlf      TMP_B, f
    decfsz  LEN, f
    goto    rlf_16bit_loop
    return
```

Przesunięcie w prawo liczby 16-bitowej

```
;; Wejście
;; | TMP_B | TMP_A | - liczba 16-bitowa
;; W - wartość o jaką przesunąć (W > 0)
;; Wyjście:
;; TMP_B:TMP_A = TMP_B:TMP_A >> W
;; Zmienia: LEN
rrf_16bit:
    movwf    LEN
rrf_16bit_loop:
    bcf      STATUS, C
    rrf      TMP_B, f
    rrf      TMP_A, f
    decfsz  LEN, f
    goto    rrf_16bit_loop
    return
```

15.7. CLRWDT

Funkcja: Wyzeruj zawartość licznika układu Watchdog

Argumenty: brak

Działanie: 00h -> WDT

00h -> preskaler

1 -> !TO

1 -> !PD

Zmieniane znaczniki: !TO, !PD

Kod rozkazu: 00 0000 0110 0100

Opis: Wykonanie tej instrukcji spowoduje wyzerowanie zawartości licznika watchdog.

Zerowany jest również preskaler, jeżeli jest przypisany do WDT.

Nie zmieniany jest współczynnik podziału preskalera.

Bity !TO i !PD są ustawiane.

16. INSTRUKCJE STERUJĄCE: SKOKU BEZWARUNKOWEGO (GOTO), BRAK OPERACJI (NOP), PRZEJŚCIE W TRYB BEZCZYNNOŚCI (SLEEP). TRYB UŚPIENIA SLEEP

16.1. GOTO k

| | |
|----------------------|---|
| Funkcja: | Skok bezwarunkowy do podanej etykiety |
| Argumenty: | $0 \leq k \leq 2047$ |
| Działanie: | $k \rightarrow PC <10:0>$ $PCLATH<4:3> \rightarrow PC<12:11>$ |
| Zmieniane znaczniki: | brak |
| Kod rozkazu: | 10 1kkk kkkk kkkk |
| Opis: | Bezwarunkowy skok do adresu określonego w argumencie rozkazu. Do licznika rozkazów PC na pozycje $PC<10..0>$ jest przesyłanych jedenaście bitów bezpośredniego adresu z argumentu rozkazu. Dwa najbardziej znaczące bity licznika rozkazów są ładowane z rejestru $PCLATH<4:3>$. Rozkaz jest wykonywany w dwóch cyklach rozkazowych. |

Przykład: GOTO THERE

Po wykonaniu instrukcji :

PC = Address(THERE)

16.1. GOTO k

| | |
|----------------------|---|
| Funkcja: | Skok bezwarunkowy do podanej etykiety |
| Argumenty: | $0 \leq k \leq 2047$ |
| Działanie: | $k \rightarrow PC <10:0>$ $PCLATH<4:3> \rightarrow PC<12:11>$ |
| Zmieniane znaczniki: | brak |
| Kod rozkazu: | 10 1kkk kkkk kkkk |
| Opis: | Bezwarunkowy skok do adresu określonego w argumencie rozkazu. Do licznika rozkazów PC na pozycje $PC<10..0>$ jest przesyłanych jedenaście bitów bezpośredniego adresu z argumentu rozkazu. Dwa najbardziej znaczące bity licznika rozkazów są ładowane z rejestru $PCLATH<4:3>$. Rozkaz jest wykonywany w dwóch cyklach rozkazowych. |

Przykład: GOTO THERE

Po wykonaniu instrukcji :

PC = Address(THERE)

16.2. NOP

| | |
|----------------------|--|
| Funkcja: | Nie wykonuj żadnej operacji przez jeden cykl rozkazowy |
| Argumenty: | brak |
| Działanie: | (PC) +1 -> PC |
| Zmieniane znaczniki: | brak |
| Kod rozkazu: | 00 0000 0xx0 0000 |
| Opis: | Nie wykonuj żadnej operacji. |

Przykład: **HERE: NOP**

Przed wykonaniem instrukcji :

PC = address(HERE)

Po wykonaniu instrukcji :

PC = address(HERE + 1)

16.3. SLEEP

| | |
|----------------------|---|
| Funkcja: | Przejdź w tryb uśpienia |
| Argumenty: | brak |
| Działanie: | 00h -> WDT 00h -> preskaler 1 -> !TO 0-> !PD |
| Zmieniane znaczniki: | !TO, !PD |
| Kod rozkazu: | 00 0000 0110 0011 |
| Opis: | Wykonanie tej instrukcji spowoduje wyzerowanie bitu !PD, ustawienie bitu !TO oraz wyzerowanie zawartości licznika watchdog i preskalera, a następnie przejście w stan uśpienia. W trybie uśpienia zatrzymywany jest oscylator mikrokontrolera. |

16.4. Tryb uśpienia SLEEP

Tryb uśpienia (Sleep Mode, Power-down Mode) jest trybem, w którym mikrokontroler jest w najbardziej energoszczędznym stanie.

Generator sygnału zegarowego jest wyłączony, więc impulsy zegarowe nie dochodzą do mikrokontrolera.

Przejście w tryb uśpienia

Tryb uśpienia jest włączany za pomocą **instrukcji SLEEP**.

Jeżeli licznik WDT jest włączony, to zostanie on wyzerowany w momencie przejścia mikrokontrolera w tryb uśpienia, ale będzie nadal pracował.

Bit !PD w rejestrze STATUS zostanie wyzerowany, a bit !TO ustawiony na jedynkę.

Porty wejścia/wyjścia zachowają swój **wcześniejszego stan**, przed wykonania instrukcji SLEEP.

Aby zapewnić jak najmniejszy pobór prądu w tym trybie, wszystkie wyprowadzenia powinny być na poziomie dodatniego lub ujemnego napięcia zasilania bez możliwości poboru prądu z tych wyprowadzeń przez układy zewnętrzne.

Także układy wewnętrzne mikrokontrolera, które pobierają minimalny prąd w stanie uśpienia powinny zostać wyłączone.

Wyprowadzenie zerujące **MCLR** powinno być w wysokim stanie logicznym.

Niektóre funkcje mikrokontrolera (np. WDT, BOR) powodujące pobór małego prądu w stanie uśpienia mogą zostać wyłączone podczas programowania mikrokontrolera za pomocą bitów rejestru konfiguracyjnego.

Mikrokontroler może zostać **wyprowadzony ze stanu uśpienia** poprzez wystąpienie jednego z następujących zdarzeń:

- uruchomienie dowolnego trybu zerowania mikrokontrolera
- obudzenie mikrokontrolera przez układ WDT (WDT Wake-up)
- zgłoszenie przerwania przez dowolny moduł, który może ustawić znacznik przerwania w trybie uśpienia:
 - wejście RB0/INT
 - zmiana stanu wyprowadzeń RB7:RB4
 - komparatory
 - przetwornik A/C
 - układ czasowy TIMER1
 - synchroniczny port szeregowy SSP
 - moduł przechwytywania (CCP)

17. INSTRUKCJE STERUJĄCE: SKOKU WARUNKOWEGO (BTFSC, BTFSS, DECFSZ, INCFSZ)

17.1. INCFSZ f, d / DECFSZ f, d

Funkcja: Zwiększa / zmniejsza o 1 zawartość rejestru o adresie f,
jeśli wynik = 0 to pomiń kolejną instrukcję

Argumenty: $0 \leq f \leq 127$, $d = 0$ lub 1

Działanie: $(f) +1 \rightarrow$ rejestr przeznaczenia określony bitem d,
jeśli wynik=0, to pomiń kolejną instrukcję: $PC + 2 \rightarrow PC$
w przeciwnym wypadku: $PC + 1 \rightarrow PC$

Zmieniane znaczniki: brak

Kod rozkazu: 00 1111 dfff ffff / 00 1011 dfff ffff

Opis: Zawartość rejestru o adresie f jest zwiększana (zmniejszana) o 1.
Jeśli $d = 0$ to wynik jest przesyłany do rejestru W. Jeśli $d = 1$ to wynik jest przesyłany do rejestru o adresie f.

Jeśli w wyniku inkrementacji / dekrementacji do rejestru przeznaczenia zostanie przesłana wartość **zerowa**, to następna instrukcja nie zostanie wykonana.

Ponieważ w trakcie wykonywania rozkazu INCFSZ / DECFSZ następna instrukcja została już pobrana do wykonania, to zamiast niej wykonana zostanie instrukcja NOP (rozkaz INCFSZ / DECFSZ wykonuje się w trakcie dwóch cykli rozkazowych)

Jeśli natomiast wynik operacji będzie różny od 0, to rozkaz INCFSZ / DECFSZ jest wykonywany w jednym cyklu (brak modyfikacji licznika rozkazów)

17.2. BTFSC f, b

| | |
|----------------------|---|
| Funkcja: | Testuj bit b w rejestrze o adresie f, jeśli wyzerowany to pomiń następną instrukcję |
| Argumenty: | 0 <= f <= 127, 0 <= b <= 7 |
| Działanie: | Jeśli $f_{} = 0$ to $PC + 2 \rightarrow PC$ w przeciwnym wypadku: $PC + 1 \rightarrow PC$ |
| Zmieniane znaczniki: | brak |
| Kod rozkazu: | 01 10bb bfff ffff |

Opis:

Jeśli bit b w rejestrze o adresie f jest wyzerowany, to następna instrukcja nie zostanie wykonana.

Ponieważ w trakcie wykonywania rozkazu BTFSC następna instrukcja została już pobrana do wykonania, to zamiast niej wykonana zostanie instrukcja NOP (rozkaz BTFSC wykonuje się w trakcie dwóch cykli rozkazowych)

Jeśli natomiast bit b w rejestrze f jest ustawiony, to rozkaz BTFSC jest wykonywany w jednym cyklu (brak modyfikacji licznika rozkazów)

17.3. BTFSS f, b

| | |
|----------------------|---|
| Funkcja: | Testuj bit b w rejestrze o adresie f, jeśli ustawiony to pomiń następną instrukcję |
| Argumenty: | 0 <= f <= 127, 0 <= b <= 7 |
| Działanie: | Jeśli $f_{} = 1$ to $PC + 2 \rightarrow PC$ w przeciwnym wypadku: $PC + 1 \rightarrow PC$ |
| Zmieniane znaczniki: | brak |
| Kod rozkazu: | 01 11bb bfff ffff |

Opis:

Jeśli bit b w rejestrze o adresie f jest ustawiony, to następna instrukcja nie zostanie wykonana.

Ponieważ w trakcie wykonywania rozkazu BTFSS następna instrukcja została już pobrana do wykonania, to zamiast niej wykonana zostanie instrukcja NOP (rozkaz BTFSS wykonuje się w trakcie dwóch cykli rozkazowych)

Jeśli natomiast bit b w rejestrze f jest wyzerowany, to rozkaz BTFSS jest wykonywany w jednym cyklu (brak modyfikacji licznika rozkazów)

18. INSTRUKCJE STERUJĄCE: WYWOŁANIA PODPROGRAMU (CALL), POWROTU Z PODPROGRAMU (RETURN, RETFIE, RETLW)

18.1. CALL k

Funkcja: Wywołaj podprogram (procedurę) spod adresu programu k

Argumenty: $0 \leq k \leq 2047$

Działanie: $PC+1 \rightarrow STOS$

$k \rightarrow PC <10:0>$

$PCLATH<4:3> \rightarrow PC<12:11>$

Zmieniane znaczniki: brak

Kod rozkazu: 10 0kkk kkkk kkkk

Opis:

W pierwszej kolejności na stos odkładany jest 13-bitowy adres powrotu, a następnie do licznika rozkazów PC na pozycje $PC<10:0>$ jest przesyłanych jedenaście bitów bezpośredniego adresu z argumentu rozkazu. Dwa najbardziej znaczące bity licznika rozkazów są ładowane z rejestru $PCLATH<4:3>$.

Rozkaz jest wykonywany w dwóch cyklach rozkazowych.

18.2. RETURN

Funkcja: Powrót z podprogramu (procedury)

Argumenty: brak

Działanie: $STOS \rightarrow PC$

Zmieniane znaczniki: brak

Kod rozkazu: 00 0000 0000 1000

Opis:

13-bitowy adres posrotny jest zdejmowany ze stosu i wpisywany do licznika rozkazów PC.

Rozkaz jest wykonywany w dwóch cyklach rozkazowych.

18.3. RETFIE

Funkcja: Powrót z procedury obsługi przerwania

Argumenty: brak

Działanie: $STOS \rightarrow PC$

$1 \rightarrow GIE$

Zmieniane znaczniki: brak

Kod rozkazu: 00 0000 0000 1001

Opis:

13-bitowy adres jest zdejmowany ze stosu i wpisywany do licznika rozkazów PC. Automatycznie zostaje włączony układ przerwań (bit $GIE=1$)

Rozkaz jest wykonywany w dwóch cyklach rozkazowych.

18.4. RETLW k

| | |
|----------------------|---|
| Funkcja: | Powrót z podprogramu (procedury) ze stałą w rejestrze W |
| Argumenty: | 0 <= k <= 255 |
| Działanie: | k → W STOS → PC |
| Zmieniane znaczniki: | brak |
| Kod rozkazu: | 11 01xx kkkk kkkk |

Opis:

13-bitowy adres jest zdejmowany ze stosu i wpisywany do licznika rozkazów PC. Do rejestru W jest ładowana 8-bitowa stała k.

Rozkaz jest wykonywany w dwóch cyklach rozkazowych.

Przykład:

```
HERE: CALL TABLE
...
TABLE:
    ADDWF PCL, f
    RETLW    k1    ; początek tablicy
    RETLW    k2    ;
    ...
    RETLW    kn    ; koniec tablicy
```

19. ORGANIZACJA PODPROGRAMÓW

Program w asemblerze, dający ten sam kod wynikowy, może być napisany na wiele sposobów. Źle napisany program po pewnym czasie będzie nieczytelny nawet dla samego autora, będzie więc programem "jednokrotnego użytku".

Dobry program, napisany w sposób czytelny i zrozumiały, będzie łatwy do uruchomienia a jego modyfikacja lub rozbudowa będzie możliwa nawet po długim czasie. Będzie on mógł być również łatwo wykorzystany, w całości lub we fragmentach, przez innego programistę.

Zapis programu w asemblerze składa się z instrukcji sterujących przebiegiem tłumaczenia programu, dyrektyw asemblera oraz rozkazów procesora z ewentualnymi parametrami.

W programie mogą również występować komentarze, umieszczone po znaku średnika. Tekst po średniku jest ignorowany.

Jednemu rozkazowi procesora odpowiada jeden wiersz programu.

Zalecane jest wyrównanie programu w kolumnach.

Kod rozkazu piszemy po znaku tabulacji, tak aby pozostawić miejsce na ewentualne etykiety. Jeśli często stosowane będą długie etykiety, to można użyć więcej niż jednej tabulacji.

Jeśli wyjątkowo długa etykieta nie mieści się w pierwszym polu, to może ona tworzyć samodzielną linię.

Komentarze powinny być również wyrównane w kolumnach. Warto również zastosować tabulację pomiędzy kodem rozkazu i ewentualnymi parametrami.

Dzięki temu łatwiej będzie znaleźć konkretny rozkaz.

Stosując nazwy symboliczne warto je dobierać tak, aby odzwierciedlały one funkcję pełnioną przez stałą lub zmienną którą oznaczają.

Nazwy składające się z dwóch wyrazów można oddzielić znakiem podkreślenia dla polepszenia ich czytelności.

Komentarze powinny tłumaczyć co dzieje się w danym fragmencie programu.

Nie powinny one powtarzać informacji która jest zawarta w samym zapisie rozkazu.

Ważną sprawą jest używanie podprogramów (procedur).

Podprogram powinien być dobrze opisany, tak aby można było go później użyć bez konieczności analizowania kodu. Dla podprogramów z parametrami należy podać sposób ich przekazywania lub zwracania.

Warto również podać jakie rejesty są używane w podprogramie aby można było to uwzględnić przy wywołaniu podprogramu.

W typowym przypadku podprogram jest fragmentem kodu wykorzystywanym wielokrotnie (być może z różnymi parametrami). Warto jednak używać podprogramów (nawet jeśli miałyby być one wywołane tylko jednokrotnie) dla polepszenia czytelności programu. Szczególnym przypadkiem podprogramu jest obsługa przerwania.

20. OBSŁUGA PRZERWAŃ

Można dodać jakiś opis ogólnego działania przerwań, bitów zezwoleń i wystąpień przerwań, bitów: globalnego zezwolenia na przerwania i bitu zezwolenia na przerwania od urządzeń peryferyjnych – **patrz zadanie 42**

```
; Procedura obsługi przerwania

w_temp      equ 0x7D ; deklaracja zmiennych tymczasowych
status_temp  equ 0x7E
pclath_temp equ 0x7F

org 0x0004          ; wektor przerwań
goto Proc_INT       ; przejście do procedury obsługi przerwań
Proc_INT           ; początek procedury obsługi przerwania
    movwf w_temp      ; zachowanie wartości rejestrów
    swapf STATUS, W
    movwf status_temp
    movf PCLATH, W
    movwf pclath_temp

; Wykrycie źródła przerwania
banksel INTCON
btfsC INTCON, TOIF   ; przerwanie od TIMERO ?
goto TO_INT          ; skok do procedury obsługi przerwania od TIMERO
btfsC INTCON, INTF   ; przerwanie zewnętrzne ?
goto INT_INT          ; skok do procedury obsługi zewnętrznego przerwania
btfsC INTCON, RBIF   ; przerwanie od zmian na POTRB ?
goto PORTB_INT        ; skok do procedury obsługi przerwania RB
btfsC EECON1, EEIF    ; przerwanie po zakończonym zapisie do pamięci EEPROM
goto EEPROM_INT        ; skok do proc. obsługi przerwania od EEPROM

; wykrycie innych przerwań
...
INT_ERR: goto INT_ERR ; pułapka na wypadek kiedy przerwanie przyjęto,
; a nie wykryto źródła
```

```

TO_INT           ; obsługa przerwania TIMER0
    bcf   INTCON, TOIF   ; zerowanie znacznika zgłoszenia przerwania
    goto  END_INT
INT_INT          ; obsługa zewnętrznego przerwania
    bcf   INTCON, INTF   ; zerowanie znacznika zgłoszenia przerwania
    goto  END_INT
PORTB_INT        ; obsługa przerwania RB
    bcf   INTCON, RBIF   ; zerowanie znacznika zgłoszenia przerwania
    goto  END_INT
EEPROM_INT       ; obsługa przerwania od EEPROM
    bcf   EECON1, EEIF   ; zerowanie znacznika zgłoszenia przerwania
    goto  END_INT        ; obsługa innych przerwań
END_INT
    movf  pclath_temp, W  ; przywrócenie wartości rejestrów
    movwf PCLATH
    swapf status_temp, W
    movwf STATUS
    swapf w_temp, F
    swapf w_temp, W
    retfie                ; wyjście z procedury obsługi przerwania

```

21. INSTRUKCJA WYJŚCIA Z PODPROGRAMU RETLW (ORGANIZACJA TABLIC W PAMIĘCI PROGRAMU)

21.1. RETLW k

Funkcja: Powrót z podprogramu (procedury) ze stałą w rejestrze W

Argumenty: $0 \leq k \leq 255$

Działanie: $k \rightarrow W$
STOS \rightarrow PC

Zmieniane znaczniki: brak

Kod rozkazu: 11 01xx kkkk kkkk

Opis:

Powrót z podprogramu. 13-bitowy adres jest zdejmowany ze stosu i wpisywany do licznika rozkazów PC. Do rejestr W jest ładowana 8-bitowa stała k będąca argumentem rozkazu. Rozkaz jest wykonywany w ciągu dwóch cykli.

Rozkaz RETLW może być używany do tworzenia w pamięci programu tablic z 8-bitowymi stałymi. Przed wywołaniem podprogramu **tablica**, do rejestrów W należy wpisać numer odczytywanego elementu tablicy, czyli przesunięcie względem adresu **tablica**. W tym przykładzie rejestr W=2.

main:

```

    movlw .2      ; W = 2, indeks komórki tablicy którą chcemy odczytać
                  ; W tym miejscu W = tablica[2] = 0x23

```

tablica:

```

    addwf PCL, f
    retlw 0x10    ; [0]
    retlw 0x12    ; [1]
    retlw 0x23    ; [2]
    retlw 0x33    ; [3]

```

; Skrócona deklaracja tablicy (równoważna powyżej) z użyciem dyrektywy preprocesora `dt`
tablica:

```
addwf PCL, f  
dt      0x10, 0x12, 0x23, 0x33
```

22. CZAS WYKONYWANIA INSTRUKCJI STERUJĄCYCH

$$T_{instr} = \frac{4}{f_{zegara}}$$

Czas wykonania jednej instrukcji to:

Większość instrukcji trwa jeden cykl zegarowy, czyli np. dla prostych instrukcji takich jak NOP, MOVLW, BSF czas wykonania dla taktowania 4MHz to 1us.

Jest kilka instrukcji które wykonują się przez dwa cykle zegarowe. Ogólnie: NOP, SLEEP – wykonywane w jednym cyklu, GOTO, CALL, RETFIE, RETLW, RETURN wykonywane w 2 cyklach, DECFSZ, INCFSZ, BTFSC, BTFSZ – to różnie, 1 lub 2 cykle

23. MIKROKONTROLERY RODZINY PIC-16. CECHY MIKROKONTROLERÓW PIC16. DODATKOWE FUNKCJE MIKROKONTROLERÓW PIC16

Architektura zbudowana na podstawie **architektury harwardzkiej** i oparta jest na **architekturze RISC**, a także posiada jej wszystkie typowe cechy.

Do tych cech należą:

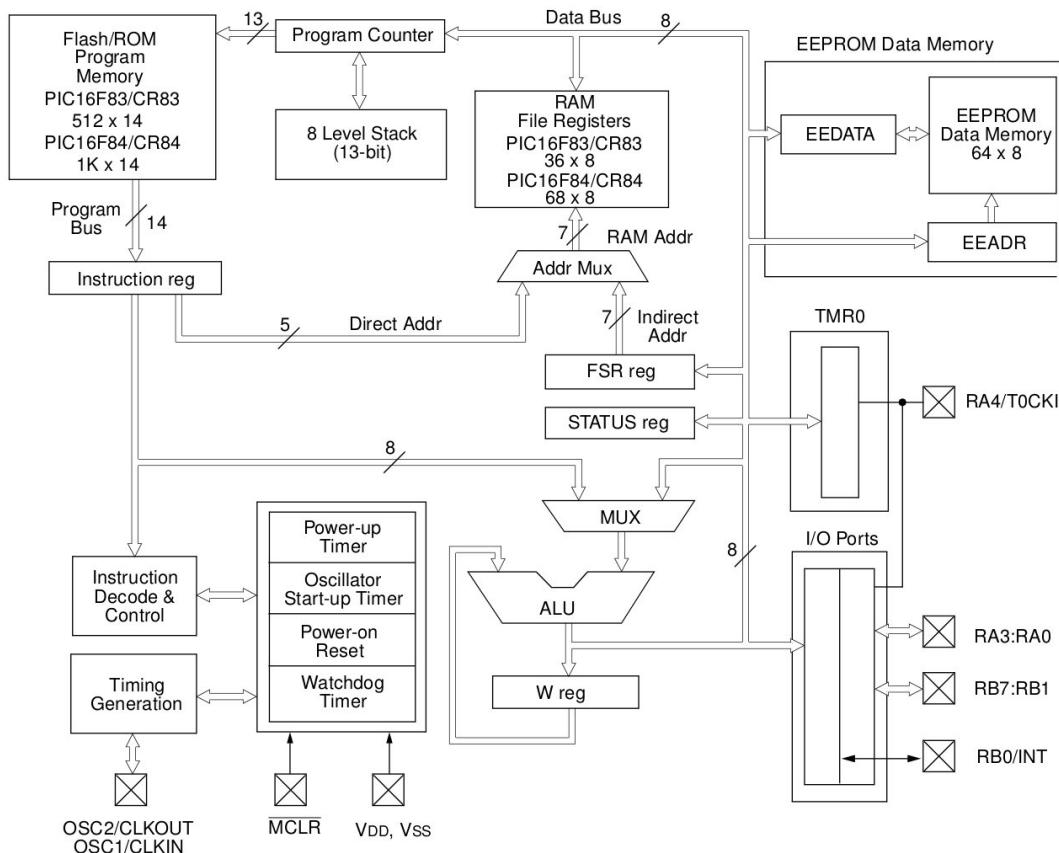
- regularny format instrukcji,
- zredukowana lista rozkazów,
- jednakowy czas wykonania instrukcji,
- duża liczba rejestrów roboczych,
- proste tryby adresowania,
- wykonanie operacji arytmetycznych i logicznych oddzielono od przesyłania do pamięci

Dodatkowe funkcje:

- maksymalna częstotliwość zegara to 20 MHz dla procesorów z pamięcią programu typu EPROM oraz 10 MHz dla procesorów z pamięcią programu typu EEPROM i FLASH;
- 14-bitowa długość słowa rozkazu;
- 8-bitowa długość słowa pamięci danych oraz rejestrów sterujących;
- 8-poziomowy stos programowy;
- od 1 do 3 układów czasowych z 8-bitowym podzielnikiem wstępny (preskalerem);
- obsługa od 4 do 14 źródeł wywołujących przerwania;
- ponad 1000000 cykli programowania pamięci danych EEPROM;
- zachowanie zawartości pamięci danych EEPROM ponad 40 lat;
- wykonane w technologii CMOS;
- szeroki zakres napięć zasilających oraz niewielki pobór prądu (poniżej 2 mA dla 5 V i 4 MHz, 15 µA dla 2 V i 32 kHz oraz poniżej 1 µA w stanie uśpienia)
- POWER-ON RESET czyli zerowanie połączeniu zasilania;
- POWER-UP TIMER tzn. przedłużenie sygnału zerowania połączeniu zasilania o 72 ms w celu uniknięcia niepożądanych zachowań układów współpracujących np. wyświetlacza LCD ze sterownikiem HD 77480;
- OSCILLATOR START-UP TIMER układ do przedłużenia zerowania wewnętrznego procesora o 1024 takty sygnału zegarowego;

- WATCHDOG TIMER - niezależny układ czasowy wewnętrz procesora, który w przypadku jego zawieszenia zeruje go (informacja o źródle wyzerowania jest pamiętana w odpowiednim rejestrze);
- CODE-PROTECTION - bit zabezpieczający program źródłowy przed odczytem;
- możliwość wprowadzenia procesora w sposób programowy w stan „uśpienia” (ang. SLEEP MODE), tzn. zawieszenie wszystkich czynności (z zapamiętaniem zawartości rejestrów);
- powrót ze stanu SLEEP następuje po wystąpieniu przerwania lub wykonania dowolnego zerowania;
- możliwość wyboru oscylatora (rezonator kwarcowy lub ceramiczny, taktowanie sygnałem zewnętrznym, zewnętrzny oscylator RC);
- wbudowany system programowania szeregowego, programowanie odbywa się poprzez dwa wyprowadzenia mikrokontrolera (RB6, RB7)

24. SCHEMAT BLOKOWY MIKROKONTROLERÓW PIC16F8x.



25. MAGISTRALE (SZYNY) MIKROKONTROLERÓW

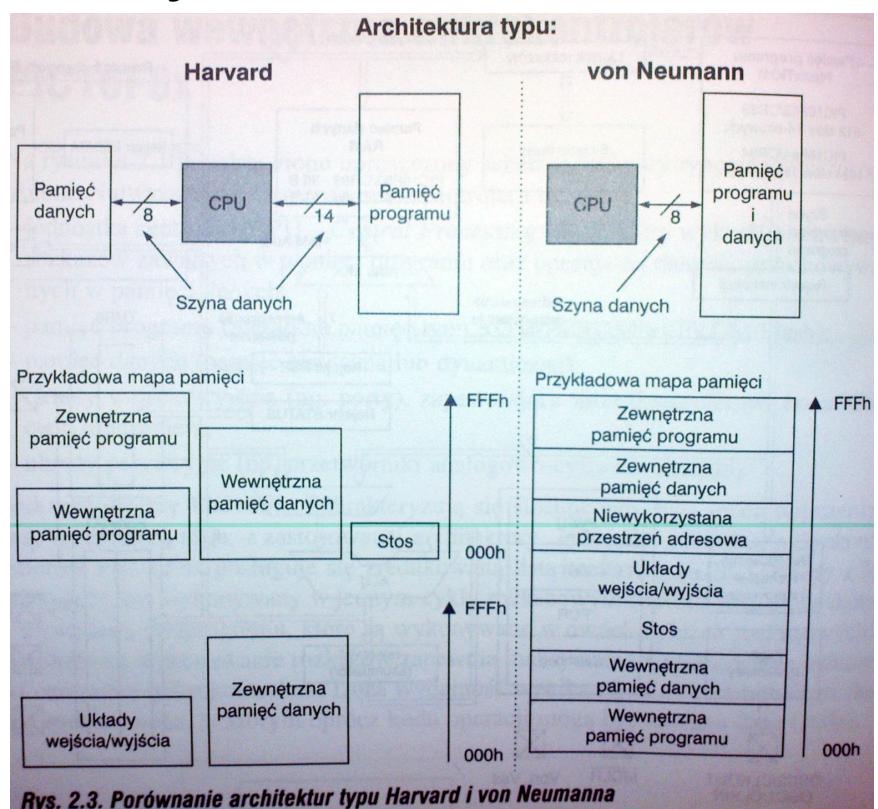
PIC16F8x. Potokowe wykonanie rozkazów

Rozdzielenie magistrali dostępu umożliwia jednoczesny dostęp do obu pamięci, co pozwala uniknąć konfliktów odwołań i skrócić cykl przetwarzania instrukcji.

Niezależne magistrale dostępu pozwalają także na stosowanie słowa rozkazu innej szerokości niż słowo danych.

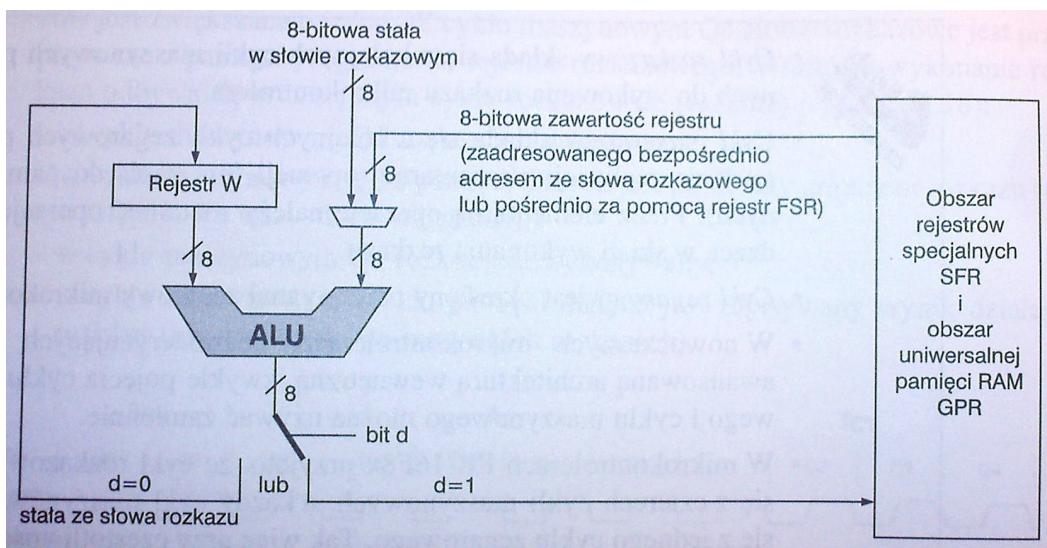
Możliwość ta wykorzystana została w mikrokontrolerach PIC16, w których magistrala danych jest 8-bitowa, a magistrala dostępu do pamięci programu – 14-bitowa. Zwiększenie szerokości magistrali pamięci programu pozwoliło na umieszczenie w słowie rozkazowym 8-bitowej stałej i umożliwia pobranie z pamięci programu całego 14-bitowego słowa rozkazowego przy jednym odwołaniu do pamięci.

Potokowe wykonanie rozkazów: patrz zadanie 11



Rys. 2.3. Porównanie architektur typu Harvard i von Neumann

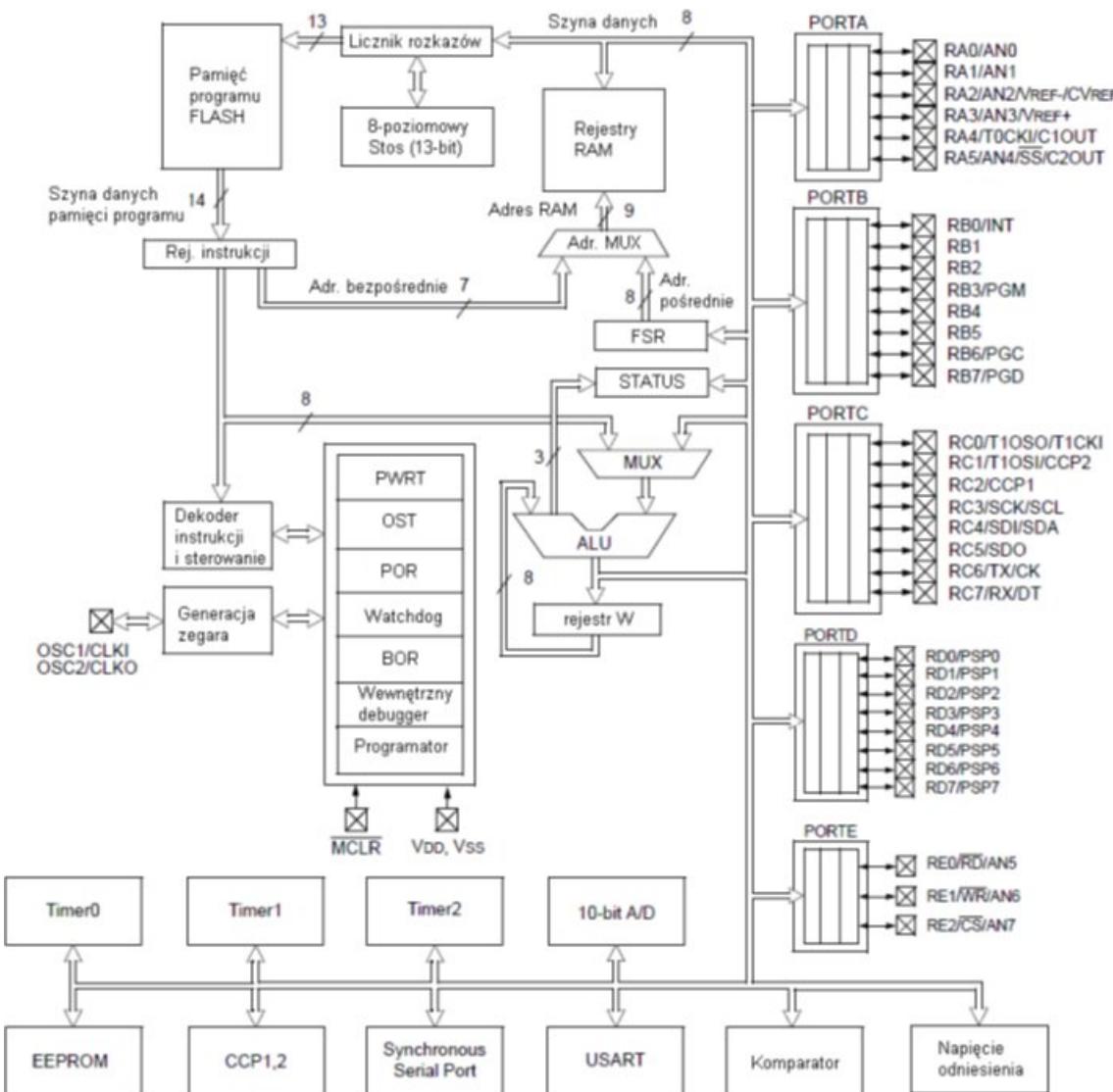
26. JEDNOSTKA ARYTMETYCZNO-LOGICZNA



Rys. 2.20. Schemat blokowy najbliższego otoczenia ALU

Jeden z operandów instrukcji zawsze w rejestrze W i spełnia on role akumulatora. Drugi może być jednym z rejestrów pamięci danych lub stała pobrana z kodu instrukcji. Instrukcje są symetryczne. Gdy d=0 to wynik umieszczany jest w rejestrze W, jeśli d=1 to trafia do rejestrów z obszaru pamięci danych. Instrukcje wykonywane jednostką ALU mogą modyfikować znaczniki w rejestrze STATUS (znacznik przeniesienia C, znacznik pomocniczego przeniesienia DC i znacznik wyniku zerowego Z)

27. SCHEMAT BLOKOWY MIKROKONTROLERA PIC16F877A.



28. NAPIĘCIE ZASILANIA TRYBY PRACY GENERATORA SYGNAŁU ZEGAROWEGO. REZONATORY KWARCOWE I CERAMICZNE. ZEWNĘTRZNY GENERATOR SYGNAŁU ZEGAROWEGO. OSCYLATOR RC

Napięcie zasilania

Tablica 2.1. Typy pamięci programu i odpowiadające im oznaczenia mikrokontrolerów

| Typ pamięci programu | Zakres napięcia zasilającego: 4,5...5,5 V | 2...5,5 V |
|----------------------|--|------------|
| ROM | PIC16CR8x | PIC16LCR8x |
| Flash | PIC16F8x | PIC16LF8x |

Standardowe napięcie zasilające dla mikrokontrolerów PIC16 wynosi 5V. Jednak mikrokontrolery PIC16 mogą pracować w znacznie szerszym zakresie napięć wynoszącym od 2 do 6 V. Należy również pamiętać o tym, że mikrokontrolery pracujące z sygnałem zegarowym o częstotliwości 20 MHz, powinny być zasilane napięciem z przedziału od 4,5 do 6 V, a mikrokontrolery pracujące z częstotliwością 4 MHz – napięciem powyżej 4 V. Wyjątkiem jest seria mikrokontrolerów PIC16 oznaczona symbolem L (np. PIC16LF84), która może pracować już przy napięciu zasilającym równym 2V.

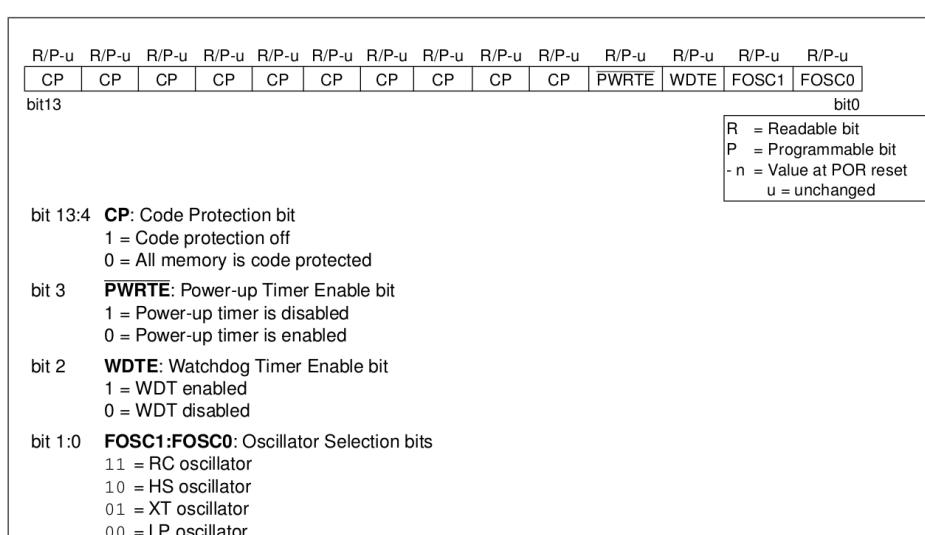
Generatory sygnału zegarowego

Mikrokontrolery PIC16 są przystosowane do pracy z 4 typami generatorów sygnału zegarowego:

1. rezonator kwarcowy o niskiej częstotliwości (LP),
2. standardowy rezonator kwarcowy (XT),
3. rezonator kwarcowy wysokiej częstotliwości (HS),
4. zewnętrzny układ generatora RC (RC).

FIGURE 8-2: CONFIGURATION WORD - PIC16F83 AND PIC16F84

Wyboru generatora dokonuje się za pomocą dwóch bitów konfiguracyjnych **FOSC1, FOSC0** w rejestrze konfiguracyjnym dostępnym tylko podczas programowania mikrokontrolera.



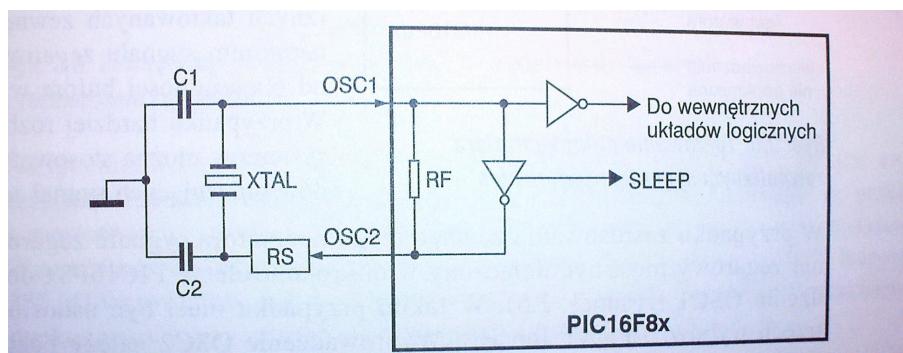
Tablica 2.2. Tryby pracy generatora sygnału zegarowego w mikrokontrolerach PIC16F8x

| FOSC1 | FOSCO | Tryb pracy | Zakres częstotliwości sygnału zegarowego | Uwagi |
|-------|-------|--|--|--|
| 1 | 1 | RC (generator R/C) | do 8/20 MHz | Niski koszt elementów zewnętrznych – wymagany jest tylko zewnętrzny rezystor i kondensator. Mała stabilność i dokładność częstotliwości. |
| 1 | 0 | HS (generator kwarcowy o wysokiej częstotliwości) | do 8/20 MHz | Stosowany w aplikacjach wymagających dużej szybkości. Pobór prądu największy z trzech trybów oscylatorów kwarcowych. |
| 0 | 1 | XT (standardowy generator kwarcowy) | do 4 MHz Średni pobór prądu. | Standardowa częstotliwość. |
| 0 | 0 | LP (rezonator kwarcowy) | do 500 kHz | Niska częstotliwość i pobór prądu (najniższy z trzech trybów oscylatorów kwarcowych). |

Różnica pomiędzy trybami HS, XT i LP jest wynikiem zmiany wzmacnienia wewnętrznego wzmacniacza operacyjnego w mikrokontrolerze, co umożliwia uzyskanie różnych zakresów częstotliwości pracy generatora zegarowego.

REZONATOR KWARCOWY I CERAMICZNY:

W trybach XT, LP i HS rezonator kwarcowy lub ceramiczny jest dołączony do wyprowadzeń OSC1 i OSC2 w celu stabilizacji częstotliwości sygnału zegarowego. Wewnętrzny generator mikrokontrolera wymaga zastosowania rezonatora o rezonansie równoległym:

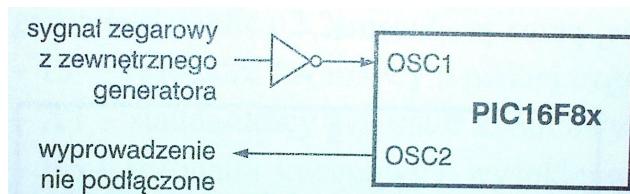


Rys. 2.4. Układ generatora sygnału zegarowego z oscylatorem kwarcowym lub rezonatorem ceramicznym

Stosowanie rezonatorów szeregowych może spowodować różnicę częstotliwości generatora w stosunku do podanej przez producenta.

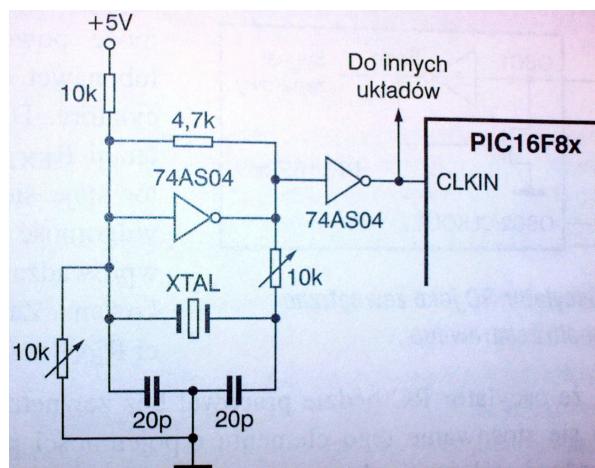
Jeżeli mikrokontroler jest sterowany sygnałem zegara ze źródła zewnętrznego, należy upewnić się, że w ustawieniach generatora jest wybrany tryb LP, XT lub HS. Zewnętrzny sygnał zegarowy powinien być dołączony do wyprowadzenia mikrokontrolera oznaczonego OSC1. Aby zredukować zakłócenia, wyprowadzenie OSC2 może zostać podłączone przez rezystor do masy, jednak może to spowodować zwiększenie poboru prądu przez mikrokontroler.

Podłączenie zewnętrznego źródła sygnału zegarowego:

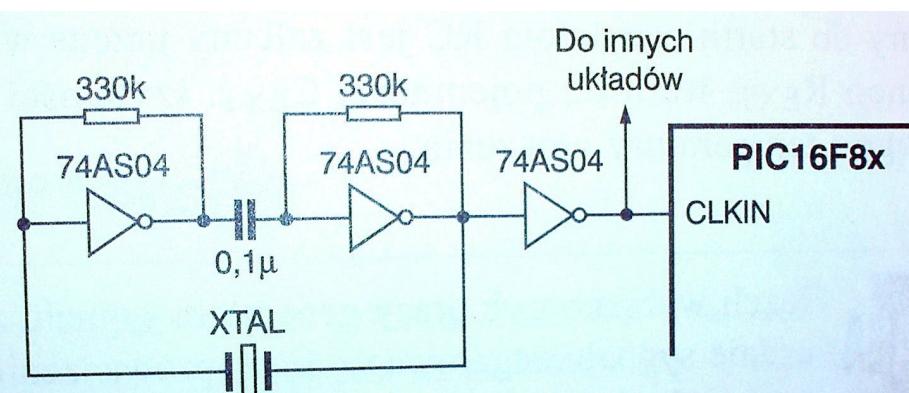


Rys. 2.5. Taktowanie mikrokontrolera zewnętrznym sygnałem zegarowym

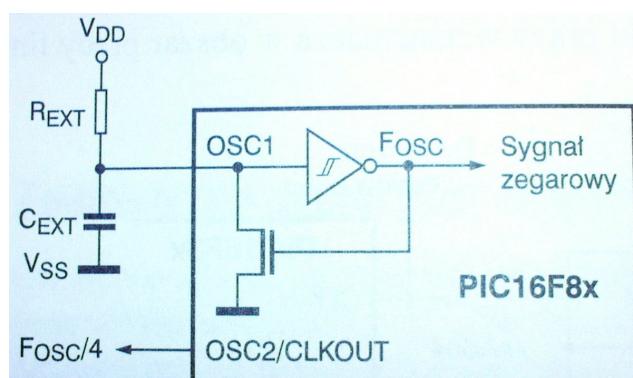
Zewnętrzny układ generatora z rezonansem równoległy:



Zewnętrzny układ generatora z rezonansem szeregowym:



OSCYLATOR RC:



Rys. 2.8. Oszylator RC jako zewnętrzne źródło sygnału zegarowego

Wartość rezystora zewnętrznego REXT poniżej 2,2 k Ω może spowodować niestabilność pracy generatora lub jego zatrzymanie. Wartości rezystancji REXT powyżej 1 M Ω powodują wrażliwość układu na zakłócenia, wilgotność i upływności. Zalecana przez producenta wartość rezystora to od 3k Ω do 100 k Ω . Najtańszym rozwiązaniem jest zastosowanie jako generatora sygnału zegarowego prostego układu RC. Częstotliwość oscylatora jest funkcją napięcia zasilania, dołączonych wartości rezystancji i pojemności oraz temperatury pracy układu. Kolejną sprawą jest różnica w wykonaniu różnych egzemplarzy układu scalonego mikrokontrolera oraz różna tolerancja parametrów elementów pasywnych (RC). Oscylator może pracować bez dołączonego kondensatora zewnętrznego, jednak producent zaleca stosowanie kondensatora o wartości powyżej 20 pF w celu eliminacji wpływu zakłóceń i poprawy stabilności pracy układu. Zbyt mała wartość pojemności może spowodować znaczne zmiany generowanej częstotliwości spowodowane zmianami pojemności zewnętrznych np. na płycie drukowanej lub obudowie mikrokontrolera. Częstotliwość oscylatora podzielona przez 4 jest dostępna na wyprowadzeniu OSC2/CLKOUT mikrokontrolera i może być wykorzystywana do synchronizacji innych układów współpracujących z mikrokontrolerem. Oscylator RC rozpoczyna pracę zaraz po tym, jak wartości napięć osiągną wymagany poziom określony w danych katalogowych. Czas startu oscylatora RC zależy przede wszystkim od:-wartości użytych elementów zewnętrznych RC;-szybkości narastania napięcia zasilającego;-temperatury otoczenia.

29. UKŁAD ZEROWANIA. SCHEMAT LOGICZNY WEWNĘTRZNEGO UKŁADU ZEROWANIA. RODZAJE ZEROWANIA. INICJALIZACJA REJESTRÓW PODCZAS ZEROWANIA

RODZAJE ZEROWANIA:

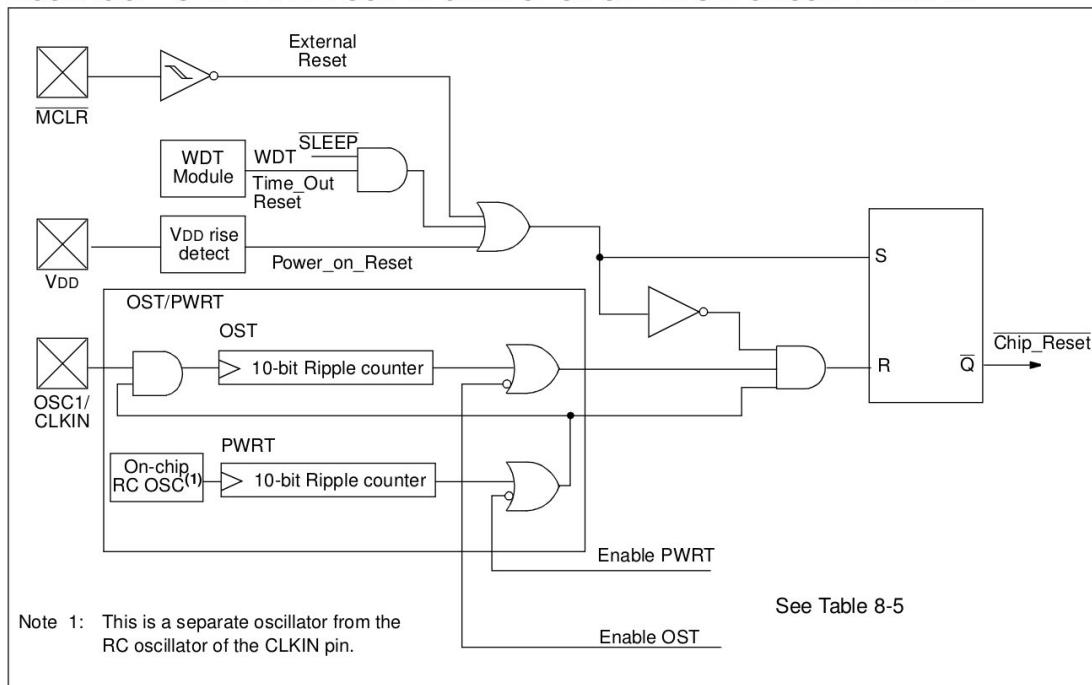
- zerowanie po włączeniu zasilania Power-on Reset (POR);
- podanie niskiego stanu na wejście MCLR mikrokontrolera w trakcie normalnej pracy;
- podanie niskiego stanu na wejście MCLR mikrokontrolera podczas trybu uśpienia;
- zerowanie przez układ licznika nadzorcy (WDT) w takcie normalnej pracy;
- zerowanie przez układ licznika nadzorcy (WDT) podczas trybu uśpienia;
- zerowanie podczas spadku napięcia zasilania poniżej wymaganego poziomu (Brown-out Reset - BOR);
- zerowanie w przypadku błędu parzystości (Parity Error Reset - PER).

Inicjalizacja rejestrów podczas zerowania:

Po włączeniu zasilania zawartość większości rejestrów mikrokontrolera ma wartość nieznaną, a do niektórych z nich jest wpisywane są pewne wartości startowe ustalone przez producenta. Stan większości rejestrów nie ulega zmianie, jeżeli mikrokontroler zostanie wyrowadzony ze stanu uśpienia przez układ licznika nadzorcy WDT. Jeżeli mikrokontroler został wyzerowany, wówczas można określić przyczynę zerowania badając bity !TO i !PD w rejestrze STATUS mikrokontrolera oraz bity POR i BOR z rejestrów PCON.

Schemat logiczny wewnętrznego układu zerowania

FIGURE 8-8: SIMPLIFIED BLOCK DIAGRAM OF ON-CHIP RESET CIRCUIT PIC16F8x



30. Określenie przyczyny restartu mikrokontrolera. Zawartość licznika rozkazów i rejestrów STATUS i PCON po wykonaniu zerowania. Tryb uśpienia.

Przyczynę restartu mikrokontrolera można określić tylko w mikrokontrolerach PIC16F87x, ponieważ zawierają one rejestr specjalny PCON, którego dwa bity $\overline{\text{POR}}$ i $\overline{\text{BOR}}$, w połączeniu z bitami $\overline{\text{TO}}$ i $\overline{\text{PD}}$ rejestru STATUS pozwalają jednoznacznie określić przyczynę restartu mikrokontrolera:

TABLE 14-4: STATUS BITS AND THEIR SIGNIFICANCE

| $\overline{\text{POR}}$ | $\overline{\text{BOR}}$ | $\overline{\text{TO}}$ | $\overline{\text{PD}}$ | Condition |
|-------------------------|-------------------------|------------------------|------------------------|---|
| 0 | x | 1 | 1 | Power-on Reset |
| 0 | x | 0 | x | Illegal, $\overline{\text{TO}}$ is set on $\overline{\text{POR}}$ |
| 0 | x | x | 0 | Illegal, $\overline{\text{PD}}$ is set on $\overline{\text{POR}}$ |
| 1 | 0 | 1 | 1 | Brown-out Reset |
| 1 | 1 | 0 | 1 | WDT Reset |
| 1 | 1 | 0 | 0 | WDT Wake-up |
| 1 | 1 | u | u | MCLR Reset during normal operation |
| 1 | 1 | 1 | 0 | MCLR Reset during Sleep or Interrupt Wake-up from Sleep |

Legend: x = don't care, u = unchanged

Zawartość rejestrów STATUS, PCON oraz licznika rozkazów po wykonaniu różnych typów zerowania:

TABLE 14-5: RESET CONDITIONS FOR SPECIAL REGISTERS

| Condition | Program Counter | Status Register | PCON Register |
|------------------------------------|-----------------------|-----------------|---------------|
| Power-on Reset | 000h | 0001 1xxx | - - - - 0x |
| MCLR Reset during normal operation | 000h | 000u uuuu | - - - - uu |
| MCLR Reset during Sleep | 000h | 0001 0uuu | - - - - uu |
| WDT Reset | 000h | 0000 1uuu | - - - - uu |
| WDT Wake-up | PC + 1 | uuu0 0uuu | - - - - uu |
| Brown-out Reset | 000h | 0001 1uuu | - - - - u0 |
| Interrupt Wake-up from Sleep | PC + 1 ⁽¹⁾ | uuu1 0uuu | - - - - uu |

Legend: u = unchanged, x = unknown, - = unimplemented bit, read as '0'

Note 1: When the wake-up is due to an interrupt and the GIE bit is set, the PC is loaded with the interrupt vector (0004h).

Ponadto jeżeli mikrokontroler jest uśpiony, a zostanie zgłoszone przerwanie, zostaje on wybudzony. Następnie wykonana zostaje pierwsza instrukcja po rozkazie SLEEP, po czym następuje wywołanie (załadowanie PC wartości 0004h, z odłożeniem adresu powrotu na stosie) procedury obsługi przerwania.

Tryb uśpienia

Mikrokontroler wchodzi w tryb uśpienia w momencie wykonania rozkazu SLEEP.

Działania wykonywane w momencie przechodzenia w stan uśpienia:

1. Jeżeli WDT jest aktywny wówczas jego licznik jest zerowany ale nadal pracuje
2. Bit $\overline{\text{TO}}$ jest ustawiany, a $\overline{\text{PD}}$ zerowany
3. Porty I/O zachowują swój stan przed wykonania rozkazu SLEEP

Aby zapewnić minimalny pobór mocy w trybie uśpienia należy zapewnić, że:

- do pinów I/O nie są podłączone żadne układy mogące pobierać dodatkowy prąd
- przetwornik ADC jest wyłączony
- zewnętrzne oscylatory są wyłączone
- porty **wejściowe** są w stanie niskim lub wysokim, a nie pozostawione niepodłączone aby zapobiec występowaniu prądów przełączających spowodowanych niestabilnymi wejściami
- wejście TOCKI powinno być w stanie stabilnym: V_{dd} lub V_{ss}
- rezystory podciągające portu B powinny być odłączone
- Pin $\overline{\text{MCLR}}$ powinien być w wysokim stanie logicznym

Wybudzenie ze stanu uśpienia

Mikrokontroler może zostać wybudzony ze stanu uśpienia z powodu następujących zdarzeń:

1. podanie niskiego stanu logicznego na wejście $\overline{\text{MCLR}}$
2. wybudzenie przez układ timera WDT (jeśli WDT nie jest zablokowany)
3. wystąpienie przerwania:
 1. zewnętrznego (zmiana na wejściu RB0/INT),
 2. spowodowanego zmianą pinów RB0...RB3
 3. od urządzeń peryferyjnych

Podanie niskiego stanu na wejście $\overline{\text{MCLR}}$ powoduje restart mikrokontrolera. Wszystkie pozostałe zdarzenia są traktowane jako kontynuacja wykonania programu. Bitы $\overline{\text{TO}}$ i $\overline{\text{PD}}$ rejestru STATUS mogą zostać użyte do określenia przyczyny resetu mikrokontrolera.

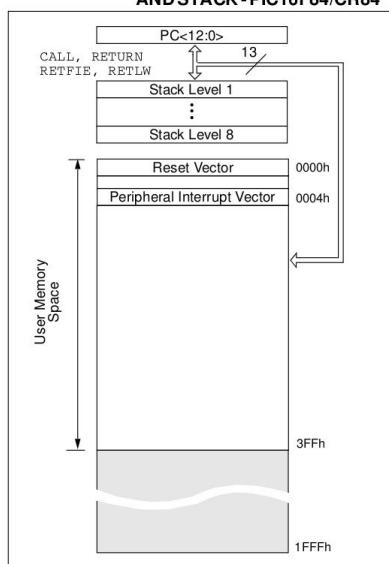
Bit $\overline{\text{PD}}$, ustawiany w momencie włączenia zasilania jest zerowany w momencie wywołania instrukcji SLEEP. Bit $\overline{\text{TO}}$ jest zerowany gdy licznik WDT się przepiąśni i spowoduje restart urządzenia.

Więcej szczegółów: zadanie 16

31. STRUKTURA PAMIĘCI PROGRAMU MIKROKONTROLERÓW PIC16F8X. STRUKTURA PAMIĘCI PROGRAMU MIKROKONTROLERA PIC16F877A

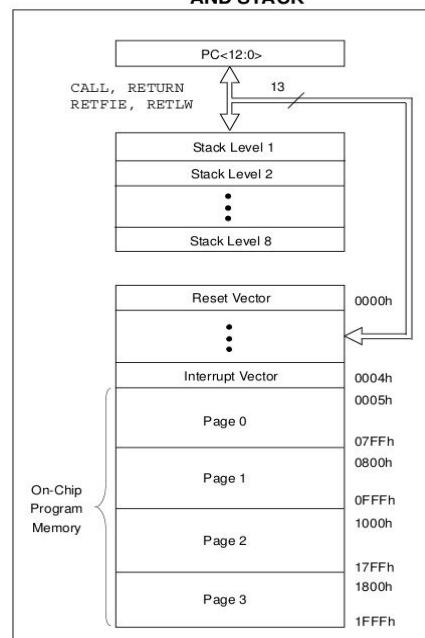
Pamięć programu może mieć różną wielkość w zależności od wersji mikrokontrolera. Jest to ważny parametr, który w dużym stopniu przyczynia się do wyboru odpowiednie wersji układu. Maksymalny rozmiar pamięci programu w mikrokontrolerach PIC16 może wynosić 8K 14-bitowych słów. Adresowalną jednostką w pamięci programu jest 14-bitowe słowo rozkazowe. Rozmiar ten wynika z pojemności licznika rozkazów (PC). Pamięć programu jest podzielona na strony o wielkości 2K słów. Numer strony pamięci programu jest określony przez bity 4 i 3 w rejestrze PCLATH.

FIGURE 4-2: PROGRAM MEMORY MAP AND STACK - PIC16F84/CR84



Rozmiar pamięci: 1K słów 14-bitowych

FIGURE 2-1: PIC16F876A/877A PROGRAM MEMORY MAP AND STACK



Rozmiar pamięci: 8K słów 14-bitowych
(4 strony po 2K słów)

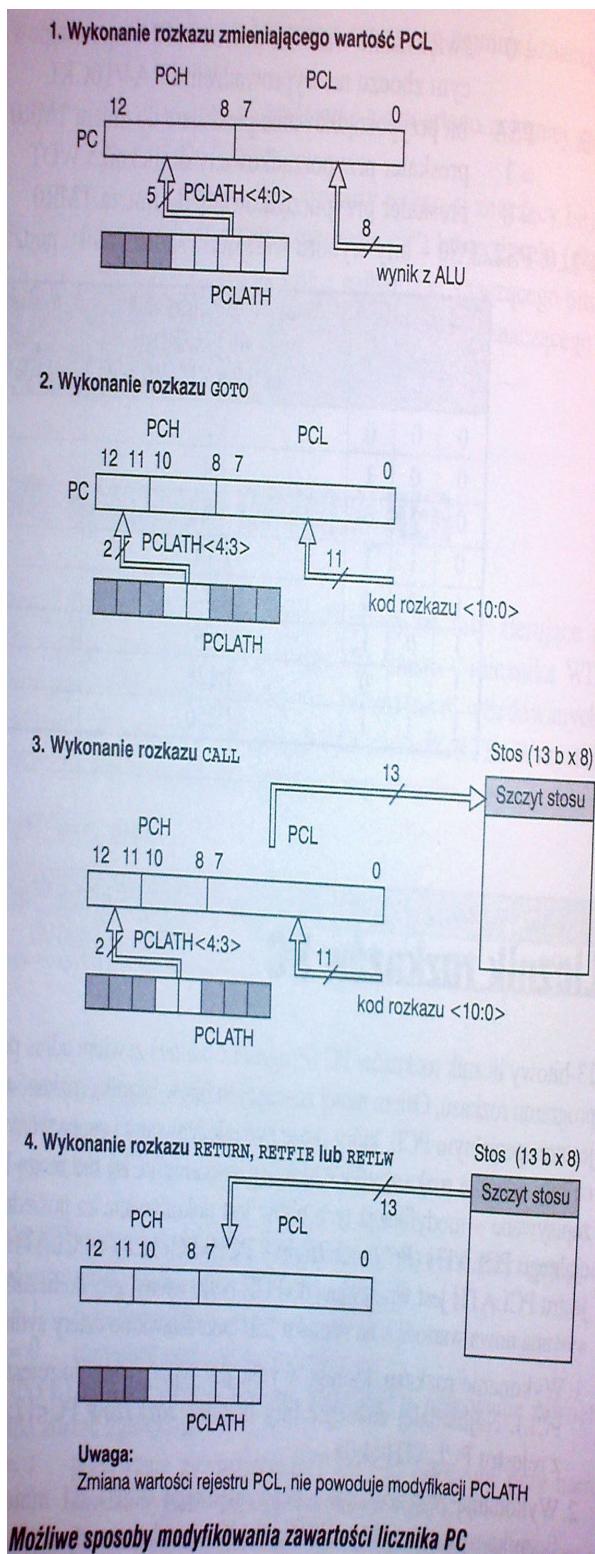
ORGANIZACJA PAMIĘCI PROGRAMU:

W pamięci programu wyróżnione zostały dwa ważne adresy (rysunki). Pierwszy z nich to adres **0000h** nazywany **wektorem restartu** (zerowania) – Reset Vector. Od tego adresu zaczyna się wykonanie programu po zerowaniu mikrokontrolera. Drugi wyróżniony adres to **wektor przerwania 0004h** – (Peripheral) Interrupt Vector, od tego adresu rozpoczyna się program obsługi przerwań.

Struktura pamięci programu mikrokontrolera PIC16F877A

Struktura pamięci programu w mikrokontrolerze PIC16F877A jest zbliżona do ogólnej struktury pamięci programu dla rodziny PIC16. Występuje tu 13-bitowy licznik rozkazów PC oraz 8-poziomowy stos. Wektor zerowania jest równy 0000h, natomiast wektor przerwania 0004h. Rozmiar pamięci programu jest równy 8KB. Pamięć jest podzielona na 4 strony, każda o rozmiarze 2KB.

32. REJESTRY PCL I PCLATH. ZMIANA ZAWARTOŚCI LICZNIKA ROZKAZÓW PC. STOS



Możliwe sposoby modyfikowania zawartości licznika PC

pamięci powinno być poprzedzone modyfikacją bitów PCLATH<4:3>.

STOS: W PIC16 zrealizowano 8-elementowy stos sprzętowy jako bufor ośmiu 13-bitowych rejestrów. Na stosie przechowywane są 13-bitowe adresy powrotu (zawartość licznika rozkazów PC) z podprogramów. Stos mikrokontrolerów PIC16 wykorzystuje się wyłącznie do przechowywania adresów powrotu z procedur i programu obsługi przerwań.

Pamięć programu jest adresowana z pomocą 13-bitowego licznika rozkazów PC (Program Counter), który przechowuje adres rozkazu pobieranego z pamięci. Osiem mniejszych znaczących bitów licznika rozkazów PC<7:0> jest zawarta w rejestrze specjalnym PCL, natomiast starsze bity PC<12:8> są określone zawartością bitów<4:0> rejestrów specjalnych PCLATH (Program Counter Latch Register).

FORMAT LICZNIKA ROZKAZÓW PC:

Cechy:

- wykonanie operacji arytmetycznych i logicznych, czyli może on być **operandem docelowym** instrukcji, w odróżnieniu od większości innych architektur, gdzie licznik PC może być modyfikowany tylko za pomocą instrukcji skoków oraz wywołania podprogramów.
- 13-bit licznik instr. PC pozwala zaadresować max do 8K słów rozkazowych
- zawijania adresów (starsze niewykorzystane bity - ignorowane).

Pamięć programu mikrokontrolerów PIC16 dzieli się na strony rozmiarem 2K słów. Numer strony jest określany przez bitów PCLATH<4:3>. Związanego z tym jest wykonanie rozkazu skoku GOTO oraz rozkazu wywołania podprogramu CALL.

Przekazanie sterowania do innej strony

pamięci powinno być poprzedzone modyfikacją bitów PCLATH<4:3>.

Stos zorganizowany jest jako **bufor cykliczny**, dlatego poziom zagnieżdżenia podprogramów jest ograniczony do ośmiu. Odłożenie na stosie więcej niż ośmiu adresów spowoduje nadpisanie przechowywanych danych i doprowadzi do naruszenia prawidłowego wykonania programu. **Przepełnienie stosu** nie jest w żaden sposób sygnalizowane, o to aby tego nie dopuścić powinien zadbać programista. Wskaźnik stosu także nie jest programowo dostępny.

33. STRUKTURA PAMIĘCI DANYCH MIKROKONTROLERÓW PIC16F8X. REJESTRY SPECJALNE I UNIWERSALNE

Pamięć danych mikrokontrolerów PIC16 ma organizację 8-bitową. Instrukcje dostępu do pamięci umożliwiają odczyt i zapis poszczególnych bajtów pamięci. Pamięć danych ma organizację bankową, dzieli się na banki o rozmiarze 128 B każdy. Maksymalnie układ mikrokontrolera może posiadać do czterech banków pamięci. Rozmiar pamięci danych i liczba zaimplementowanych banków zależy od wersji mikrokontrolera.

W pamięci danych umieszczone są rejesty dwóch rodzajów:

- -rejestry specjalne SFR (Special Function Register);
- -rejestry uniwersalne GPR (General Purpose Registers). Rejestry uniwersalne GPR służą do przechowywania danych, operandów instrukcji, a rejesty SFR – do komunikacji z blokami peryferyjnymi mikrokontrolera.

Rejestry SFR zajmują obszary o niskich adresach w poszczególnych bankach pamięci, powyżej obszaru SFR umieszczone są rejesty GPR. Umieszczenie rejestrów GPR i SFR w jednym obszarze adresowania pozwala na ujednolicenie sposobów dostępu tych funkcjonalnie różnych rejestrów.

Najważniejsze, najczęściej wykorzystywane rejesty SFR powielane są w kilku bankach pamięci w celu zmniejszenia liczby przełączeń banków, przyśpieszenia dostępu tych rejestrów, usprawnienia sterowania blokami peryferyjnymi.

Pole adresu zawarte w kodzie instrukcji jest 7-bitowe i pozwala zaadresować do 128 B pamięci. Jeśli mikrokontroler posiada pamięć danych większą niż 128B, wykorzystuje się mechanizm podziału pamięci na banki rozmiarem 128B. Bity RP1:RP0 w rejestrze STATUS przechowują numer aktywnego banku pamięci (0-3), 7-bitowy adres zawarty w kodzie instrukcji adresuje rejestr z aktywnego banku pamięci.

STRUKTURA PAMIĘCI DANYCH MIKROKONTROLERÓW PIC16F8X

FIGURE 4-1: REGISTER FILE MAP -
PIC16F83/CR83

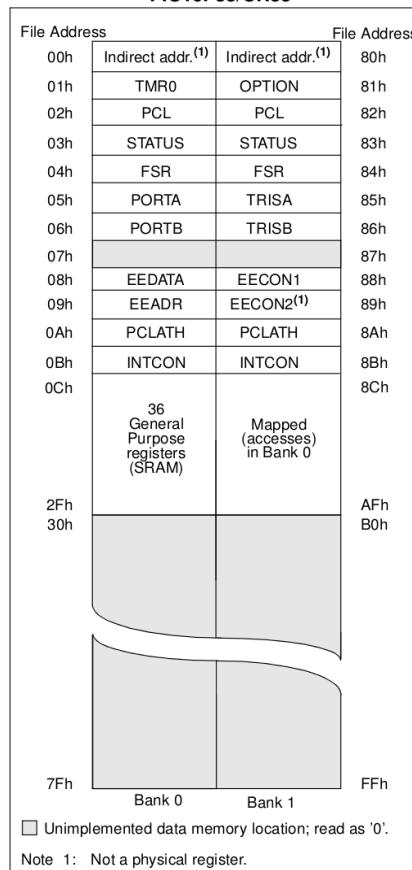
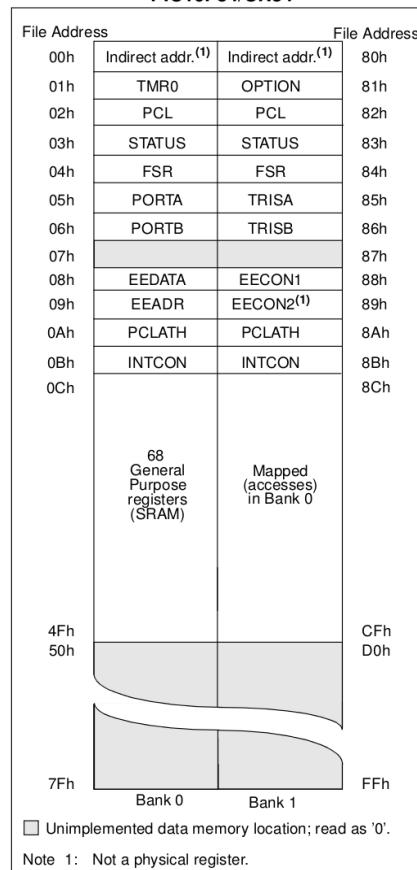


FIGURE 4-2: REGISTER FILE MAP -
PIC16F84/CR84



34. BEZPOŚREDNI I POŚREDNI DOSTĘP DO PAMIĘCI. ADRESOWANIE BEZPOŚREDNIE. PRZYKŁAD ADRESOWANIA BEZPOŚREDNIEGO. ADRESOWANIE POŚREDNIE. PRZYKŁAD ADRESOWANIA POŚREDNIEGO

Pamięć adresowana może być na dwa sposoby: pośredni i bezpośredni.

W trybie adresowania **bezpośredniego** adres rejestru pamięci danych zawarty jest w kodzie rozkazu. Jest to **7-bitowy adres**, który wskazuje słowo w aktywnym banku pamięci, a wybór odpowiedniego banku dokonywany jest za pomocą bitów RP1, RP0 z rejestru STATUS.

Efektywny 9-bitowy adres pamięci tworzy się poprzez połączenie 2-bitowego pola RP1:RP0 (starsze bity adresu) oraz 7-bitowego adresu zawartego w kodzie rozkazu. Za adresowanie rejestrów z obszaru pamięci, który nie został zaimplementowany, powoduje odwołanie się do odpowiednich komórek pamięci w młodszych banków (zawijanie adresów polegające na ignorowaniu najbardziej znaczących bitów adresu).

Przykład adresowania bezpośredniego:

```
; program wpisuje wartość 0 do rejestrów PORTA i TRISA  
MOVLW 0          ; do rejestru W ładuj 0  
BCF   STATUS, RP0 ; przełącz się do banku 0  
BCF   STATUS, RP1 ;  
MOVWF PORTA       ; przepisz zawartość W do rejestru  
                  ; PORTA (bank 0)  
BSF   STATUS, RP0 ; przełącz się do banku 1  
MOVWF TRISA       ; przepisz zawartość W do rejestru  
                  ; TRISA (bank 1)
```

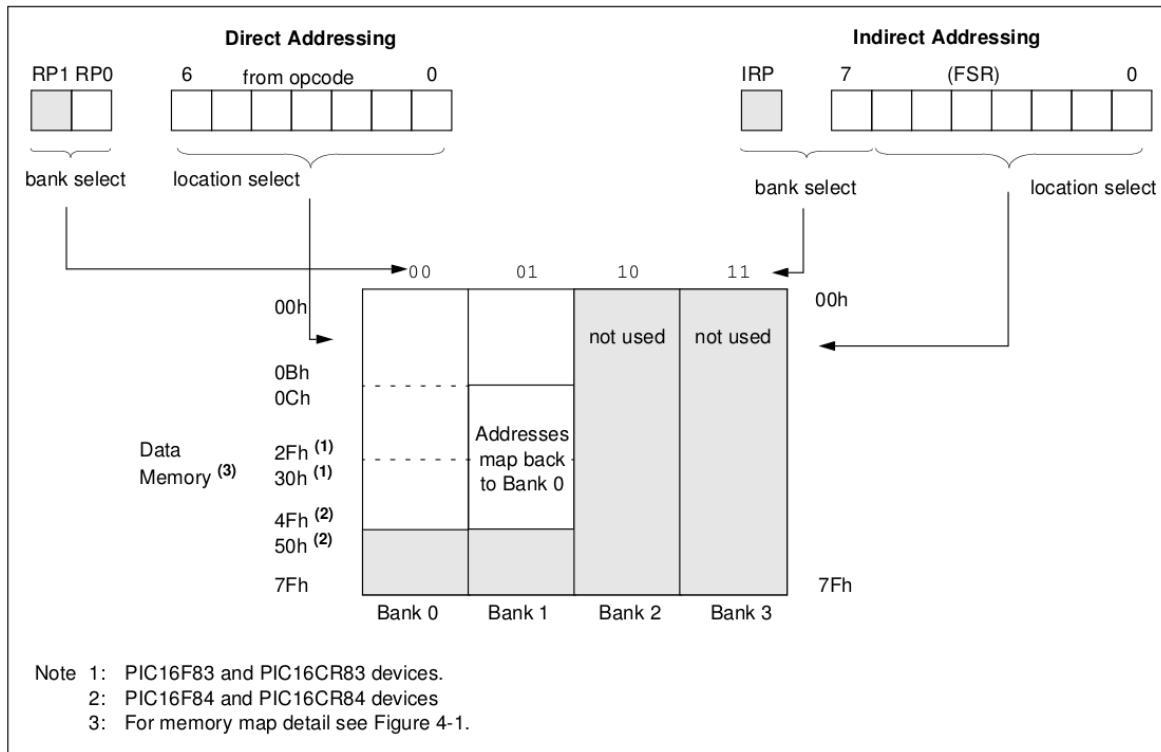
ADRESOWANIE POŚREDNIE. PRZYKŁAD ADRESOWANIA POŚREDNIEGO

W trybie adresowania pośredniego **osiem mniejszych znaczących bitów** adresu pamięci danych znajduje się w rejestrze specjalnym FSR (File Select Register), **starszy bit adresu** stanowi bit IRP z rejestru STATUS.

Tryb adresowania pośredniego uzyskuje się przy odwołaniu do rejestru INDF o adresie 00h. Użycie rejestru INDF jako operandu powoduje, że właściwy adres argumentu pobierany jest z rejestru FSR i rozszerzany o wartość bitu IRP (starszy bit). Tryb adresowania pośredniego wymaga uprzedniego załadowania adresu komórki pamięci do FSR i odpowiedniego ustawienia bitu wyboru banków pamięci IRP.

```
; program zeruje 16 komórek pamięci od adresu 0x20  
MOVLW 0x20      ; zainicjuj wskaźnik do pamięci  
MOVWF FSR  
NEXT:  
CLRF  INDF      ; wyzeruj rejestr INDF  
INCF  FSR,F     ; inkrementuj wskaźnik do pamięci  
BTFS  FSR,4     ; sprawdź czy wskaźnik równy 16  
GOTO  NEXT      ; nie - skocz do NEXT  
CONTINUE:      ; tak - idź dalej  
    . . .
```

FIGURE 4-1: DIRECT/INDIRECT ADDRESSING



Ilustracja 1: Bezpośrednie/pośrednie adresowanie pamięci danych dla PIC16F8x

35. STRUKTURA PAMIĘCI DANYCH MIKROKONTROLERA PIC16F877A I PIC16F84

Pamięć danych w mikrokontrolerze PIC16F877A jest podzielona na 4 banki, które zawierają rejesty ogólne przeznaczenia (GPR – General Purpose Registers) oraz specjalne rejesty funkcyjne (SFR – Special Function Registers).

Wybór banku pamięci:

Każdy bank pamięci zawiera 128 bajtów danych. Pod najniższymi adresami w każdym banku są umieszczone rejesty specjalne SFR. Pod wyższymi adresami są rejesty ogólnego przeznaczenia GPR, zaimplementowane jako pamięć statyczna RAM.

Wszystkie banki zawierają rejesty specjalne. Niektóre z rejestrów specjalnych są powielone w innych bankach, co umożliwia redukcję kodu i szybszy dostęp do rejestrów (nie trzeba przełączać się między bankami).

Wybór aktywnego banku pamięci odbywa się za pomocą bitów RP1:RP0 rejestru STATUS:

| RP1:RP0 | Bank |
|---------|------|
| 00 | 0 |
| 01 | 1 |
| 10 | 2 |
| 11 | 3 |

Każdy rejestr ogólnego przeznaczenia jest 8-bitowy i może być dostępny bezpośrednio lub pośrednio za pomocą rejestrów FSR.

Struktura pamięci danych PIC16F84: patrz zadanie 33

36. REJESTRY SPECJALNE SFR MIKROKONTROLERA PIC16F877A oraz PIC16F84

Specjalne rejesty funkcyjne (SFR) są używane przez jednostkę centralną mikrokontrolera oraz układy peryferyjne do sterowania funkcjami mikrokontrolera. Rejestry SFR są zaimplementowane jako pamięć statyczna RAM. Rejestry specjalne mikrokontrolera PIC16F877A

FIGURE 2-4: PIC16F873A/874A REGISTER FILE MAP

■ Unimplemented data memory locations, read as '0'

- * Not a physical register

Note 1: These registers are not implemented on the PIC16F873A.

- 1: These registers are not implemented on the PIC16F8/3A.
- 2: These registers are reserved; maintain these registers clear.

Rejestry SRF PIC16F84: patrz zadanie 33

| Nazwa | Bank | Adresy | opis |
|--------|---------|-----------------------|--------------------------------|
| INDF | 0,1,2,3 | 00h,80h,100h,180h | adresowania pośredniego |
| TMR0 | 0,2 | 01h,101h | Układ czasowy timer0 |
| PCL | 0,1,2,3 | 02h, 82h, 102h,182h. | Liczniok rozkazowy(m. bity) |
| STATUS | 0,1,2,3 | 03h, 83h 103h 183h | Rejestr stanu |
| FSR | 0,1,2,3 | 04h, 84h, 104h, 184h, | Do adresowania pośredniego |
| PORTA | 0 | 05h | Port we/wy A |
| PORTB | 0,2 | 06h, 106h | Port we/wy B |
| PORTC | 0 | 07h | Port we/wy C |
| PORTD | 0 | 08h | Port we/wy D |
| PORTE | 0 | 09h | Port we/we C |
| PCLATH | 0,1,2,3 | 0 Ah,8Ah,10Ah,18Ah, | Liczniok rozkazów (s. bity) |
| INTCON | 0,1,2,3 | 0Bh 8 Bh 10 Bh 18 Bh | Rejestr sterujacy przerwaniami |

37. REJESTR STATUS. STRUKTURA REJESTRU STATUS. BITY REJESTRU STATUS

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|------------|------------|------------|------------|------------|----------|-----------|----------|
| IRP | RP1 | RP0 | !TO | !PD | Z | DC | C |
| R/W – 0 | R/W – 0 | R/W – 0 | R – 1 | R – 1 | R/W – x | R/W – x | R/W – x |

Legenda: R – bit może być odczytywany, W – zapisywany, – n – wartość po włączeniu zasilania (POR)

Rejestr STATUS zawiera różne ważne znaczniki systemowe np. określające wynik wykonanej przez jednostkę ALU operacji.

- bit 0 - **C** – znacznik przeniesienia, zostanie ustawiony ($C=1$), jeśli przy wykonaniu operacji wystąpi przeniesienie ze starszego bitu wyniku;
- bit 1 - **DC** – znacznik przeniesienia pomocniczego, zostanie ustawiony ($DC=1$), jeśli przy wykonaniu operacji wystąpi przeniesienie z młodszej tetryady wyniku (bitu 3);
- bit 2 - **Z** – znacznik wyniku zerowego, zostanie ustawiony ($Z=1$), jeśli wynik operacji jest równy zero;
- bit 3 - **!PD** – znacznik uśpienia mikrokontrolera, zostanie wyzerowany ($PD=0$) po wprowadzeniu mikrokontroler instrukcją SLEEP w tryb uśpienia;
- bit 4 - **!TO** – znacznik przepełnienia licznika nadzorcy WDT, zostanie wyzerowany ($TO=0$) po przepełnieniu licznika WDT;
- bity 6 i 5 - **RP1:RP0** – bity rozszerzenia adresu przy adresowaniu bezpośrednim pamięci danych. Bity te stanowią starszą część 9-bitowego adresu pamięci danych, wartość zapisana w tych bitach określa numer banku pamięci (0 - 3), do którego wykonuje się odwołanie. Młodsza 7-bitowa część adresu pobierana jest z kodu instrukcji i adresuje rejestr wybranego banku pamięci;
- bit 7 - **IRP** – starszy 9-ty bit adresu przy adresowaniu pośrednim pamięci danych. Młodsze 8 bitów adresu przechowują się w rejestrze FSR.

Rejestr stanu STATUS może występować jako **operand instrukcji** oraz **rejestr docelowy** wyniku. W takim przypadku niemożliwa jest zmiana stanu bitów !PD i !TO, a bity znaczników C, DC oraz Z zostaną zmodyfikowane przez jednostkę ALU zgodnie z tym, czy powstały odpowiednie przeniesienia i czy otrzymany został wynik zerowy.

Do programowej modyfikacji zawartości rejestrystu STATUS należy używać instrukcji, które nie zmieniają stanu znaczników (bcf, bsf, swapf, movwf).

Odczyt i modyfikacja:

```
MOVF STATUS,W ; przenieś zawartość rejestrystu ; STATUS do W  
BCF STATUS, RP0 ; wyzeruj bit RP0 rejestrystu STATUS  
BSF STATUS, RP1 ; ustaw bit RP1 rejestrystu STATUS
```

38. Porty wejścia/wyjścia mikrokontrolerów PIC16F8x. Porty wejścia/wyjścia mikrokontrolera PIC16F877A. Rejestry PORT_x i TRIS_x. Inicjalizacja portów

Porty A, B, C, D i E mikrokontrolerów są portami wejścia/wyjścia ogólnego przeznaczenia. Wyprowadzenia portów mogą również pełnić wiele funkcji dodatkowych, takich jak np.:

- wejście układu czasowo-licznikowego;
- wejście sygnału przerwania;
- wejście przetwornika analogowo-cyfrowego;
- wejścia i wyjścia interfejsu szeregowego i równoległego,
- magistrali I2C;
- wyprowadzenia układu CCP, itp.

Mikrokontroler PIC16F877A posiada 5 portów ogólnego przeznaczania: A, B, C, D i E. Porty te mogą pełnić następujące funkcje:

- port A:
 - wejście/wyjście cyfrowe,
 - wejście analogowe,
 - wejście napięcia odniesienia,
 - wejście zewnętrznego zegara dla układu TIMER0,
 - wyjście komparatora,
 - wejście wyboru urządzenia podległego dla portu SSP;
- port B:
 - wejście/wyjście cyfrowe,
 - wejście do programowania mikrokontrolera,
 - wejście przerwania zewnętrznego;
- port C:
 - wejście/wyjście cyfrowe,
 - wejście/wyjście modułów CCP,
 - wejście/wyjście dla różnego typu magistral szeregowych (SPI, I2C),
 - wejście/wyjście dla kontrolera transmisji szeregowej USART,
 - wyjście oscylatora i wejście zegarowe dla układu TIMER1;
- port D:
 - wejście/wyjście cyfrowe,
 - wejście/wyjście portu równoległego PSP;
- port E:
 - wejścia sterujące portem równoległym PSP,
 - wejście analogowe.

Wybór kierunku pracy

Z każdym portem wejścia/wyjścia są związane dwa specjalne rejesty: PORTx i TRISx, gdzie 'x' odpowiada nazwie portu określonej kolejną literą alfabetu (A, B, C, D, E). Rejestry PORTx stanowią interfejs pomiędzy jednostką centralną mikrokontrolera a portami.

Za pomocą tych rejestrów można wysyłać dane na zewnątrz mikrokontrolera, wpisując odpowiednią wartość do odpowiedniego rejestru PORTx. Rejestry PORTx służą też do przekazywania jednostce centralnej wartości wejściowych przychodzących do portów mikrokontrolera, poprzez odczyt odpowiednich rejestrów PORTx.

Rejestry TRISx służą do konfiguracji portów wejścia/wyjścia poprzez określenie kierunku pracy danego portu (wejście lub wyjście). Ustalanie kierunku pracy portu odbywa się poprzez ustawienie odpowiednich bitów w rejestrze TRISx. Ustawienie jedynki w rejestrze TRISx powoduje przełączenie odpowiedniego wyrowadzenia portu PORTx na tryb wejścia, natomiast wpisanie zera – przełączenie na wyjście. W celu łatwiejszego zapamiętania wystarczy zauważyc podobieństwo pomiędzy znakami 1 – I (czyli Input – wejście) oraz 0 – O (czyli Output – wyjście).

Iinicjalizacja portów

Wszystkie porty wejścia/wyjścia należy przed użyciem zainicjalizować. Polega to głównie na wyborze kierunku przesyłania danych za pomocą rejestrów TRISx. Iinicjalizację portów można przeprowadzić wykonując następujące czynności:

1. wybrać bank w którym znajdują się rejesty PORTx (bank nr 0);
2. wyzerować rejesty zatrzaskowe portu poprzez wpisanie 0 do rejestru PORTx;
3. wybrać bank w którym znajdują się rejesty TRISx (bank nr 1);
4. wybrać kierunek przepływu danych portu x, poprzez ustawienie poszczególnych bitów rejestrów TRISx.

W przypadku mikrokontrolera PIC16F877A, **port A** po włączeniu zasilania jest ustawiany w **tryb wejścia analogowego**. Powoduje to, że przy odczycie tego portu zwracana jest wartość 0. Aby można było używać portu A jako wejścia/wyjścia cyfrowego należy ustawić **odpowiednia wartość (06h) w rejestrze ADCON1**.

39. Architektura wyprowadzeń portu A. Funkcje wyprowadzeń portu A. Schemat blokowy wyprowadzeń RA3:RA0 portu A. Wyprowadzenie RA4. Schemat blokowy wyprowadzenia RA5 portu A dla mikrokontrolera PIC16F877A.

Port A jest dwukierunkowym portem ogólnego przeznaczenia mikrokontrolerów PIC16. Dla mikrokontrolera PIC16F877A wyprowadzenia portu A są współdzielone z układami CCP oraz przetwornikiem analogowo-cyfrowym.

Dla mikrokontrolera PIC16F877A szerokość portu to 6 bitów, a dla mikrokontrolera PIC16F84 – 5 bitów. Podczas czytania danych z rejestru PORTA sprawdzane są aktualne stany wyprowadzeń portu A, natomiast podczas zapisywania do rejestru PORTA – dane są zapamiętywane w zatrzaskach portu. Podczas wszystkich operacji zapisu do rejestru wykonywana jest sekwencja kroków: czytaj, modyfikuj, zapisz. Czyli zapis do portu oznacza, że aktualny stan wyprowadzeń zostanie najpierw przeczytany, następnie zmodyfikowany przez CPU, dopiero na końcu nowa wartość będzie zapisana do zatrzasków portu.

PIC16F8x

| Name | Bit0 | Buffer Type | Function |
|-----------|------|-------------|--|
| RA0 | bit0 | TTL | Input/output |
| RA1 | bit1 | TTL | Input/output |
| RA2 | bit2 | TTL | Input/output |
| RA3 | bit3 | TTL | Input/output |
| RA4/T0CKI | bit4 | ST | Input/output or external clock input for TMR0. Output is open drain type. |

Legend: TTL = TTL input, ST = Schmitt Trigger input

TABLE 5-2 SUMMARY OF REGISTERS ASSOCIATED WITH PORTA

| Address | Name | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 | Value on Power-on Reset | Value on all other resets |
|---------|-------|-------|-------|-------|-----------|--------|--------|--------|--------|-------------------------|---------------------------|
| 05h | PORTA | — | — | — | RA4/T0CKI | RA3 | RA2 | RA1 | RA0 | ---x xxxx | ---u uuuu |
| 85h | TRISA | — | — | — | TRISA4 | TRISA3 | TRISA2 | TRISA1 | TRISA0 | ---1 1111 | ---1 1111 |

Legend: x = unknown, u = unchanged, - = unimplemented read as '0'. Shaded cells are unimplemented, read as '0'

PIC16F87xA

TABLE 4-1: PORTA FUNCTIONS

| Name | Bit# | Buffer | Function |
|---------------------|-------|--------|--|
| RA0/AN0 | bit 0 | TTL | Input/output or analog input. |
| RA1/AN1 | bit 1 | TTL | Input/output or analog input. |
| RA2/AN2/VREF-/CVREF | bit 2 | TTL | Input/output or analog input or VREF- or CVREF. |
| RA3/AN3/VREF+ | bit 3 | TTL | Input/output or analog input or VREF+. |
| RA4/T0CKI/C1OUT | bit 4 | ST | Input/output or external clock input for Timer0 or comparator output. Output is open-drain type. |
| RA5/AN4/SS/C2OUT | bit 5 | TTL | Input/output or analog input or slave select input for synchronous serial port or comparator output. |

Legend: TTL = TTL input, ST = Schmitt Trigger input

TABLE 4-2: SUMMARY OF REGISTERS ASSOCIATED WITH PORTA

| Address | Name | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 | Value on: POR, BOR | Value on all other Resets |
|---------|--------|-------|-------|-------------------------------|-------|-------|-------|-------|-------|--------------------|---------------------------|
| 05h | PORTA | — | — | RA5 | RA4 | RA3 | RA2 | RA1 | RA0 | --0x 0000 | --0u 0000 |
| 85h | TRISA | — | — | PORTA Data Direction Register | | | | | | --11 1111 | --11 1111 |
| 9Ch | CMCON | C2OUT | C1OUT | C2INV | C1INV | CIS | CM2 | CM1 | CM0 | 0000 0111 | 0000 0111 |
| 9Dh | CVRCON | CVREN | CVROE | CVRR | — | CVR3 | CVR2 | CVR1 | CVR0 | 000- 0000 | 000- 0000 |
| 9Fh | ADCON1 | ADFM | ADCS2 | — | — | PCFG3 | PCFG2 | PCFG1 | PCFG0 | 00-- 0000 | 00-- 0000 |

Legend: x = unknown, u = unchanged, - = unimplemented locations read as '0'. Shaded cells are not used by PORTA.

Note: When using the SSP module in SPI Slave mode and SS enabled, the A/D converter must be set to one of the following modes, where PCFG3:PCFG0 = 0100, 0101, 011x, 1101, 1110, 1111.

SCHEMAT BLOKOWY WYPROWADZEŃ RA3:RA0 PORTU A. WYPROWADZENIE RA4. SCHEMAT BLOKOWY WYPROWADZENIA RA5 PORTU A DLA MIKROKONTROLERA PIC16F877A.

Wyprowadzenie RA4 jest współdzielone z wejściem zegara modułu TIMER0. Wejście RA4/T0CKI posiada na wejściu przerzutnik Schmitta oraz wyjście z otwartym drenem.

Przerzutnik Schmitta ma dwa progi przełączania, przy których wyjście zmienia stan na przeciwny. Osiągnięcie przez napięcie wejściowe danego progu jest zależne od kierunku zmiany napięcia wejściowego. Dla napięcia narastającego właściwym jest próg górny, dla opadającego - dolny. Odległość między progami nazywana jest szerokością pętli histerezy.

Iinicjalizacja portu A dla mikrokontrolera PIC16F877A

Dla mikrokontrolera PIC16F877A wyprowadzenia portu A są współdzielone z wejściami analogowymi przetwornika A/C. Podczas **zerowania** mikrokontrolera port A jest ustawiany w **trybie wejścia analogowego** i każdy odczyt z tego portu zwróci wartość równą zero.

Dlatego należy pamiętać, aby w przypadku wykorzystywania portu A jako portu cyfrowego przełączyć go w tryb cyfrowy wpisując do rejestru ADCON1 wartość 06h.

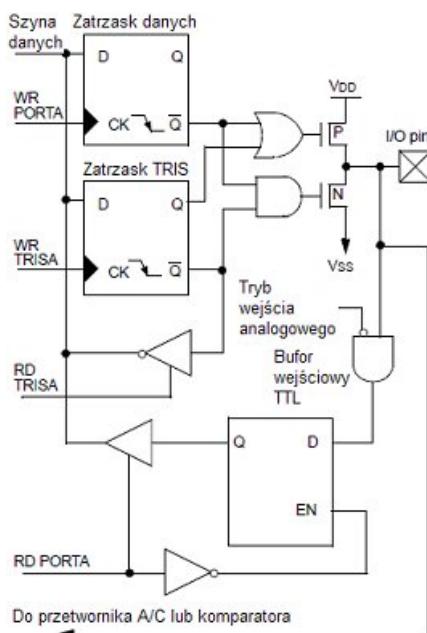
Iinicjalizację portu A pokazano na przykładzie:

```

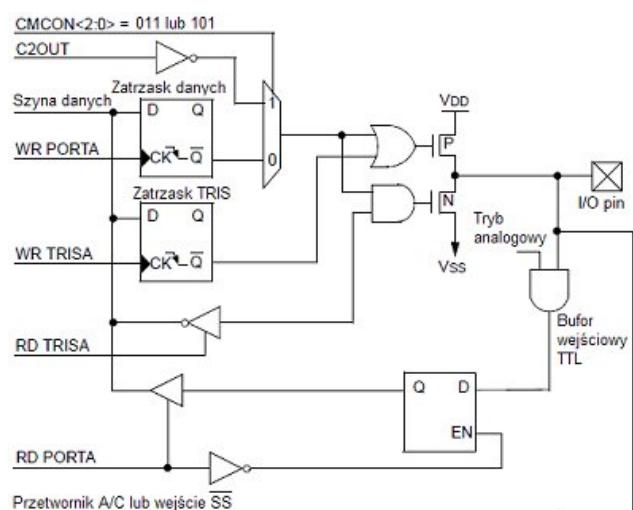
bcf      STATUS, RP0      ; wybór banku 0
clrf    PORTA             ; inicjalizacja PORTA przez zerowanie zatrzasków wyjściowych
bsf      STATUS, RP0      ; wybór banku 1
movlw   B'00000110'        ; przełączenie wejść na cyfrowe
movwf   ADCON1            ; poprzez odłączenie przetwornika A/C
clrf    TRISA              ; ustawienie wyprowadzeń PORTA na wyjścia

```

Schemat portów RA0:RA3



Schemat portu RA4



40. Architektura wyprowadzeń portu B. Inicjalizacja portu B. Funkcje wyprowadzeń portu B. Schemat blokowy wyprowadzeń RB3:RB0 i RB7:RB4 portu B dla mikrokontrolera PIC16F877A.

Port B jest dwukierunkowym 8-bitowym portem ogólnego przeznaczenia mikrokontrolerów PIC16.

Dla mikrokontrolera PIC16F877A wyprowadzenie portu B są współdzielone z wejściami programatora niskonapięciowego i debugera sprzętowego.

Każde wyprowadzenie portu B ma możliwość **podłączenia rezystora podciągającego (pull-up)**. Opcja ta jest aktywowana poprzez wyzerowanie bitu RBPU w rejestrze OPTION_REG. Rezystory podciągające są automatycznie wyłączone, gdy wyprowadzenia są skonfigurowane jako wyjścia. Funkcja ta jest też wyłączana w momencie zerowania mikrokontrolera.

Inicjalizację portu B pokazano na przykładzie:

```
bcf STATUS, RP0      ; wybór banku 0
clrf PORTB          ; inicjalizacja przez zerowanie zatrzasków wyjściowych
bsf STATUS, RP0      ; wybór banku 1
bcf OPTION_REG, RBPU ; włączenie rezystorów pull-up
movlw B'00001111'    ; ustawienie wyprowadzeń RB3:RB0 jako wejście oraz
movwf TRISB          ; wyprowadzeń RB4:RB7 ; jako wyjście
```

Funkcje poszczególnych pinów portu B

PIC16F8x

| Name | Bit | Buffer Type | I/O Consistency Function |
|---------|------|-----------------------|---|
| RB0/INT | bit0 | TTL/ST ⁽¹⁾ | Input/output pin or external interrupt input. Internal software programmable weak pull-up. |
| RB1 | bit1 | TTL | Input/output pin. Internal software programmable weak pull-up. |
| RB2 | bit2 | TTL | Input/output pin. Internal software programmable weak pull-up. |
| RB3 | bit3 | TTL | Input/output pin. Internal software programmable weak pull-up. |
| RB4 | bit4 | TTL | Input/output pin (with interrupt on change). Internal software programmable weak pull-up. |
| RB5 | bit5 | TTL | Input/output pin (with interrupt on change). Internal software programmable weak pull-up. |
| RB6 | bit6 | TTL/ST ⁽²⁾ | Input/output pin (with interrupt on change). Internal software programmable weak pull-up. Serial programming clock. |
| RB7 | bit7 | TTL/ST ⁽²⁾ | Input/output pin (with interrupt on change). Internal software programmable weak pull-up. Serial programming data. |

Legend: TTL = TTL input, ST = Schmitt Trigger.

Note 1: This buffer is a Schmitt Trigger input when configured as the external interrupt.

2: This buffer is a Schmitt Trigger input when used in serial programming mode.

PIC16F877A

| Name | Bit# | Buffer | Function |
|------------------------|-------|-----------------------|--|
| RB0/INT | bit 0 | TTL/ST ⁽¹⁾ | Input/output pin or external interrupt input. Internal software programmable weak pull-up. |
| RB1 | bit 1 | TTL | Input/output pin. Internal software programmable weak pull-up. |
| RB2 | bit 2 | TTL | Input/output pin. Internal software programmable weak pull-up. |
| RB3/PGM ⁽³⁾ | bit 3 | TTL | Input/output pin or programming pin in LVP mode. Internal software programmable weak pull-up. |
| RB4 | bit 4 | TTL | Input/output pin (with interrupt-on-change). Internal software programmable weak pull-up. |
| RB5 | bit 5 | TTL | Input/output pin (with interrupt-on-change). Internal software programmable weak pull-up. |
| RB6/PGC | bit 6 | TTL/ST ⁽²⁾ | Input/output pin (with interrupt-on-change) or in-circuit debugger pin. Internal software programmable weak pull-up. Serial programming clock. |
| RB7/PGD | bit 7 | TTL/ST ⁽²⁾ | Input/output pin (with interrupt-on-change) or in-circuit debugger pin. Internal software programmable weak pull-up. Serial programming data. |

Legend: TTL = TTL input, ST = Schmitt Trigger input

Note 1: This buffer is a Schmitt Trigger input when configured as the external interrupt.

2: This buffer is a Schmitt Trigger input when used in Serial Programming mode or in-circuit debugger.

3: Low-Voltage ICSP Programming (LVP) is enabled by default which disables the RB3 I/O function. LVP must be disabled to enable RB3 as an I/O pin and allow maximum compatibility to the other 28-pin and 40-pin mid-range devices.

SCHEMAT BLOKOWY WYPROWADZEŃ RB3:RB0 I RB7:RB4 PORTU B DLA MIKROKONTROLERA PIC16F877A

Port B posiada także **wejście przerwania zewnętrznego** (bit nr 0) oznaczone jako RB0/INT. Wszystkie wyprowadzenia posiadają bufor TTL na wejściu. Linia 0 portu posiada na wejściu przerzutnik Schmitta, kiedy jest wykorzystywana jako wejście przerwania zewnętrznego.

Linie 6 i 7 (dla mikrokontrolera PIC16F877A) posiadają na wejściu przerzutnik Schmitta, gdy są wykorzystywane do **programowania mikrokontrolera**.

Cztery wyprowadzenia portu B (RB7:RB4) mają funkcję **generacji przerwania** w momencie zmiany stanu. Jest to możliwe tylko w przypadku konfiguracji tych wyprowadzeń jako wejścia. Stan wejść jest wówczas porównywany z zawartością zatrzasków pamiętających poprzednią wartość na wyprowadzeniu. „Różniące się” wyjścia są połączone za pomocą bramki OR i generują przerwanie sygnalizowane bitem RBIF w rejestrze INTCON. Przerwanie to może także wybudzić mikrokontroler ze stanu uśpienia.

Należy pamiętać o wyzerowaniu znacznika RBIF w procedurze obsługi przerwania. Przed wyzerowaniem RBIF należy jednak odczytać rejestr PORTB, gdyż dopiero wówczas znikną różnice pomiędzy stanami na wyprowadzeniach RB7:RB4 i wartościami zapamiętanymi w zatrzaskach.

FIGURE 4-4: BLOCK DIAGRAM OF RB3:RB0 PINS

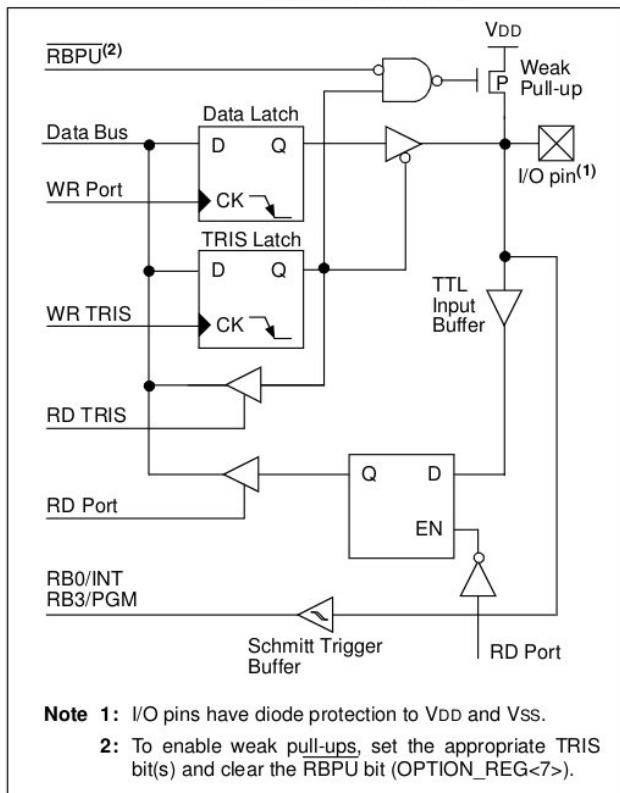
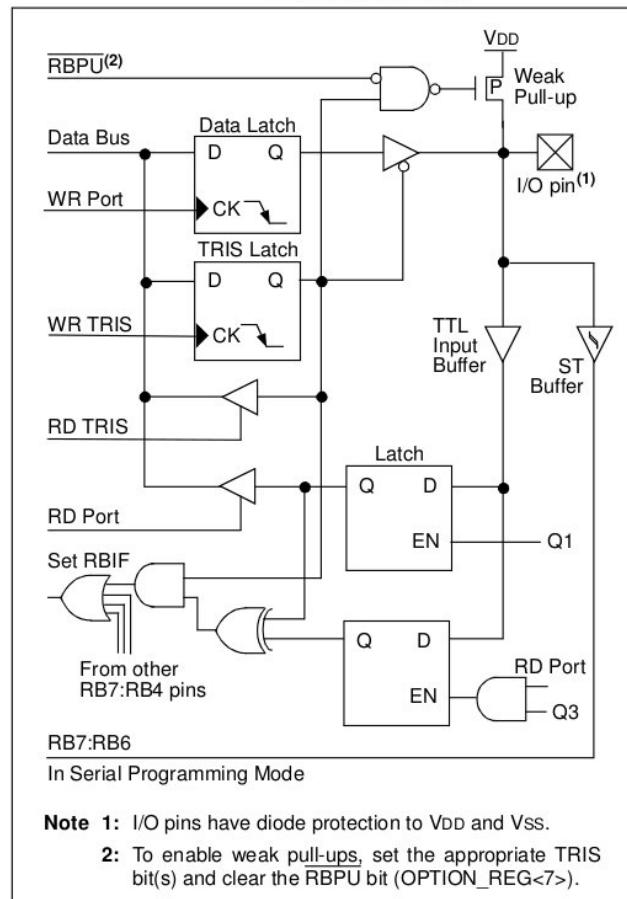


FIGURE 4-5: BLOCK DIAGRAM OF RB7:RB4 PINS



41. DOŁĄCZENIE KLAWISZY DO MIKROKONTROLERA. PROCEDURA OBSŁUGI KLAWIATURY (PRZYCISKI PODŁĄCZONE DO LINII RB3...RB0, KOD PRZYCISKU JEST ZWRACANY W REJESTRZE W)

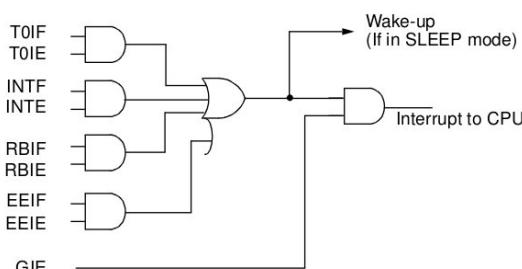
Klawiatura jest elementem często stosowanym w układach sterowania. Jednak styki klawiatury są elementami mechanicznymi dlatego styki podczas przełączania drgają, powodując powstanie serii impulsów. Organia te mogą trwać od 100 µs do 15 ms.

Organie musi być eliminowane tak, aby po naciśnięciu klawisza mógł być odczytany jego stabilny stan.

Algorytm obsługi klawiatury.

1. Czekanie na puszczenie przycisku.
2. Czekanie na przyciśnięcie przycisku.
3. Odliczanie opóźnienia eliminującego drgania styków (ok. 20 ms).
4. Sprawdzenie stanu klawiatury:
 - Jeżeli żaden przycisk nie jest wciśnięty, to powrót do punktu 2 (wykryto zakłócenie).
 - Inaczej (wykryto wciśnięcie) generowanie kodu przyciśniętego klawisza.
5. Czekanie na puszczenie przycisku.

42. UKŁAD PRZERWAŃ. UOGÓLNIONY SCHEMAT UKŁADU PRZERWAŃ. REJESTRY STEROWANIA UKŁADEM PRZERWAŃ



Ilustracja 2: Układ przerwań mikrokontrolerów PIC16F8x

Mikrokontrolery PIC16 mogą obsługiwać wiele źródeł przerwań. Zwykle jest to jedno przerwanie od jednego urządzenia peryferyjnego. Niektóre moduły mogą generować kilka przerwań (np. moduł USART).

Źródła przerwań mogą być następujące:

1. wyprowadzenie INT (przerwanie zewnętrzne);
2. przepełnienie rejestru licznika TMR0;
3. zmiana stanu na wejściach RB7:RB4;
4. zmiana stanu komparatora;
5. port równoległy PSP;
6. moduł USART;
7. odbiornik portu szeregowego;
8. nadajnik portu szeregowego;
9. zakończenie konwersji w przetworniku A/C;
10. wyświetlacz LCD;
11. zakończenie zapisu w pamięci EEPROM;
12. przepełnienie rejestru licznika TMR1;
13. przepełnienie rejestru licznika TMR2;
14. moduły CCP1 i CCP2;
15. moduł transmisji SSP

W mikrokontrolerze istnieje **przynajmniej jeden rejestr** do sterowania maskowaniem przerwań oraz do sprawdzania ich stanu. Jest to rejestr **INTCON**. Dodatkowo, jeżeli mikrokontroler posiada urządzenia peryferyjne, to posiada również **dodatkowe rejesty** do zarządzania przerwaniami od tych urządzeń (np. PIC16F877A). W zależności od typu mikrokontrolera mogą to być rejesty: **PIE1; PIR1; PIE2; PIR2**.

W rejestrze INTCON znajduje się bit **GIE** (Global Interrupt Enable), który włącza (1) lub wyłącza (0) wszystkie przerwania.

Poszczególne przerwania mogą być uaktywniane lub wyłączone za pomocą odpowiednich bitów w rejestrze INTCON lub w rejestrach PIE1 i PIE2. Bit GIE jest zawsze wyzerowany podczas restartu mikrokontrolera.

Instrukcja powrotu z przerwania (**RETFIE**) wychodzi z procedury przerwania jak również **ustawia bit GIE**, co pozwala na obsługę kolejnego przerwania.

W rejestrze INTCON znajdują się zawsze **bity** sterujące przerwaniami od:

- Wejścia zewnętrzne INT,
- zmiany stanu na wyprowadzeniach RB7:RB4 portu B
- przepełnienia licznika układu TIMER0.

Rejestr INTCON często posiada także bit **PEIE** (Peripheral Interrupt Enable) do aktywacji przerwań od układów peryferyjnych mikrokontrolera.

Kiedy nadchodzi przerwanie bit GIE zostaje wyzerowany, aby wyłączyć przerwania przychodzące, adres powrotu jest odkładany na stosie, a licznik rozkazów jest ładowany wartością 0004h.

Następnie **należy zidentyfikować** przerwanie poprzez sprawdzanie znaczników **flag** poszczególnych **przerwań** w rejestrach INTCON, PIR1 i PIR2.

Ustawiony znacznik przerwania powinien **zostać** z powrotem **wyzerowany programowo** przed ponownym włączeniem wszystkich przerwań, aby uniknąć rekurencji.

Znaczniki identyfikujące przerwania, są ustawiane w wyniku nastąpienia jakiegoś zdarzenia niezależnie od stanu bitów maskujących przerwania, w tym bitu GIE. Kiedy instrukcja zerująca bit GIE jest wykonywana, inne przerwania czekające na obsługę w następnym cyklu są ignorowane.

Jednostka centralna wykonuje instrukcję NOP w następnym cyklu po instrukcji zerowania bitu GIE. Przerwania, które zostały zignorowane nadal czekają na obsługę do czasu, aż bit GIE zostanie z powrotem ustawiony na jedynkę.

Pozostawienie wyzerowanego bitu GIE nie blokuje ustawiania flag wystąpienia niezamaskowanych przerwań przez układ sterowania mikrokontrolera.

Jeśli mikrokontroler jest w stanie uśpienia, gdy nadejdzie przerwanie jest on **wybudzany**, a następnie:

1. Jeżeli GIE=1 wykonywana jest pierwsza instrukcja po rozkazie SLEEP, następnie zostaje wywołana (CALL) procedura obsługi przerwania z jednoczesnym wyzerowaniem bitu GIE do momentu wyjścia z procedury obsługi przerwania (RETFIE)

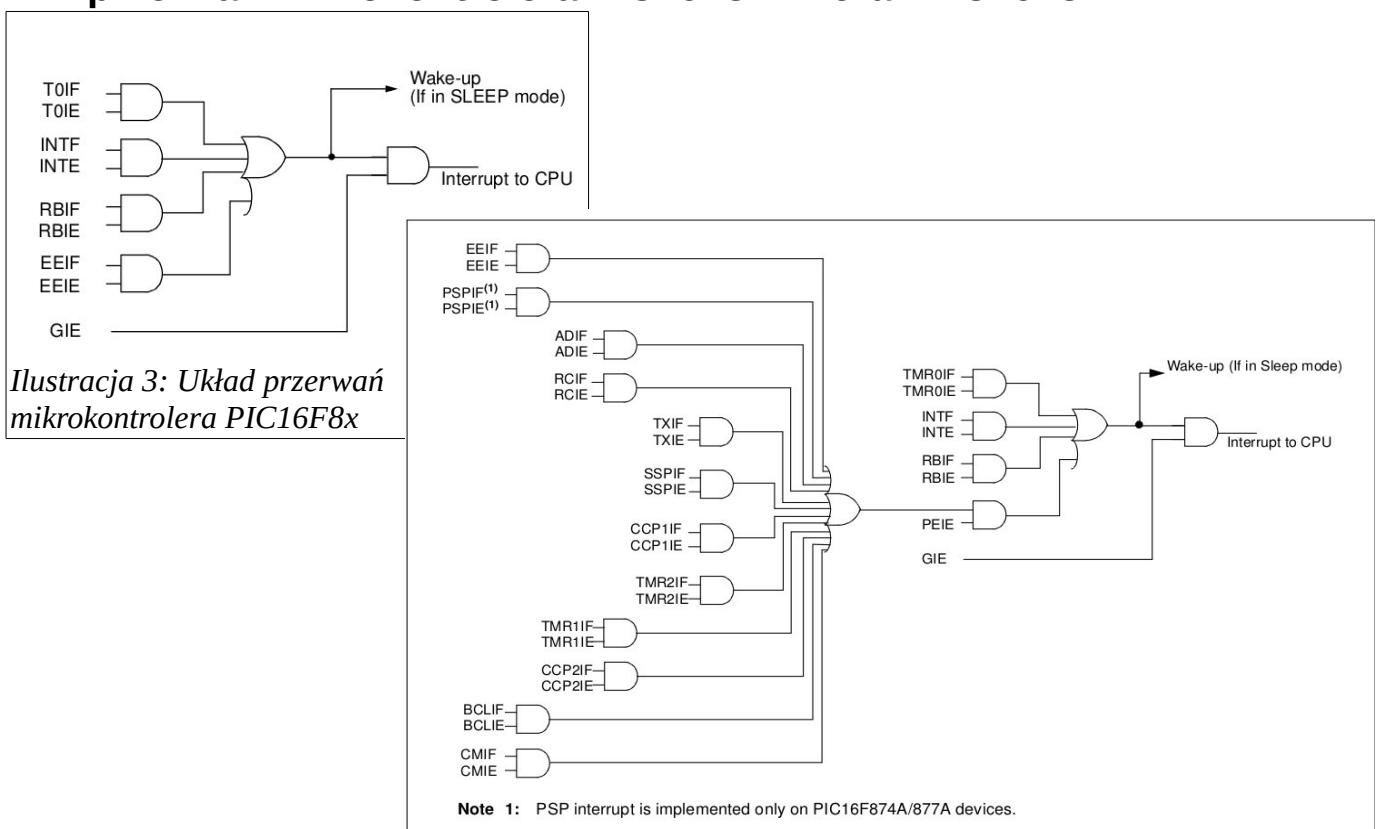
Aby zablokować wykonywanie pierwszej instrukcji po rozkazie sleep należy bezpośrednio po rozkazie SLEEP umieścić instrukcje NOP.

2. Jeżeli GIE=0 program wykonuje się od kolejnej instrukcji za rozkazem SLEEP, procedura obsługi przerwania nie jest wywoływana.

43. OBSŁUGA PRZERWAŃ. ALGORYTM PROCEDURY OBSŁUGI PRZERWAŃ. STRUKTURA PROCEDURY OBSŁUGI PRZERWANIA

Patrz zadanie 20, 42

44. Układ przerwań mikrokontrolera PIC16F877A. Struktura układu przerwań mikrokontrolera PIC16F877A oraz PIC16F84



Ilustracja 4: Układ przerwań mikrokontrolerów PIC16F877A

45. PRZERWANIE OD UKŁADU TIMER0. ALGORYTM PROCEDURY OBSŁUGI PRZERWANIA OD UKŁADU TIMER0

Możliwość identyfikacji przerwania od układu czasowego TIMER0 istnieje zarówno w mikrokontrolerze PIC16F877A, jak i w mikrokontrolerze PIC16F84. Przerwanie jest generowane wówczas, gdy rejestr TMR0 zmieni swój stan **z wartości FFh na 00h**.

Kiedy nastąpi przepełnienie licznika, zostaje ustalony znacznik **T0IF** w rejestrze **INTCON**. Przerwanie to może zostać wyłączone poprzez wyzerowanie bitu **T0IE** w rejestrze **INTCON**, jednak nie blokuje to możliwości ustawienia znacznika **T0IF** podczas przepełnienia rejestrów TMR0. Bit znacznika **T0IF** musi być **wyzerowany w procedurze obsługi przerwania** przed powtórnym włączeniem przerwań.

Algorytm obsługi przerwania TIMER0 pokazano na przykładzie

```
w_temp      EQU 0x7D      ; zmienne używane do zachowywania kontekstu
status_temp EQU 0x7E      ; (zawartości rejestrów)
pclath_temp EQU 0x7F      ;

ORG      0x000          ; wektor zerowania
nop
goto     MAIN           ; skok do głównego programu
ORG      0x004          ; wektor przerwania
movwf    w_temp         ; zapisz zawartości rejestrów:
movf     STATUS,w       ; W
movwf    status_temp    ; STATUS
movf     PCLATH,w       ; PCLATH
movwf    pclath_temp    ;
banksel  INTCON        ; przejdź do banku 0
btfs    INTCON, TOIF   ; sprawdź czy przyszło przerwanie od TIMER0
goto     INT_END        ; jeśli nie to przejdź na koniec
bcf    INTCON, TOIF    ; jeśli tak to wyzeruj flagę TOIF
...                  ; obsłuz przerwanie
...
INT_END
movf    pclath_temp,w  ; przywróć poprzednie zawartości rejestrów
movwf   PCLATH         ; PCLATH
movf    status_temp,w  ; STATUS
movwf   STATUS          ; W
swapf   w_temp,f       ;
swapf   w_temp,w       ;
retfie                         ; powrót z przerwania

MAIN          ; program główny
; inicjalizacja preskalera
bsf    STATUS, RP0      ; wybór banku 1 w celu uzyskania dostępu do OPTION_REG
movf   OPTION_REG, 0
andlw b'10000000'        ; wyczyść wszystkie bity poza (7)
iorlw b'01000000'        ; INTEDG=1, TOCS=0, TOSE=0, PSA=0, PS = 1/2
movwf  OPTION_REG
bcf    STATUS, RP0      ; wybór banku 0

bsf    INTCON, GIE      ; włącz przerwania
bsf    INTCON, TOIE     ; włącz przerwania od TMR0

_E
movlw .123            ; początkowa wartość licznika TMR0
movwf   TMR0            ;
sleep                ; przejdź w tryb uśpienia do czasu wystąpienia przerwania
goto    _E              ; po obsłużeniu przerwania
```

46. REJESTR INTCON. STRUKTURA I BITY REJESTRU INTCON. REJESTRY PIE1 I PIE2. STRUKTURA I BITY REJESTRÓW PIE1 I PIE2. REJESTRY PIR1 I PIR2. STRUKTURA I BITY REJESTRÓW PIR1 I PIR2

Rejestr INTCON jest podstawowym rejestrem służącym do konfiguracji systemu przerwań.

| | | | | | | | |
|------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| GIE | EEIE | T0IE | INTE | RBIE | T0IF | INTF | RBIF |
| R/W – 0 | R/W – 0 | R/W – 0 | R/W – 0 | R/W – 0 | R/W – 0 | R/W – 0 | R/W – x |

Legenda: R – bit może być odczytywany, W – zapisywany, – n – wartość po włączeniu zasilania (POR)

- bit 7 (**GIE**): włączenie lub wyłączenie wszystkich przerwań (bit GIE posiada najwyższy priorytet: Global Interrupt Enable):
 - 1 - włączenie wszystkich nie zamaskowanych przerwań,
 - 0 - wyłączenie wszystkich przerwań,
- bit 6:
 - dla PIC16F84 (**EEIE**): uaktywnienie przerwania wywoływanego po zakończeniu zapisu danej do pamięci EEPROM wewnętrz procesora:
 - 1 - po zakończeniu zapisu zostanie wywołane przerwanie,
 - 0 - po zakończeniu zapisu nie zostanie wywołane przerwanie,
 - dla PIC16F877A (**PEIE**): uaktywnienie przerwań od urządzeń peryferyjnych:
 - 1 – uaktywnia przerwania od urządzeń peryferyjnych,
 - 0 - wyłącza przerwania od urządzeń peryferyjnych,
- bit 5 (**T0IE**): bit uaktywniający wywołanie przerwania po przepełnieniu licznika TIMER0:
 - 1 - jeżeli chcemy aby przerwanie zostało wywoływane po przepełnieniu TIMER0,
 - 0 - jeżeli nie chcemy aby przerwanie zostało wywoływane po przepełnieniu TIMER0,
- bit 4 (**INTE**): uaktywnienie przerwania zewnętrznego (z wyprowadzenia RB0/INT):
 - 1 - przerwanie aktywne,
 - 0 - przerwanie nie aktywne,
- bit 3 (**RBIE**): włączenie przerwania pochodzącego od zmiany stanu na porcie PORTB:
 - 1 - przerwanie aktywne,
 - 0 - przerwanie nie aktywne,
- bit 2 (**T0IF**): bit informujący o przerwaniu spowodowanym przepełnieniem licznika TIMER0:
 - 1 - nastąpiło przepełnienie licznika (bit ten powinien być zerowany programowo),
 - 0 - nie nastąpiło przepełnienie licznika,

- bit 1 (**INTF**): bit informujący o tym, że ostatnie przerwanie zostało wywołane odpowiednią zmianą sygnału na końcówce RB0/INT:
 - 1 - nastąpiło przerwanie pochodzące od RB0/INT,
 - 0 - nie nastąpiło przerwanie pochodzące od RB0/INT,
- bit 0 (**RBIF**): bit informujący o tym, że ostatnie przerwanie zostało wywołane zmianą stanu na jednym z wyprowadzeń portu B (RB7:RB4):
 - 1 - jeżeli nastąpiła zmiana stanu na jednym z wyprowadzeń RB7:RB4 (bit musi być kasowany programowo),
 - 0 - jeżeli nie było zmian stanów na wyprowadzeniach RB7:RB4.

PIR1, PIR2, PIE1, PIE2

Rejestry PIE1 i PIE2 występują w mikrokontrolerze PIC16F877A (nie występują w PIC16F8x) są dodatkowymi rejestrami służącymi do aktywacji lub maskowania **przerwań od układów peryferyjnych**.

Aby identyfikacja tych przerwań była możliwa konieczne jest włączenie całego systemu przerwań od układów **peryferyjnych**, co wykonuje się poprzez ustawienie na 1 bitu **PEIE** w rejestrze **INTCON**.

| Address | Name | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 | Value on: POR, BOR |
|--------------------|--------|----------------------|-------|--------|-------|-------|--------|--------|--------|-----------------------|
| 0Bh ⁽³⁾ | INTCON | GIE | PEIE | TMR0IE | INTE | RBIE | TMR0IF | INTF | RBIF | 0000 000x |
| 0Ch | PIR1 | PSPIF ⁽³⁾ | ADIF | RCIF | TXIF | SSPIF | CCP1IF | TMR2IF | TMR1IF | 0000 0000 |
| 0Dh | PIR2 | — | CMIF | — | EEIF | BCLIF | — | — | CCP2IF | -0-0 0--0 |
| 8Ch | PIE1 | PSPIE ⁽²⁾ | ADIE | RCIE | TXIE | SSPIE | CCP1IE | TMR2IE | TMR1IE | 0000 0000 |
| 8Dh | PIE2 | — | CMIE | — | EEIE | BCLIE | — | — | CCP2IE | -0-0 0--0 |

PIR1

- bit 7 (PSPIF): bit informujący o przerwaniu od portu równoległego PSP:
 - 1 – miała miejsce operacja odczytu lub zapisu (bit musi być zerowany programowo),
 - 0 – nie było operacji odczytu i zapisu,
- bit 6 (ADIF): bit informujący o przerwaniu od przetwornika A/C
 - 1 - konwersja A/C została zakończona,
 - 0 - konwersja A/C nie została zakończona,
- bit 5 (RCIF): bit informujący o przerwaniu od bufora odbiornika USART:
 - 1 – bufor odbiornika jest pełny (dane zostały odebrane),
 - 0 – bufor odbiornika jest pusty,
- bit 4 (TXIF): bit informujący o przerwaniu od bufora nadajnika USART:
 - 1 – bufor nadajnika jest pusty (dane zostały wysłane),
 - 0 – bufor nadajnika jest pełny;
- bit 3 (SSPIF): bit informujący o przerwaniu od synchronicznego portu szeregowego SSP:
 - 1 – wystąpiło przerwanie od SSP (bit musi być zerowany programowo),
 - 0 – nie było przerwania od SSP,

- bit 2 (CCP1IF): bit informujący o przerwaniu od modułu **CCP1** (Capture, Compare, PWM): tryb **Capture**:
 - 1 – wykryto operację przechwytywania w rejestrze TMR1 (bit musi być zerowany programowo),
 - 0 – nie wykryto operacji przechwytywania w rejestrze TMR1;
- bit 1 (TMR2IF): bit informujący o przerwaniu od licznika TIMER2:
 - 1 – wykryto, że zawartości rejestrów TMR2 i PR2 są identyczne (zero progr.)
 - 0 - zawartości rejestrów TMR2 i PR2 są różne,
- bit 0 (TMR1IF): bit informujący o przerwaniu od licznika TIMER1:
 - 1 - nastąpiło przepełnienie licznika TIMER1 (bit ten powinien być zerowany programowo),
 - 0 - nie nastąpiło przepełnienie licznika TIMER1.

PIR2

- bit 6 (CMIF): bit informujący o przerwaniu pochodząącym od komparatora:
 - 1 – wejście komparatora zmieniło się (bit musi być zerowany programowo),
 - 0 - wejście komparatora nie zmieniło się,
- bit 4 (EEIF): bit informujący o przerwaniu pochodzącym od pamięci EEPROM:
 - 1 – operacja zapisu do pamięci EEPROM została zakończona (zero progr.)
 - 0 – operacja zapisu do pamięci EEPROM trwa lub się nie rozpoczęła,
- bit 3 (BCLIF): bit informujący o przerwaniu wywoływanym przy kolizji na magistrali I2C:
 - 1 – **wystąpiła kolizja** na magistrali I2C w module SSP skonfigurowanym w trybie I2C,
 - 0 – nie wykryto kolizji,
- bit 0 (CCP2IF): bit informujący o przerwaniu pochodzącym od modułu CCP2 (Capture, Compare, PWM):

PIE1

- bit 7 (PSPIE): aktywacja przerwania od zapisu/odczytu z portu równoległego PSP:
 - 1 - włączenie przerwania od PSP,
 - 0 - wyłączenie przerwania od PSP,
- bit 6 (ADIE): aktywacja przerwania od przetwornika A/C
 - 1 - włączenie przerwania od przetwornika A/C,
 - 0 - wyłączenie przerwania od przetwornika A/C,
- bit 5 (RCIE): aktywacja przerwania od zakończenia operacji odbioru bajtu przez moduł USART:
 - 1 - włączenie przerwania od USART,
 - 0 - wyłączenie przerwania od USART,
- bit 4 (TXIE): aktywacja przerwania od zakończenia operacji wysyłania bajtu przez moduł USART:
 - 1 - włączenie przerwania od USART,
 - 0 - wyłączenie przerwania od USART,

- bit 3 (SSPIE): włączenie przerwania pochodzącego od synchronicznego portu szeregowego SSP:
 - 1 - przerwanie od SSP aktywne,
 - 0 - przerwanie od SSP nieaktywne,
- bit 2 (CCP1IE): włączenie przerwania pochodzącego od modułu CCP1 (Capture, Compare, PWM):
 - 1 - przerwanie od CCP1 aktywne,
 - 0 - przerwanie od CCP1 nieaktywne,
- bit 1 (TMR2IE): aktywacja przerwania wywoływanego po przepłynięciu licznika TIMER2:
 - 1 - przerwanie od TIMER2 aktywne,
 - 0 - przerwanie od TIMER2 nieaktywne,
- bit 0 (TMR1IE): aktywacja przerwania wywoływanego po przepłynięciu licznika TIMER1:
 - 1 - przerwanie od TIMER1 aktywne,
 - 0 - przerwanie od TIMER1 nieaktywne.

PIE2

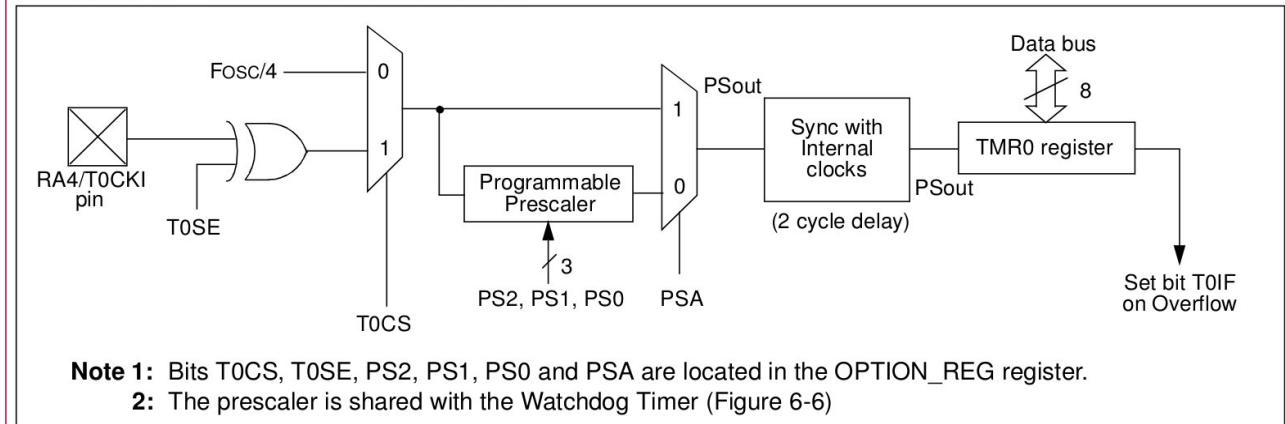
- bit 6 (CMIE): włączenie przerwania pochodzącego od komparatora:
 - 1 - przerwanie od komparatora aktywne,
 - 0 - przerwanie od komparatora nieaktywne,
- bit 4 (EEIE): włączenie przerwania pochodzącego pamięci EEPROM:
 - 1 - przerwanie od pamięci EEPROM aktywne,
 - 0 - przerwanie od pamięci EEPROM nieaktywne,
- bit 3 (BCLIE): aktywacja przerwania wywoływanego przy kolizji na magistrali I2C:
 - 1 - przerwanie od magistrali I2C aktywne,
 - 0 - przerwanie od magistrali I2C nieaktywne,
- bit 0 (CCP2IE): włączenie przerwania pochodzącego od modułu CCP2 (Capture, Compare, PWM):
 - 1 - przerwanie od CCP2 aktywne,
 - 0 - przerwanie od CCP2 nieaktywne.

Rejestry PIR1 i PIR2 występują w mikrokontrolerze PIC16F877A (nie występują w PIC16F8x) są dodatkowymi rejestrami służącymi do identyfikacji przerwań od układów peryferyjnych.

Przerwanie jest zgłoszane przez dane urządzenie peryferyjne, jeżeli odpowiedni bit w jednym z tych rejestrów jest ustawiony na jedynkę. **Wyłączanie przerwań** w rejestrach **PIE1** i **PIE2** nie blokuje możliwości ustawiania znaczników w rejestrach PIR1 i PIR2. Znaczniki w rejestrach **PIR1** i **PIR2** powinny być **zerowane programowo** po zakończeniu obsługi przerwania.

47. UKŁADY CZASOWO-LICZNIKOWE. UKŁAD CZASOWY TIMER0. INFORMACJE OGÓLNE

FIGURE 6-1: TMR0 BLOCK DIAGRAM



Układ czasowy TIMER0 jest 8-bitowym czasomierzem lub licznikiem impulsów. Posiada on następujące właściwości:

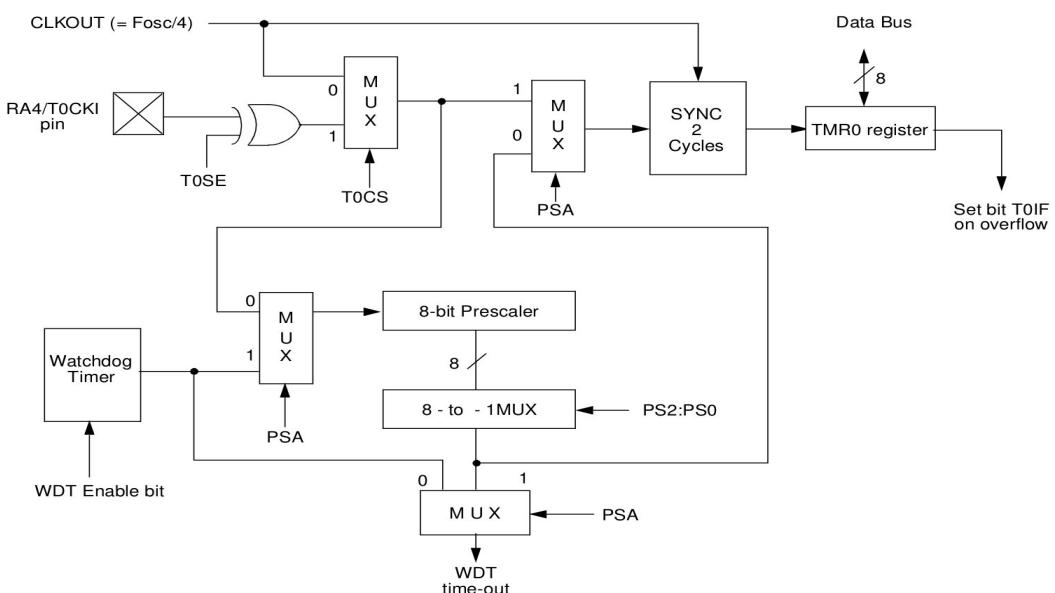
1. możliwość odczytu i zapisu;
2. możliwość dołączenia 8-bitowego preskalera;
3. możliwość wyboru źródła zliczanych impulsów (wewnętrzne lub zewnętrzne);
4. możliwość wyboru zbocza zewnętrznego zegara, na które układ będzie reagował;
5. możliwość wywołania przerwania przy przepłynięciu licznika (zmianie zawartości z FFh na 00h).

Układ czasowy TIMER0 występuje zarówno w mikrokontrolerze PIC16F877A, jak i w mikrokontrolerze PIC16F8x. Sposób działania tego układu czasowego dla obu mikrokontrolerów jest taki sam.

Mikrokontroler PIC16F877A oprócz TIMER0 posiada też liczniki:

- 16-bitowy TIMER1
- 8-bitowy TIMER2

SCHEMAT BLOKOWY UKŁADU CZASOWEGO TIMER0 Z PRESKALEREM



Note: T0CS, T0SE, PSA, PS2:PS0 are bits in the OPTION_REG register.

48. ZLICZANIE IMPULSÓW ZEWNĘTRZNYCH W UKŁADZIE CZASOWYM TIMER0

Kiedy preskaler jest niewykorzystywany, wejście zegara zewnętrznego jest tożsame z wyjściem preskalera.

Synchronizacja sygnału z wejścia T0CKI z zegarem wewnętrznym odbywa się za pomocą próbkowania wyjścia preskalera w cyklach Q2 i Q4 zegara wewnętrznego. Dlatego niezbędne jest to, aby sygnał z wejścia T0CKI miał poziom wysoki przynajmniej przez czas dwóch okresów oscylatora (oraz krótkie opóźnienie wynoszące 20 ns) oraz poziom niski także przynajmniej przez czas równy dwóm okresom oscylatora (z opóźnieniem 20 ns).

Kiedy preskaler jest używany, częstotliwość sygnału zegara zewnętrznego jest dzielona przez szeregowy licznik asynchroniczny i na wyjściu preskalera otrzymujemy przebieg symetryczny. Z tego powodu przy określaniu wymagań próbkowania należy wziąć pod uwagę szeregowy licznik asynchroniczny. Niezbędne jest, aby sygnał na wejściu T0CKI miał okres przynajmniej równy czterem okresom sygnału oscylatora (plus krótkie opóźnienie wynoszące 40 ns).

Z powodu synchronizacji sygnału wejściowego z wewnętrznym zegarem powstaje małe opóźnienie pomiędzy zboczem sygnału z wejścia T0CKI a momentem inkrementacji rejestru TMR0.

48. PRZEPEŁNIENIE LICZNIKA TIMER0. DOŁĄCZENIE PRESKALERA DO UKŁADU TIMER0

Układ czasowy TIMER0 ma możliwość generacji przerwania w momencie przepełnienia licznika. Przepełnienie licznika skutkuje ustawieniem bitu T0IF w rejestrze INTCON na jedynkę.

Przerwanie to może zostać zamaskowane poprzez wyzerowanie bitu T0IE w rejestrze INTCON. Bit T0IF powinien zostać wyzerowany programowo w procedurze obsługi przerwania przed ponownym włączeniem przerwania od układu TIMER0.

Układ czasowy TIMER0 **nie jest zdolny do wybudzenia** mikrokontrolera ze stanu uśpienia (ponieważ jest taktowany z tego samego oscylatora co procesor, oscylator ten jest wyłączony w trybie uśpienia)

Układ czasowy TIMER0 posiada możliwość dołączenia wewnętrznego 8-bitowego preskalera, który jest jednocześnie postskalerem dla układu licznika nadzorczy (WDT).

Użycie preskalera z układem TIMER0 powoduje, że nie jest on dostępny dla licznika nadzorczy i odwrotnie.

Wybór układu, z którym połączony jest preskaler jest dokonywany za pomocą bitu PSA w rejestrze OPTION_REG. Jeżeli bit PSA jest ustawiony na jedynkę – preskaler jest dołączony do licznika nadzorczy (WDT), a jeśli jest wyzerowany – do układu TIMER0. Współczynnik podziału preskalera jest ustawiany za pomocą bitów PS2:PS0 w rejestrze OPTION_REG. Kiedy preskaler jest dołączony do układu TIMER0, **wszystkie instrukcje modyfikujące rejestr TMR0 zerują preskaler**. Kiedy preskaler jest dołączony do licznika nadzorczy instrukcja CLRWDT zerująca licznik nadzorczy – **zeruje także preskaler**. Do preskalera nie można zapisywać żadnych danych ani ich odczytywać.

49. REJESTRY TMR0 I OPTION_REG. KONFIGURACJA UKŁADU PRESKALERA. Inicjalizacja TIMER0: w trybie czasomierza i w trybie licznika

| | | | | | | | |
|--------------|---------------|-------------|-------------|------------|------------|------------|------------|
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| !RBPU | INTEDG | T0CS | T0SE | PSA | PS2 | PS1 | PS0 |
| R/W – 1 | R/W – 1 | R/W – 1 | R/W – 1 | R/W – 1 | R/W – 1 | R/W – 1 | R/W – 1 |

Legenda: R – bit może być odczytywany, W – zapisywany, – n – wartość po włączeniu zasilania (POR)

Rejestr OPTION_REG służy do ustawiania opcji układu czasowego TIMER0. Za pośrednictwem rejestrów OPTION_REG można także ustawić **aktywne zbocze sygnału przerwania zewnętrznego** oraz aktywować **rezystory podciągające (pull-up)** dla wyprowadzeń **portu B**.

| Nr. | Nazwa | Znaczenie | Przyjmowane wartości | uwagi | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|-------|------------------|--|--|---|-----|-----|-----|------------------|-----------------|---|---|---|-----|-----|---|---|---|-----|-----|---|---|---|-----|-----|---|---|---|------|-----|---|---|---|------|------|---|---|---|------|------|---|---|---|-------|------|---|---|---|-------|-------|
| 7 | !RBPU | Bit dołączający /odłączający wbudowane rezystory podciągające do napięcia zasilającego linii portu PORTB | = 0 – rezystory podciągające dołączone = 1 – rezystory podciągające odłączone | Rezystory podciągające domyślnie są odłączone. Ponadto znajdują się one odłączane gdy linia portu zostanie skonfigurowana jako wyjście. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 6 | INTEDG | Bit wyboru zbocza, przy którym przerwanie zewnętrzne INT ma zostać zgłoszone | = 0 – przerwanie zgłasiane przy opadającym zboczu sygnału na wyprowadzeniu RB0/INT = 1 – przerwanie zgłasiane przy narastającym zboczu sygnału na wyprowadzeniu RB0/INT | – | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 5 | T0CS | Bit wyboru źródła zliczanych impulsów dla timera TMR0 | = 0 – TMR0 ma zliczać impulsy sygnału o częstotliwości sygnału zegarowego podzielonego przez 4 = 1 – TMR0 ma zliczać impulsy sygnału z wyprowadzenia RA4/T0CKI | – | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 4 | T0SE | Bit wyboru zbocza, przy którym następuje zwiększenie wartości timera TMR0 | = 0 – zwiększenie wartości TMR0 ma następować przy narastającym zboczu na wyprowadzeniu RA4/T0CKI = 1 – zwiększenie wartości TMR0 ma następować przy opadającym zboczu na wyprowadzeniu RA4/T0CKI | – | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 3 | PSA | Bit przyporządkowania preskalera do timera TMR0 lub licznika WDT | = 0 – preskaler przyporządkowany do timera TMR0 = 1 – preskaler przyporządkowany do licznika WDT | – | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 2...0 | PS2...PS0 | Bity wyboru współczynnika preskalera | wartości podziału | <table border="1"> <thead> <tr> <th>PS2</th><th>PS1</th><th>PS0</th><th>Podział dla TMR0</th><th>Podział dla WDT</th></tr> </thead> <tbody> <tr> <td>0</td><td>0</td><td>0</td><td>1:2</td><td>1:1</td></tr> <tr> <td>0</td><td>0</td><td>1</td><td>1:4</td><td>1:2</td></tr> <tr> <td>0</td><td>1</td><td>0</td><td>1:8</td><td>1:4</td></tr> <tr> <td>0</td><td>1</td><td>1</td><td>1:16</td><td>1:8</td></tr> <tr> <td>1</td><td>0</td><td>0</td><td>1:32</td><td>1:16</td></tr> <tr> <td>1</td><td>0</td><td>1</td><td>1:64</td><td>1:32</td></tr> <tr> <td>1</td><td>1</td><td>0</td><td>1:128</td><td>1:64</td></tr> <tr> <td>1</td><td>1</td><td>1</td><td>1:256</td><td>1:128</td></tr> </tbody> </table> | PS2 | PS1 | PS0 | Podział dla TMR0 | Podział dla WDT | 0 | 0 | 0 | 1:2 | 1:1 | 0 | 0 | 1 | 1:4 | 1:2 | 0 | 1 | 0 | 1:8 | 1:4 | 0 | 1 | 1 | 1:16 | 1:8 | 1 | 0 | 0 | 1:32 | 1:16 | 1 | 0 | 1 | 1:64 | 1:32 | 1 | 1 | 0 | 1:128 | 1:64 | 1 | 1 | 1 | 1:256 | 1:128 |
| PS2 | PS1 | PS0 | Podział dla TMR0 | Podział dla WDT | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 0 | 0 | 0 | 1:2 | 1:1 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 0 | 0 | 1 | 1:4 | 1:2 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 0 | 1 | 0 | 1:8 | 1:4 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 0 | 1 | 1 | 1:16 | 1:8 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 1 | 0 | 0 | 1:32 | 1:16 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 1 | 0 | 1 | 1:64 | 1:32 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 1 | 1 | 0 | 1:128 | 1:64 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 1 | 1 | 1 | 1:256 | 1:128 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

Tryby pracy TIMER0

Układ czasowy TIMER0 może pracować w dwóch trybach: w trybie **licznika** i trybie **czasomierza**. Wyboru trybu dokonuje się za pomocą ustawień bitu T0CS w rejestrze OPTION_REG. Tryb **czasomierza** jest wybierany poprzez **wyzerowanie** bitu **T0CS** w rejestrze OPTION_REG. W tym trybie **zawartość** licznika jest **inkrementowana** w każdym **cyklu rozkazowym** (bez preskalera).

Jeżeli do rejestru licznika TMR0 jest zapisywana jakaś wartość, inkrementacja jest zawieszana na 2 następne cykle. Użytkownik powinien to skorygować poprzez wpisanie odpowiedniej wartości do rejestru TMR0.

Iinicjalizację układu TIMER0 w trybie czasomierza pokazano na przykładzie

```
; program zlicza za pomocą TIMER0 125 impulsów zegara wewnętrznego  
; z podzielnikiem 1/8 (1000 impulsów)  
bsf      STATUS, RP0      ; bank 1  
movlw    B'00000010'      ; ustaw_tryb_czasomierza, prescaler na 1/8  
movwf    OPTION_REG       ; zliczanie_impułsów zegarowych  
bcf      STATUS, RP0      ; bank 0  
movlw    .256 - .125      ; licz do 125  
movwf    TMRO
```

Tryb **licznika** wybiera się poprzez ustawienie na **jedynkę** bitu **T0CS** w rejestrze OPTION_REG. W trybie licznika zawartość licznika będzie **inkrementowana** w przypadku **nadejścia** każdego narastającego lub opadającego **zbocza** na wyprowadzeniu **RA4/T0CKI**.

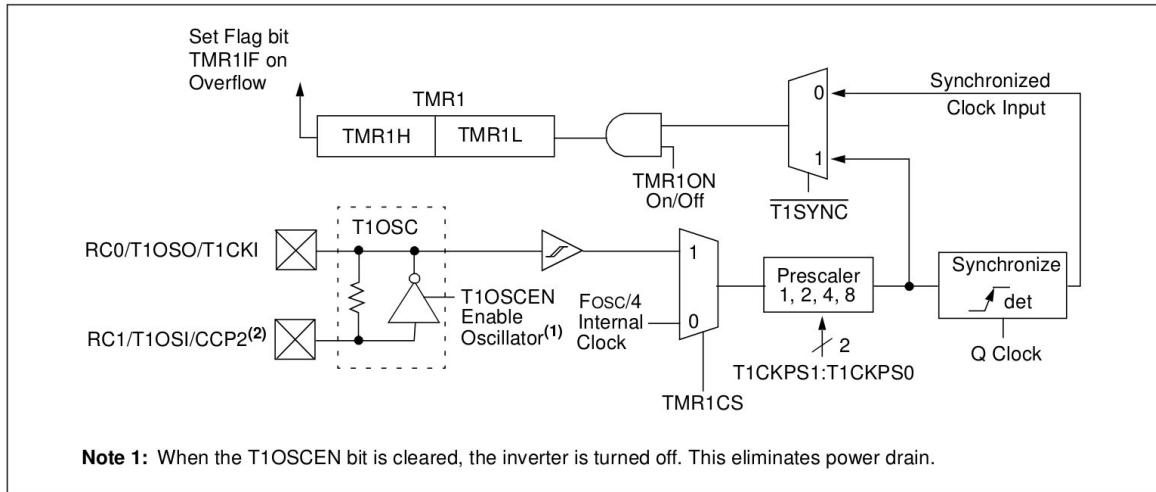
Zbocze, na które licznik ma reagować jest ustalane za pomocą bitu **T0SE** w rejestrze OPTION_REG. **Wyzerowanie** tego bitu oznacza, że układ będzie reagował na zbocze **narastające**, a ustawienie na **jedynkę** oznacza, że układ będzie reagował na zbocze **opadające** sygnału na wyprowadzeniu **RA4/T0CKI**.

Iinicjalizację układu TIMER0 w trybie licznika pokazano na przykładzie:

```
; program zlicza za pomocą TIMER0 125 impulsów zegara wewnętrznego  
; z podzielnikiem 1/4 (500 impulsów)  
bsf      STATUS, RP0      ; bank 1  
movlw    B'00100001'      ; ustaw tryb licznika, prescaler na 1/4  
movwf    OPTION_REG       ; zliczanie impulsów zegarowych  
bcf      STATUS, RP0      ; bank 0  
movlw    .256 - .125      ; licz do 125  
movwf    TMRO
```

50. Układ czasowy TIMER1. Informacje ogólne. Schemat blokowy układu czasowego TIMER1. Rejestr T1CON. Bity rejestru T1CON

FIGURE 6-2: TIMER1 BLOCK DIAGRAM



Układ czasowy TIMER1 występuje w mikrokontrolerze PIC16F877A, natomiast nie występuje w mikrokontrolerze PIC16F84. Układ TIMER1 jest **16-bitowym licznikiem** lub **czasomierzem** składającym się z dwóch rejestrów: **TMR1H** i **TMR1L**. Możliwy jest oczywiście zarówno odczyt, jak i zapis tych rejestrów. Para rejestrów **TMR1H:TMR1L** jest inkrementowana od wartości 0000h do FFFFh i po przepełnieniu – licznik zaczyna liczyć z powrotem od 0000h.

Układ posiada **preskaler**, który może wstępnie podzielić częstotliwość zliczanych impulsów przez wartość **1, 2, 4 lub 8**. Wszystkie **instrukcje zapisujące** do rejestrów **TMR1H** i **TMR1L** **zerują preskaler**.

Układ TIMER1 ma możliwość **zgłaszania przerwania**, które jest generowane przy przepełnieniu licznika. Przerwanie jest sygnalizowane poprzez ustawienie na jedynkę bitu **TMR1F** w rejestrze **PIR1**. Przerwanie to może być aktywowane lub maskowane za pomocą ustawiania lub zerowania bitu **TMRIE** w rejestrze **PIE1**. Układ czasowy TIMER1 może pracować w dwóch trybach: w trybie czasomierza i w trybie licznika. Tryb pracy jest wybierany za pomocą bitu **TMR1CS** w rejestrze **T1CON**.

W trybie **czasomierza** (**TMR1CS=0**) układ czasowy jest inkrementowany w każdym cyku rozkazowym. W trybie **licznika** układ czasowy zwiększa swoją zawartość o 1 przy nadejściu każdego narastającego zbocza zewnętrznego sygnału zegarowego.

Układ czasowy TIMER1 może być **włączany** lub **wyłączany** poprzez ustawianie lub zerowanie bitu **TMR1ON** w rejestrze **T1CON**. Układ czasowy TIMER1 posiada także wewnętrzne wejście zerujące.

Sygnal Reset może być generowany przez jeden z dwóch modułów CCP (Capture, Compare, PWM). W przypadku, gdy włączony jest oscylator zewnętrzny układu TIMER1 (bit **T1OSCEN** w rejestrze **T1CON** ustawiony na jedynkę), wyprowadzenia **RC1/T1OSI/CCP2** i **RC0/T1OSO/T1CKI** stają się wejściami. Wówczas zawartość bitów nr 1 i nr 0 rejestru **TRISC** jest ignorowana i z wyprowadzeń tych jest zawsze odczytywana wartość zero.

REJESTR T1CON. BITY REJESTRU T1CON

T1CON: TIMER1 CONTROL REGISTER (ADDRESS 10h)

| U-0 | U-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 |
|-----|-----|---------|---------|---------|--------|--------|--------|
| — | — | T1CKPS1 | T1CKPS0 | T1OSCEN | T1SYNC | TMR1CS | TMR1ON |

bit 7

bit 0

Rejestr T1CON służy do konfiguracji układu czasowego TIMER1:

- T1CKPS1, T1CKPS0 – wybór podziału częstotliwości przez preskaler; podział częstotliwości odbywa się w stosunku $1:2^N$, gdzie N jest wartością binarną, w której bity 1 i 0 są określone wartościami bitów odpowiednio T1CKPS1 i T1CKPS0;
- T1OSCEN – włączenie (wartością 1) lub wyłączenie (wartością 0) układu generatora zegarowego na wyprowadzeniach T1OSI i T1OSO; gdy generator nie jest potrzebny należy go wyłączyć w celu zmniejszenia poboru prądu przez układ;
- T1SYNC – włączenie (wartością 0) lub wyłączenie (wartością 1) synchronizacji sygnału zewnętrznego z wewnętrznym układem taktującym; gdy TMR1CS = 0 wartość tego bitu nie ma znaczenia (źródłem impulsów jest zegar systemowy);
- TMR1CS – wybór źródła impulsów, które będzie zliczał TIMER1; gdy TMR1CS = 1 źródłem impulsów jest wyprowadzenie T1CKI, dla TMR1CS = 0 impulsy pochodzą od zegara systemowego;
- TMR1ON – włączenie (wartością 1) lub wyłączenie (wartością 0) układu TIMER1.

51.1. FUNKCJONOWANIE UKŁADU TIMER1 W TRYBIE CZASOMIERZA.

Aby układ czasowy TIMER1 pracował w trybie czasomierza należy **wyzerować** bit **TMR1CS** w rejestrze **T1CON**. W tym trybie, wejściowym sygnałem zegarowym układu jest **zegar systemowy**, którego częstotliwość jest **podzielona przez 4**. Ustawienie bitu kontroli synchronizacji T1SYNC w rejestrze T1CON, w tym przypadku nie ma żadnego znaczenia, gdyż sygnał wejściowy dla układu czasowego. Jest już synchroniczny z wewnętrznym zegarem systemowym.

Przykład

```
; program zlicza za pomocą TIMER1 1250 impulsów zegara wewnętrznego  
; z podzielnikiem 1/8 (10000 impulsów)  
    clrf STATUS          ; bank 0  
    movlw B'00110000'    ; ustaw tryb czasomierza, prescaler na 1/8  
    movwf T1CON  
    movlw 1Eh            ; licz do 1250 -> 65536 - 1250 = 64286 (0FB1Eh)  
    movwf TMR1L          ; załaduj młodszą część  
    movlw 0FB             ;  
    movwf TMR1H          ; załaduj starszą część  
    bsf    T1CON, TMR1ON ; włącz TIMER1
```

51.2. FUNKCJONOWANIE UKŁADU TIMER1 W TRYBIE LICZNIKA SYNCHRONICZNEGO I ASYNCHRONICZNEGO

Układ czasowy TIMER1 może pracować w trybie **synchronicznym** lub **asynchronicznym**, zależnie od ustawienia bitu **T1SYNC** rejestrze **T1CON**. Gdy TIMER1 jest inkrementowany ze źródła zewnętrznego, jego wartości +1. Następuje przy narastającym zboczu sygnału. Po przełączeniu ukł czasowego w tryb licznika, ukł. musi odczytać jedno zbocze opadające przed rozpoczęciem zliczania zbocz narastających sygnału wejść.

W przypadku pracy ukł. TIMER1 w trybie licznika asynch., UC jest inkrementowany przy każdym narastającym zboczu na wyprowadzeniu RC1/T1OSI/CCP2, kiedy bit T1OSCEN w rejestrze T1CON jest ustawiony na 1 oraz wyprowadzeniu RC0/T1OSO/T1CKI, gdy bit T1OSCEN jest 0. Jeżeli bit **T1SYNC** w rejestrze **T1CON** jest 0, to wejśc. sygnał zegarowy jest **synchronizowany** z zegarem systemowym.

Synchronizacja jest wykonywana po podziale częstotl. syg. przez preskaler, który jest szeregowym licznikiem asynchr. Przy takiej konfiguracji, podczas trybu uśpienia, ukł. TIMER1 nie będzie inkrementowany nawet, gdy zewn. sygnał zegarowy jest obecny, ponieważ układ synchronizacji jest wyłączony. Jednakże układ preskalera dalej będzie kontynuował swoje działanie.

```
; program zlicza za pomocą TIMER1 1250 impulsów zewnętrznych  
; z podzielnikiem 1/8 (10000 impulsów)  
  
    clrf STATUS      ; bank 0  
    movlw B'00111010' ; ustaw tryb licznika synchronicznego,  
    movwf T1CON      ; preskaler na 1/8  
    movlw 1Eh         ; licz do 1250 -> 65536 - 1250 = 64286 (0FB1Eh)  
    movwf TMR1L      ; załaduj młodszą część  
    movlw 0FB          ;  
    movwf TMR1H      ; załaduj starszą część  
    bsf   T1CON, TMR1ON ; włącz TIMER1
```

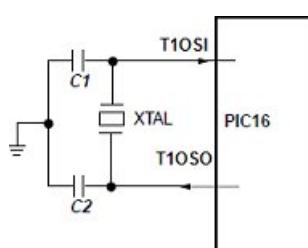
Jeżeli bit **T1SYNC** w rejestrze T1CON jest ustawiony na **jedynkę**, wejściowy sygnał zegarowy nie jest synchronizowany z zegarem systemowym. Wówczas układ TIMER1 pracuje asynchronicznie w stosunku do zegara systemowego. Układ czasowy **kontynuuje** swoje **działanie** po wprowadzeniu mikrokontrolera w **stan uśpienia** i może wygenerować **przerwanie** przy **przepełnieniu licznika**, które wyprowadza mikrokontroler ze stanu uśpienia. W trybie asynchronicznym układ czasowy TIMER1 **nie może** być wykorzystywany jako **podstawa** czasu dla **modułów CCP**.

```

; program zlicza za pomocą TIMER1 1250
; impulsów zewnętrznych
; z podzielnikiem 1/8 (10000 impulsów)
    clrf STATUS           ; bank 0
    movlw B'00111110'     ; ustaw tryb licznika asynchronicznego,
    movwf T1CON            ; preskaler na 1/8
    movlw 1Eh               ; licz do 1250 -> 65536 - 1250 = 64286 (0FB1Eh)
    movwf TMR1L             ; załajduj młodszą część
    movlw OFB                ;
    movwf TMR1H             ; załajduj starszą część
    bsf   T1CON, TMR1ON      ; włącz TIMER1

```

51.3. ŹRÓDŁO SYGNAŁU ZEGAROWEGO DLA UKŁADU TIMER1. SPOSÓB PODŁĄCZENIA ELEMENTÓW DODATKOWYCH ŽRÓDŁA SYGNAŁU ZEGAROWEGO DLA UKŁADU CZASOWEGO TIMER1. ZEROVANIE UKŁADU TIMER1



Pomiędzy wyprowadzeniami **T1OSI** a **T1OSO** jest wbudowany **oscylator kwarcowy**, który może być uaktywniany poprzez ustawienie na **jedynkę** bitu **T1OSEN** w rejestrze **T1CON**. Jest to oscylator o niskim poborze mocy o częstotliwości **do 200 kHz**. **Kontynuuje** on także swoją pracę po wprowadzeniu mikrokontrolera **w tryb uśpienia**. Oscylator jest przystosowany do pracy z rezonatorem kwarcowym o częstotliwości 32 kHz.

Rejestry TMR1H i TMR1L nie są zerowane przy włączeniu zasilania (POR), ani przy żadnym innym trybie zerowania. Rejestry te są zerowane tylko przez moduły CCP. Rejestr T1CON jest zerowany przy włączeniu zasilania (POR) i podczas spadku napięcia zasilania (BOR). Każdy inny typ zerowania nie ma wpływu na zawartość tego rejestru.

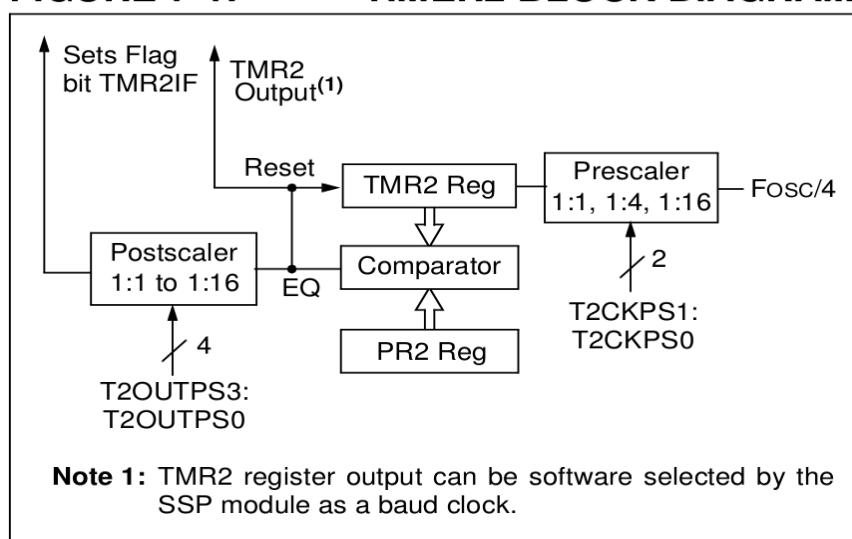
Jeżeli moduł CCP jest skonfigurowany w trybie komparatora i generuje specjalne zdarzenie (bity CCP1M3:CCP1M0 = 1011), to sygnał tego zdarzenia zeruje układ czasowy TIMER1. Operacja ta nie ustawia znacznika przerwania TMR1IF od układu TIMER1. Układ czasowy musi być w tym przypadku skonfigurowany jako czasomierz albo licznik synchroniczny.

Jeżeli układ TIMER1 pracuje w trybie licznika asynchronicznego, taka operacja zerowania może nie działać i nie powinna być używana. Jeżeli jest wykonywany zapis do układu TIMER1 i jednocześnie układ jest zerowany przez moduł CCP, pierwszeństwo ma operacja zapisu.

52.1. UKŁAD CZASOWY TIMER2. INFORMACJE OGÓLNE. SCHEMAT BLOKOWY UKŁADU CZASOWEGO TIMER2

Układ czasowy TIMER2 występuje w mikrokontrolerze PIC16F877A, natomiast nie występuje w mikrokontrolerze PIC16F84. Układ TIMER2 jest 8-bitowym czasomierzem składającym się z dwóch rejestrów: **TMR2** i **PR2**. Rejestr TMR2 może być odczytywany, jak i zapisywany oraz jest zerowany podczas każdego restartu mikrokontrolera.

FIGURE 7-1: TIMER2 BLOCK DIAGRAM



52.2. REJESTR T2CON. PRESKALER I POSTSKALER DLA UKŁADU TIMER2

T2CON: TIMER2 CONTROL REGISTER (ADDRESS 12h)

bit 7 **Unimplemented:** Read as ‘0’

bit 6-3 **TOUTPS3:TOUTPS0:** Timer2 Output Postscale Select bits

0000 = 1:1 postscale

0001 = 1:2 postscale

0010 = 1:3 postscale

1

1

1

1111 = 1:16 postscale

bit 2 **TMR2ON**: Timer2 On bit

1 = Timer2 is on

0 = Timer2 is off

T2CKPS1-T2CK

00 – Prescaler is 1

01 = Prescaler is 4

1x = Prescaler is 1

III. Results

podziału czę

T2CKPS0 rejestru T2CON. Podział częstotliwości sygnału zegarowego odbywa się w stosunku $1:2N+1$, gdzie N jest wartością binarną, w której bity 1 i 0 są określone wartościami bitów odpowiednio T2CKPS1 i T2CKPS0, przy czym dla N=2 i N=3 wartość podziału wynosi 16.

Podział częstotliwości postskalera ustala się za pomocą bitów TOUTPS3, TOUTPS2, TOUTPS1, TOUTPS0. Podział ten odbywa się w stosunku 1:(N+1), gdzie N jest wartością binarną, w której bity 3, 2, 1 i 0 są określone wartościami bitów odpowiednio TOUTPS3, TOUTPS2, TOUTPS1 i TOUTPS0.

Układy preskalera i postskalera są zerowane w przypadku wystąpienia następujących sytuacji:

- zapisu do rejestru TMR2;
- zapisu do rejestru T2CON;
- każdego typu zerowania mikrokontrolera (POR, MCLR, WDT lub BOR).

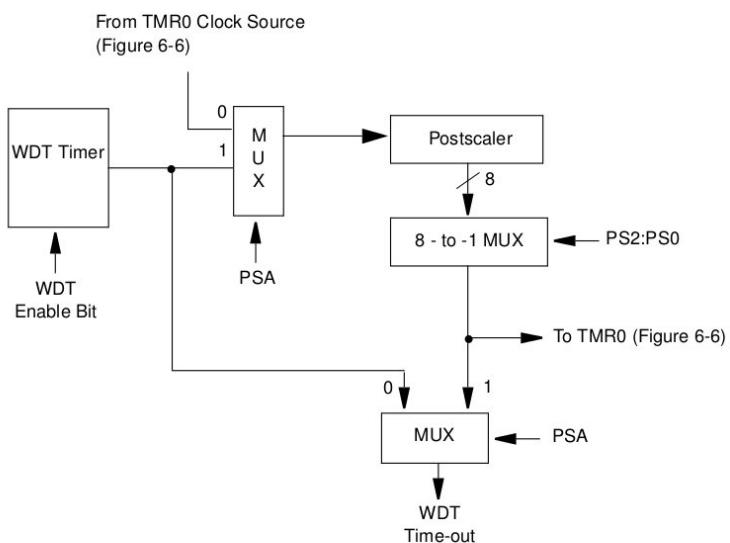
Rejestr TMR2 nie jest zerowany w przypadku zapisu do rejestru T2CON.

52.3. Inicjalizacja układu TIMER2

```
; program zlicza za pomocą TIMER2
; 125 impulsów zegara wewnętrznego
; z preskalerem 1/8 i postskalerem 1/4

clrf STATUS           ; bank 0
movlw B'0001101'       ; prescaler na 1/8 ; postskaler na 1/4
movwf T2CON            ;
movlw .125             ; licz do 125
bsf STATUS,RPO         ; bank 1
movwf PR2               ;
bcf STATUS,RPO         ; bank 0
bsf T2CON,TMR2ON       ; włącz TIMER2
```

53. UKŁAD LICZNIKA NADZORCY WDT. SCHEMAT BLOKOWY LICZNIKA NADZORCY WDT. KONFIGURACJA POSTSKALERA DLA UKŁADU WDT



Note: PSA and PS2:PS0 are bits in the OPTION_REG register.

Licznik nadzorcy, inaczej strażnik (WDT – Watchdog Timer) jest swobodnie działającym wewnętrznym oscylatorem RC, który nie potrzebuje żadnych komponentów zewnętrznych. Oscylator **RC układu WDT** jest **odseparowany** od oscylatora RC mikrokontrolera podłączonego do wyprowadzenia OSC1/CLKIN. Oznacza to, że licznik nadzorcy będzie działał nawet, jeżeli sygnał zegarowy na wyprowadzeniach OSC1 i OSC2 **zostanie zatrzymany**, np. przy wywołaniu instrukcji **SLEEP**. Układ WDT jest **włączany** i **wyłączany** za pomocą **rejestru konfiguracyjnego** mikrokontrolera przy **programowaniu** i nie można go wyłączyć programowo.

Licznik nadzorcy posiada nominalny **czas przepelnienia licznika 18 ms** (bez postskalera). Czas ten zależy od temp., nap. zasilania i wykonania konkretnego ukł. scalonego. Jeżeli wymagany dłuższy czas przepelnienia -> można zastosować postskaler ze stosunkiem podziału do 1:128.

Instrukcje CLRWDT oraz SLEEP – wyzer. licznika WDT oraz postskalera i zabezpieczają go przed przepelnieniem i generacją sygnału zerowania. Po przepelnieniu licznika WDT zer. jest bit !TO w rejestrze STATUS. Kiedy postskaler jest dołączony do licznika WDT, trzeba wykonać instrukcję CLRWDT przed zmianą współczynnika podziału postskalera, aby uniknąć resetu mikrokontrolera:

```
bsf      STATUS, RPO    ; wybierz bank 1
movlw    B'00001011'    ; podłącz postskaler do WDT ze współczynnikiem 1:8
clrwdt          ; zeruj układ WDT
movwf    OPTION_REG   ; zmień zawartość OPTION_REG z ustawieniami układu WDT
```

54.1. WYSWIETLACZ ALFANUMERYCZNY LCD. INFORMACJE OGÓLNE

Wyświetlacz ciekłokrystaliczny LCD (ang. Liquid Crystal Display) jest to rodzaj wyświetlacza, w którym wykorzystywane jest zjawisko skręcenia płaszczyzny polaryzacji przez ciekłe kryształy. Światło jak każda fala elektromagnetyczna może być spolaryzowane, tzn. drgania ośrodka mogą odbywać się w jednej wyróżnionej płaszczyźnie.

Elementem optycznym, który służy do polaryzowania światła jest filtr polaryzacyjny (polaroid). Niektóre substancje mają możliwość skręcania płaszczyzny polaryzacji. Jeżeli spolaryzowane światło zostanie przepuszczone przez polaroid o płaszczyźnie polaryzacji zgodnej z polaryzacją światła padającego, to zostanie ono przepuszczone. Jeżeli jednak płaszczyzna polaryzacji światła zostanie skręcona, to polaroid przepuści tylko tą część, która pozostała zgodna z jego płaszczyzną polaryzacji.

Ciekłe kryształy potrafią skrącać płaszczyznę polaryzacji światła, a dodatkowo kąt skręcenia może być ustalany za pomocą pola elektrycznego. Ta cecha została wykorzystana do budowy wyświetlacza LCD. Sterowanie zwykłym wyświetlaczem LCD jest dość skomplikowane, dlatego wielu producentów dostarcza układy sterowników, pozwalające na sterowanie wyświetlaczami za pomocą prostego interfejsu łączącego sterownik z mikrokontrolerem.

Alfanumeryczny wyświetlacz LCD oparty na kontrolerze HD44780 stał się w tej dziedzinie standardem przemysłowym. Alfanumeryczny wyświetlacz LCD oparty na HD44780 pozwala na wyświetlenie maksymalnie 4 wierszy po 20 znaków każdy. Znaki mogą pochodzić z wewnętrznego generatora znaków, jak również z generatora znaków użytkownika (maksymalnie 8 znaków). Prosty interfejs łączący wyświetlacz z mikrokontrolerem wymaga od 6 do 11 linii sygnałowych w zależności od wybranego wariantu. Dodatkowo wyświetlacz wymaga podania napięcia zasilającego oraz napięcia sterującego kontrastem wyświetlacza

54.2. INTERFEJS KOMUNIKACYJNY ZE STEROWNIKIEM HD44780. WARIANTY PODŁĄCZENIA WYSWIETLACZA DO MIKROKONTROLERA. PODŁĄCZENIE WYSWIETLACZA LCD DO MIKROKONTROLERA PIC16FX ZA POMOCĄ 4-BITOWEJ MAGISTRALI DANYCH I 2 SYGNAŁÓW STERUJĄCYCH E I RS

Komunikacja ze sterownikiem HD44780 odbywa się za pomocą dwukierunkowego interfejsu komunikacyjnego o poziomach TTL. Interfejs komunikacyjny składa się z następujących linii:

1. **D0-D7** – dwukierunkowa magistrala danych; w przypadku pracy 8-bitowej wykorzystywane są wszystkie sygnały, dla 4-bitowej starszy półbajt (D4-D7);
2. **RS (Register Select)** – sygnał wyboru odbiorcy przesyłanych danych: gdy RS = 1 komunikacja odbywa się z rejestrów danych sterownika, dla RS = 0 zapisywane dane trafiają do rejestru konfiguracyjnego sterownika, odczyt następuje z rejestru stanu sterownika;
3. **R/W (Read/Write)** – sygnał wyboru kierunku transmisji danych; dla R/W = 1 dane są odczytywane z wyświetlacza, gdy R/W = 0 dane są przesyłane do wyświetlacza;
4. **E (Enable)** – sygnał strobujący magistrali danych; wartość E = 1 oznacza, że na magistrali danych są dane gotowe do odebrania.

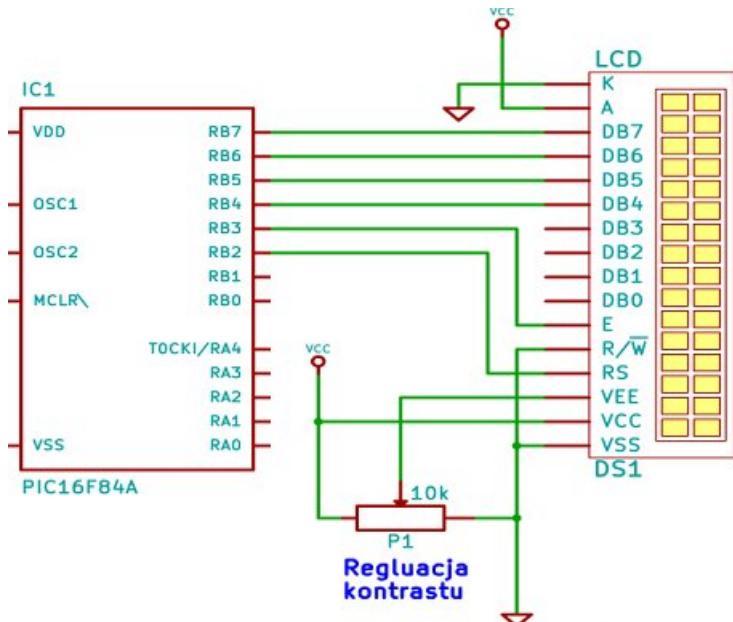
Istnieje kilka wariantów podłączenia wyświetlacza do mikrokontrolera:

- 8-bitowy interfejs danych wraz ze wszystkimi sygnałami sterującymi (11 linii sygnałowych),
- 8-bitowy interfejs danych z sygnałami E i RS (10 linii sygnałowych),
- 4-bitowy interfejs danych wraz ze wszystkimi sygnałami sterującymi (7 linii sygnałowych),
- 4-bitowy interfejs danych z sygnałami E i RS (6 linii sygnałowych).

Podłączenie wyświetlacza LCD do mikrokontrolera PIC16F84 za pomocą 4-bitowej magistrali danych i 2 sygnałów sterujących E i RS. Do poprawnej pracy wystarczy **6 linii: D4-D7** (interfejs 4-bitowy), **RS** oraz **E**. Sygnał **R/W** jest w takim przypadku na stałe podłączony do **logicznego „0”**, co oznacza, że **można** będzie wyłącznie **wysyłać** dane do sterownika wyświetlacza. W takiej konfiguracji **nie** jest **możliwe odczytywanie** danych z **wyświetlacza**, w tym m.in. sprawdzenie, czy wyświetlacz jest gotowy do przyjmowania kolejnych danych.

Dlatego wszystkie operacje na wyświetlaczu wymagają **opóźnień** zgodnych z **informacjami** zawartymi w dokumentacji **sterownika wyświetlacza**.

Przykład podłączenia wyświetlacza LCD do mikrokontrolera za pomocą 4-bitowej magistrali danych i 2 sygnałów sterujących E i RS przedstawiony został na rysunku. Wszystkie sygnały sterujące zostały podłączone do portu B



55.1. PAMIĘĆ STEROWNIKA HD44780. PAMIĘĆ DDRAM. PAMIĘĆ CGRAM

Sterownik HD44780 posiada trzy rodzaje pamięci: DDRAM, CGRAM i CGROM.

Pamięć DDRAM (ang. Display Data RAM) służy do przechowywania kodów znaków wyświetlanych na wyświetlaczu.

Pamięć CGRAM (ang. Character Generator RAM) przechowuje wzorce znaków definiowanych przez użytkownika.

Pamięć CGROM (ang. Character Generator ROM) przechowuje wzorce znaków wyświetlanych na wyświetlaczu (208 znaków w rozmiarze 5x8 pikseli oraz 32 znaki w rozmiarze 5x10 pikseli).

Pamięć **DDRAM** ma rozmiar 80 bajtów. Sposób odwzorowania pamięci DDRAM na znaki wyświetlane na wyświetlaczu zależy od kilku czynników: liczba wyświetlanych wierszy (1 lub 2), przesunięcie wyświetlania uzyskane za pomocą komendy „Cursor/Display Shift” lub podczas wpisywania danych przy ustawionym bicie S w komendzie „Entry Mode Set”.

Pamięć **CGRAM** posiada rozmiar 64 bajtów, które pozwalają na przechowywanie wzorców do 8 znaków o rozmiarze 5x8 pikseli lub 4 znaków o rozmiarze 5x10 pikseli. Znaki zdefiniowane przez użytkownika można wyświetlać używając kodów od 0 do 7 (5x8) lub od 0 do 3 (5x10) wysyłanych do pamięci DDRAM

55.2. REJESTRY STEROWNIKA HD44780. REJESTRY INSTRUKCJI IR, DANYCH DR, STANU SR. WYBÓR REJESTRU, Z KTÓRYM BĘDZIE NASTĘPOWAŁA KOMUNIKACJA

Sterownik HD44780 posiada trzy rejesty:

1. rejestr instrukcji IR (ang. Instruction Register),
2. rejestr danych DR (ang. Data Register) oraz
3. rejestr stanu SR (ang. Status Register).

Rejestr IR służy do konfiguracji i sterowania działaniem sterownika. Wysyłane są do niego kody instrukcji sterujących, takich jak instrukcja kasowania ekranu, czy przesunięcia wyświetlanego obszaru pamięci. Dane do rejestru IR mogą być wyłącznie zapisywane.

Rejestr DR służy do komunikacji z pamięciami RAM sterownika. Jeżeli do rejestru DR jest zapisywana wartość, sterownik przepisuje ją z rejestru DR do pamięci DDRAM lub CGRAM zależnie od bieżących ustawień. Jeżeli za pomocą rejestru IR ustawiany jest adres pamięci DDRAM lub CGRAM do bieżących operacji, sterownik automatycznie umieszcza w rejestrze DR zawartość komórki pamięci spod ustawionego adresu. Odczyt rejestru DR powoduje zapisanie w nim zawartości komórki pamięci spod kolejnego adresu, co pozwala odczytać sekwencyjnie całą pamięć sterownika.

Rejestr SR służy do sprawdzania stanu sterownika (rejestr tylko do odczytu). Znaczenie poszczególnych bitów rejestru przedstawiono poniżej: BF – flaga zajętości (ang. Busy Flag) sterownika wyświetlacza; gdy $BF = 1$, to sterownik wykonuje wewnętrzną operację i nie jest możliwa komunikacja z rejestrami IR i DR; gdy $BF = 0$, to sterownik jest gotowy do komunikacji z rejestrami IR i DR;

AC0-AC6 – licznik adresu (ang. Address Count), zawiera adres aktualnie zapisywanej komórki pamięci DDRAM lub CGRAM

| BF | AC6 | AC5 | AC4 | AC3 | AC2 | AC1 | AC0 |

Wybór rejestru, z którym będzie następowała komunikacja, jest realizowany poprzez ustawienie odpowiednich stanów na liniach RS i R/W.

56. KOMENDY STERUJĄCE. DODATKOWE PARAMETRY KOMEND. OPIS KOMEND STEROWNIKA HD44780

Sterownik HD44780 posiada 8 komend sterujących, których kody zapisywane są w rejestrze IR. Identyfikacja komendy odbywa się poprzez ustalenie pozycji najstarszej jedynki w kodzie komendy. Wiele komend posiada dodatkowe parametry, które dołączane są do komendy na bitach młodszych, niż bit identyfikujący komendę.

Komendy sterujące w skrócie (dla: RS=0, R/W=0)

| bity D7 ... D0 | Funkcja |
|-----------------------------|---|
| 0x01 | Clear display |
| 0x02 | Cursor Home |
| 0x04 I/D S | Entry Mode Set |
| 0x08 D C B | Włącz/wyłącz wyświetlacz / kurSOR / miganie |
| 0x10 S/C R/L - - | Przesunięcie wyświetlacza/kursora |
| 0x20 DL N F - - | Funkcje podstawowe |
| 0x40 { Adres CGRAM 6bit } | Ustawienie adresu CGRAM |
| 0x80 { Adres DDRAM 7bit } | Ustawienie adresu DDRAM |

I/D – zwiększa(1)/zmniejsza(0) adres DDRAM po wpisaniu znaku

S – włącz(1)/wyłącz(0) przesuwanie wyświetlacza przy zapisie znaku do DDRAM

–

D – włącz(1)/wyłącz(0) wyświetlacz

C – włącz(1)/wyłącz(0) kurSOR

B – włącz(1)/wyłącz(0) miganie znaku na pozycji kursora

–

S/C – przesuń wyświetlacz(1)/kursor(0)

R/L – o jeden znak w prawo(1)/lewo(0)

–

DL – magistrala danych 4 bity(0)/8 bitów(1)

N – jeden(0)/dwa(1) wiersze

F – 5x8(0)/5x10(1) czcionka

Zapis danej do CGRAM lub DDRAM: RS=1, R/W=0, D7...D0 = zapisywana dana.

Wybór aktywnej pamięci (i adresu) dokonuje się poprzez wysłanie komendy ustawienia adresu CGRAM lub DDRAM.

Komenda „Display Clear” (kod 0x01)

Powoduje wyczyszczenie wyświetlacza (cała pamięć DDRAM jest wypełniana kodem spacji - 0x20), powrót kurSORA pod adres 0x00 oraz przywraca wyświetlanie od adresu 0x00 (likwiduje przesunięcie realizowane komendą Display Shift).

Komenda „Return Home” (kod 0x02)

Powoduje powrót kurSORA pod adres 0x00 oraz przywraca wyświetlanie od adresu 0x00 (likwiduje przesunięcie realizowane komendą Display Shift).

Komenda „Set CGRAM” (kod 0x40)

Służy do ustawiania adresu pamięci CGRAM, pod który będą zapisywane dane. Po wykonaniu tej komendy każdorazowe wysłanie danej do wyświetlacza powoduje zapisanie jej w pamięci CGRAM. Po zapisaniu danej adres jest zwiększany lub zmniejszany w zależności od ostatnio ustawionej wartości bitu l/D komendy Entry Mode Set. Aby móc wpisywać dane do pamięci DDRAM należy wykonać komendę „Set DDRAM”.

Komenda „Set DDRAM” (kod 0x80)

Służy do ustawiania adresu pamięci DDRAM, pod który będą zapisywane dane. Pozwala ona na umieszczanie tekstu w dowolnym miejscu wyświetlacza.

57.1. ALGORYTM WYSYŁANIA DANYCH DLA INTERFEJSU 8-BITOWEGO. ALGORYTM WYSYŁANIA DANYCH DLA INTERFEJSU 4- BITOWEGO

Algorytm wysyłania danych dla interfejsu 8-bitowego

Wysyłanie danych do wyświetlacza w przypadku interfejsu 8-bitowego wymaga wykonania następujących czynności:

1. ustawienie zera logicznego na linii R/W; w przypadku, gdy korzysta się z uproszczonej wersji interfejsu linia ta jest zwarta do masy na stałe;
2. ustawienie odpowiedniej wartości na linii RS;
3. ustawienie właściwej wartości na liniach danych;
4. ustawienie na linii E logicznej jedynki, a następnie logicznego zera; Czas trwania logicznej jedynki i logicznego zera na linii E należy odczytać z dokumentacji sterownika (typowo jest to 0,5 µs dla każdego z poziomów logicznych).

Algorytm wysyłania danych dla interfejsu 4-bitowego

Wysyłanie danych do wyświetlacza w przypadku interfejsu 4-bitowego jest bardziej złożone i wymaga wykonania następujących czynności:

1. ustawienie zera logicznego na linii R/W; w przypadku, gdy korzysta się z uproszczonej wersji interfejsu linia ta jest zwarta do masy na stałe;
2. ustawienie odpowiedniej wartości na linii RS;
3. ustawienie na liniach danych wartości starszych czterech bitów wartości, która ma być wysłana do sterownika;
4. ustawienie na linii E logicznej jedynki, a następnie logicznego zera;
5. ustawienie na liniach danych wartości młodszych czterech bitów wartości, która ma być wysłana do sterownika;
6. ustawienie na linii E logicznej jedynki, a następnie logicznego zera;

Czas trwania logicznej jedynki i logicznego zera na linii E podczas wysyłania starszego i młodszego pół bajtu należy odczytać z dokumentacji sterownika (typowo jest to 0,5 µs dla każdego z poziomów logicznych).

57.2. INICJALIZACJA WYSWIETLACZA LCD. INICJALIZACJA WYSWIETLACZA LCD DLA INTERFEJSU 4-BITOWEGO. PROCEDURA INICJALIZACJI WYSWIETLACZA DLA INTERFEJSU 4-BITOWEGO

Po włączeniu zasilania sterownik wyświetlacza wykonuje procedurę inicjalizacji, po której jest gotowy do pracy. **Wewnętrzna procedura inicjalizacji** wykonuje następujące operacje:

1. Kasowanie ekranu (wyczyszczenie zawartości pamięci DDRAM).
2. Ustawienie funkcji: interfejs 8-bitowy, wyświetlanie w jednym wierszu, znaki 5x8.
3. Ustawienie trybu pracy: wyświetlacz wyłączony, kurSOR wyłączony, migotanie wyłączone.
4. Wyłączenie przesuwania, włączenie inkrementacji.

Inicjalizacja wyświetlacza dla interfejsu 4-bitowego

1. oczekanie min. 15 ms;
2. wysłanie komendy „Function Set” dla 8-bitowego interfejsu;
3. oczekanie min. 4.1 ms;
4. ponowne wysłanie komendy „Function Set” dla 8-bitowego interfejsu;
5. oczekanie min. 100 µs;
6. ponowne wysłanie komendy „Function Set” dla 8-bitowego interfejsu;
7. oczekanie min. 40 µs;
8. wysłanie komendy „Function Set” dla 4-bitowego interfejsu;
9. w tym miejscu interfejs jest nadal 8-bitowy, więc wysyłany jest tylko starszy półbajt; oczekanie min. 40 µs;
10. wysłanie komendy „Function Set” dla 4-bitowego interfejsu z jednoczesnym ustawieniem liczby wierszy i fontu;
11. oczekanie min. 40 µs;
12. wysłanie komendy „Display Off”;
13. oczekanie min. 40 µs;
14. wysłanie komendy „Display Clear”;
15. oczekanie min. 40 µs;
16. wysłanie komendy „Entry Mode Set” z odpowiednimi parametrami;
17. oczekanie min. 40 µs

58.1. PAMIĘĆ EEPROM I FLASH. INFORMACJE OGÓLNE. REJESTRY ADRESU I DANYCH

Mikrokontrolery PIC16 posiadają pamięć danych EEPROM oraz pamięć programu FLASH, które można czytać lub zapisywać podczas normalnej pracy mikrokontrolera. Pamięć EEPROM służy do przechowywania danych, natomiast pamięć FLASH jest pamięcią programu. Pamięci te nie są bezpośrednio mapowane w przestrzeni adresowej pamięci danych, natomiast są one adresowane pośrednio za pomocą rejestrów specjalnych SFR.

W mikrokontrolerach PIC16 używanych jest 6 rejestrów do obsługi pamięci EEPROM i FLASH: EECON1; EECON2; EEDATA; EEDATAH; EEADR; EEADRH; Mikrokontroler PIC16F84 nie umożliwia dostępu do pamięci programu FLASH podczas normalnej pracy mikrokontrolera. Dostęp do pamięci EEPROM w mikrokontrolerze PIC16F84 odbywa się w taki sam sposób jak dla mikrokontrolera PIC16F877A.

Mikrokontroler PIC16F877A posiada 256 8-bitowej pamięci EEPROM o adresach od 0 do FFh. Pamięć FLASH w mikrokontrolerze PIC16F877A ma rozmiar **8K słów**, gdzie każde **słowo** jest **14-bitowe**. Powoduje to konieczność użycia dwóch rejestrów danych **EEDATAH:EEDATA**.

Adres do pamięci FLASH jest **13 bitowy** i także wymaga użycia dwóch rejestrów **EEADRH:EEADR**. Pamięć EEPROM umożliwia odczyt i zapis pojedynczego bajta danych. Pamięć FLASH umożliwia odczyt tylko pojedynczego bajta danych, ale pozwala na zapis jednocześnie bloku 4 słów.

Podczas zapisu danych komórki pamięci EEPROM i FLASH są automatycznie kasowane. Rejestry adresu i danych: Mikrokontroler PIC16F877A posiada dwa rejesty adresu – EEADRH i EEADR. Za ich pomocą można zaadresować maksymalnie 256 bajtów danych pamięci EEPROM oraz 8K słów pamięci FLASH. Przy wyborze adresu pamięci danych EEPROM tylko mniej znaczący bajt adresu jest zapisywany do rejestru EEADR, a przy wyborze adresu pamięci programu FLASH dodatkowo bardziej znaczący bajt adresu zapisuje się do rejestru EEADRH.

Mikrokontroler PIC16F877A posiada dwa rejesty danych – EEDATAH i EEDATA. Za ich pomocą przesyła się dane do pamięci przy zapisie oraz odbiera się dane z pamięci przy odczycie. Przy odczycie pamięci danych EEPROM – 8-bitowe dane trafiają tylko do rejestru danych EEDATA, natomiast przy odczycie pamięci programu FLASH – 14-bitowe dane są umieszczone odpowiednio – starsza część słowa – w rejestrze EEDATAH, a młodsza część słowa – w rejestrze EEDATA.

Mikrokontroler PIC16F84 nie umożliwia dostępu do pamięci programu FLASH, dlatego wykorzystywane są tylko rejestr adresu EEADR do adresowania pamięci danych EEPROM oraz rejestr EEDATA do transmisji danych.

58.2. REJESTRY STERUJĄCE EECON1 I EECON2. ZNACZENIE POSZCZEGÓLNYCH BITÓW W REJESTRZE EECON1

Dostęp do pamięci EEPROM i FLASH jest kontrolowany za pomocą dwóch rejestrów konfiguracyjnych: EECON1 i EECON2. Podstawowym rejestrem konfiguracyjnym jest register EECON1. Rejestr **EECON2 nie jest fizycznym rejestrem**. W wyniku odczytania tego rejestrzu zawsze otrzymamy 0. Rejestr EECON2 jest używany przy zapisywaniu danych do pamięci EEPROM lub FLASH. Jednym z kroków operacji zapisu jest wpisanie do rejestrów **EECON2 wartości: 55h i AAh**.

EECON1 REGISTER (ADDRESS 18Ch)

| R/W-x | U-0 | U-0 | U-0 | R/W-x | R/W-0 | R/S-0 | R/S-0 |
|-------|-----|-----|-----|-------|-------|-------|-------|
| EEPGD | — | — | — | WRERR | WREN | WR | RD |

bit 7

bit 0

- bit 7 - **EEPGD**: bit umożliwiający wybór pamięci: 1 – pamięć programu FLASH, 0 – pamięć danych EEPROM;
- bit 3 - **WRERR**: flaga błędu zapisu do pamięci EEPROM/FLASH: 1 – zapis do pamięci EEPROM/FLASH został przerwany (przez zerowanie niskim poziomem na wejściu MCLR lub przez układ licznika nadzorcy WDT), 0 – operacja zapisu przebiegła pomyślnie;
- bit 2 - **WREN**: bit zezwalający na zapis do pamięci EEPROM/FLASH: 1 – zezwolenie na zapis, 0 – zapis zabroniony;
- bit 1 - **WR**: bit kontroli zapisu: 1 – inicjuje cykl zapisu, bit jest zerowany kiedy zapis zostanie zakończony, bit może być ustawiony na 1 tylko programowo, 0 – cykl zapisu do pamięci został zakończony;
- bit 0 - **RD**: bit kontroli odczytu: 1 – inicjuje cykl odczytu, bit jest zerowany kiedy odczyt zostanie zakończony, bit może być ustawiony na 1 tylko programowo, 0 – nie inicjuje cyklu odczytu.

59.1. ODCZYTYWANIE I ZAPISYWANIE DANYCH DO PAMIĘCI EEPROM

Zarówno mikrokontroler PIC16F84 jak i PIC16F877A posiada możliwość zapisu i odczytu pamięci EEPROM. Dla obu mikrokontrolerów operacja ta przebiega w identyczny sposób. Aby odczytać dane z pamięci EEPROM należy wykonać przedstawioną poniżej sekwencję kroków. Zapisać adres żądanej komórki pamięci do rejestru EEADR. Wyzerować bit EEPGD=0 w rejestrze EECON1. Ustawić bit RD=1 w rejestrze EECON1. Przeczytać dane z rejestru EEDATA.

; program odczytuje z pamięci EEPROM komórkę

; o adresie podanym w zmiennej ADRES

```
BSF    STATUS,RP1      ;
BCF    STATUS,RP0      ; przełącz się do banku 2
MOVF   ADRES,W        ; przenieś adres
MOVWF  EEADR          ; do rejestru EEADR
BSF    STATUS,RP0      ; przełącz się do banku 3
BCF    EECON1,EPPGD   ; wyzeruj bit EEPGD w EECON1 (EEPROM)
BSF    EECON1,RD       ; ustaw bit RD w EECON1 (odczyt)
BCF    STATUS,RP0      ; przełącz się do banku 2
MOVF   EEDATA,W       ; przenieś odczytane dane do rejestru W
```

Aby zapisać dane do pamięci EEPROM należy wykonać przedstawioną poniżej sekwencję kroków. Zapisać adres komórki pamięci, w której chcemy zachować dane do rejestru EEADR. Wpisać dane do rejestru EEDATA. Wyzerować bit EEPGD=0 w rejestrze EECON1. Ustawić bit WRN=1 w rejestrze EECON1. Wyłączyć przerwania (jeśli są włączone). Zapisać wartość 55h do rejestru EECON2 w dwóch krokach (najpierw do zapisujemy wartość 55h do rejestru W, potem zawartość rejestru W przenosimy do rejestru EECON2). Zapisać wartość AAh do rejestru EECON2 w taki sam sposób. Ustawić bit WR=1 w rejestrze EECON1. Włączyć przerwania. Oczekwać na przerwanie lub wyzerowanie bitu WR rejestrze EECON1. Wyzerować bit WRN=0 w rejestrze EECON1.

; program zapisuje do pamięci EEPROM zawartość zmiennej DANE do komórki o
; adresie podanym w zmiennej ADRES

```
banksel ADRES          ; przełącz na bank pamięci zmiennej ADRES
movf    ADRES, w         ; przenieś adres do w
banksel EEADR           ; przełącz na bank pamięci rejestru EEADR
movwf   EEADR            ; zapisz adres komórki EEPROM do odczytania
banksel DANE             ; przełącz na bank pamięci zmiennej DANE
movf    DANE, w           ; przenieś dane do w
banksel EEDATA           ; przełącz na bank pamięci rejestru EEDATA
movwf   EEDATA            ; zapisz dane do zapисania w rejestrze EEDATA
bcf     INTCON, GIE       ; wyłącz przerwania
```

```

banksel EECON1      ; przełącz się na bank pamięci rejestru EECON1
bcf     EECON1, EEPGD ; zeruj bit EEPGD w rejestrze EECON1 (zapis do EEPROM)
bsf     EECON1, WREN  ; ustaw zezwolenie na zapis

banksel EECON2      ; przełącz się na bank pamięci rejestru EECON2
movlw   0x55
movwf   EECON2      ; zapisz 55h do EECON2
movlw   0xAA
movwf   EECON2      ; zapisz AAh do EECON2

banksel EECON1      ; przełącz się na bank pamięci rejestru EECON1
bsf     EECON1, WR    ; ustaw bit WR aby zacząć zapis
bsf     INTCON, GIE   ; włącz przerwania

btfsr   EECON1, WR    ; poczekaj na zakończenie zapisu
goto   $-1             ;

bcf     EECON1, WREN  ; zablokuj zapisywanie

```

59.2. Odczytywanie i zapisywanie danych do pamięci FLASH (PIC16F877A)

Aby odczytać zawartość pamięci programu, użytkownik musi wpisać dwa bajty adresu do rejestrów EEADR i EEADRH, ustawić bit EEPGD rejestrze EECON1 oraz ustawić bit RD rejestrze EECON1.

Po ustawieniu bitu RD, kontroler pamięci flash zużywa dwa kolejne cykle rozkazowe na odczyt danych. Powoduje to zignorowanie dwóch instrukcji bezpośrednio występujących po: „bsf EECON1, RD”.

Dane dostępne są do odczytu w rejestrach EEDATA i EEDATH i znajdują się tam do czasu kolejnej operacji odczytu lub zapisu.

Odczyt pamięci programu

```

bsf     STATUS, RP1
bcf     STATUS, RPO  ; bank 2
movlw   H_ADRES
movwf   EEADRH       ; starszy bajt adresu do odczytu
movlw   L_ADRES
movwf   EEADR        ; młodszy bajt adresu do odczytu
bsf     STATUS, RPO  ; bank 3
bsf     EECON1, EEPGD ; EEPGD=1 odczyt danych z pamięci programu (FLASH)
bsf     EECON1, RD    ; rozpoczęcie odczytu pamięci FLASH

```

```

;
nop ; Dowolne instrukcje (muszą być dwie) są ignorowane
nop ; ponieważ pamięć programu jest wtedy odczytywana
;
bcf STATUS, RP0 ; bank 2
movf EEDATA, w ; W = młodszy bajt odczytanych danych
movwf L_DATA ; przenieś odczytany bajt do zmiennej L_DATA
movf EEDATH, w ; W = starszy bajt odczytanych danych
movwf H_DATA ; przenieś odczytany bajt do zmiennej H_DATA

```

Zapis pamięci programu

```

; Podana procedura zapisująca zakłada, że:
; 1. W ADDRH:ADDRL znajduje się prawidłowy początkowy adres
;    (najmniej znaczące dwa bity = '00')
; 2. Przygotowano 8 bajtów do zapisu począwszy od adresu DATADDR
; 3. ADDRH, ADDRL, i DATADDR są umieszczone w pamięci współdzielonej:
;    0x70 - 0x7F
;

bsf STATUS,RP1
bcf STATUS,RP0 ; bank 2
movf ADDRH,W ; załaduj początkowy adres pamięci FLASH
movwf EEADRH ; starszy bajt do rejestru EEADRH
movf ADDRL, W ;
movwf EEADR ; młodszy bajt do rejestru EEADR
movf DATADDR, W ; początkowy adres danych do zapisu
movwf FSR ; do rejestru wskaźnikowego

LOOP_WRITE:
    movf INDF, W ; załaduj pierwszy bajt do zapisania
    movwf EEDATA ; do młodszego bajtu słowa do zapisania
    incf FSR, F ; inkrementuj wskaźnik
    movf INDF, W ; załaduj drugi bajt do zapisania
    movwf EEDATH ; do starszego bajtu słowa do zapisania
    incf FSR, F ; inkrementuj wskaźnik
    bsf STATUS, RP0 ; bank 3
    bsf EECON1, EEPGD ; EEPGD=1, zapis do pamięci flash
    bsf EECON1, WREN ; zezwól na zapis
    bcf INTCON, GIE ; wyłącz przerwania (jeśli używane)
    movlw 0x55
    movwf EECON2 ; zapisz 55h do EECON2
    movlw 0xAA
    movwf EECON2 ; zapisz AAh do EECON2
    bsf EECON1, WR ; rozpoczęcie zapisu

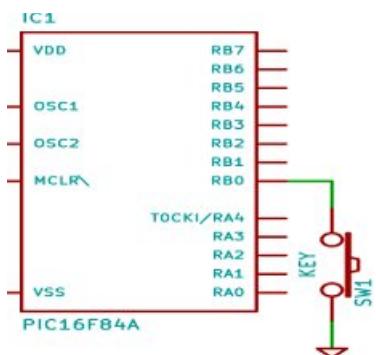
```

```

nop ; dowolne dwie instrukcje (muszą być dwie)
nop ; ignorowane przez procesor
bcf EECON1, WREN ; zablokuj zapis
bsf INTCON, GIE ; włącz przerwania (jeśli używane)
bcf STATUS, RP0 ; bank 2
incf EEADR, F ; inkrementuj adres komórki flash do zapisu
movwf EEADR, W ; Sprawdź czy najmłodsze dwa bity adresu = '00'
andlw 0x03 ; Wyzeruj wszystkie bity poza dwoma najmłodszymi w W
xorlw 0x00 ;
btfs STATUS, Z ; wyjdź jeśli zapisano już 4 słowa
goto LOOP_WRITE ; kontynuuj jeśli zapisano mniej niż cztery słowa

```

60. PRZYCISKI. ODCZYTYWANIE STANU PRZYCISKU. DRGANIE (ISKRZENIE) PRZYCISKÓW



Bardzo często istnieje potrzeba sterowania systemem wbudowanym za pomocą przycisku lub zestawu przycisków, a nawet klawiaturą. Przyciski najczęściej wykonywane są w postaci mikroprzełączników, które po naciśnięciu zwierają styki.

Ponieważ zwolniony przycisk stanowi rozwarcie, wejście mikrokontrolera należy wstępnie spolaryzować za pomocą rezystora podciągającego (ang. pull-up).

Może to być rezistor zewnętrzny o wartości $10\text{k}\Omega$ podłączony do źródła zasilania i wejścia mikrokontrolera, jak również wewnętrzny rezistor pull-up.

Mikrokontrolery PIC16 posiadają możliwośćłączenia rezystorów podciągających na liniach portu PORTB za pomocą wyzerowania ustawienia bitu RBPU w rejestrze OPTION_REG. Żaden inny port nie posiada wbudowanych rezystorów podciągających.

Odczytywanie wartości na wejściu mikrokontrolera wymaga wyłącznie sprawdzenia stanu odpowiedniego bitu rejestru PORTB. Należy zwrócić uwagę, że stan **naciśnięcia przycisku** sygnalizowany jest za pomocą **logicznego zera** (niskiego poziomu), natomiast **stan zwolnienia** przycisku za pomocą **logicznej jedynki** (wysokiego poziomu).

Odczytywanie stanu przycisku. Po naciśnięciu przycisku program przechodzi do części oznaczonej w komentarzu „**WCISNIĘTO**”, gdzie można wykonać niezbędne operacje. Następnie program oczekuje na zwolnienie przycisku. Po zwolnieniu przycisku program wraca do początku pętli.

```

MAIN:
    banksel OPTION_REG
    bcf      OPTION_REG, NOT_RBPU ; włączenie pull-up na PORTB
    banksel TRISB
    movlw   0xFF      ; przełączenie portu B na wejścia
    movwf   TRISB
    banksel PORTB

KB_READ
    btfsc  PORTB, RB0 ; czy naciśnięty przycisk?
    goto   $-1          ; nie - wróć o jeden wiersz w góre
                          ; NCIŚNIĘTO
    ; tutaj wykonujemy operacje po wciśnięciu przycisku
    btfss  PORTB, RB0 ; czy zwolniono przycisk?
    goto   $-1          ; nie - wróć o jeden wiersz w góre
    goto   KB_READ      ; pętla odczytywania stanu klawisza

END

```

Organie.: Każdy przycisk w trakcie naciskania przez pewien moment **generuje ciąg zer i jedynek**. Stan taki, zwany **drganiem (iskrzeniem)**, jest normalnym zjawiskiem wynikającym z fizycznej budowy przełącznika. Procedura sprawdzająca, czy przycisk został naciśnięty odczyta taki stan, jako wielokrotne naciśnięcie i zwolnienie przycisku. Stan iskrzenia trwa z reguły kilka milisekund.

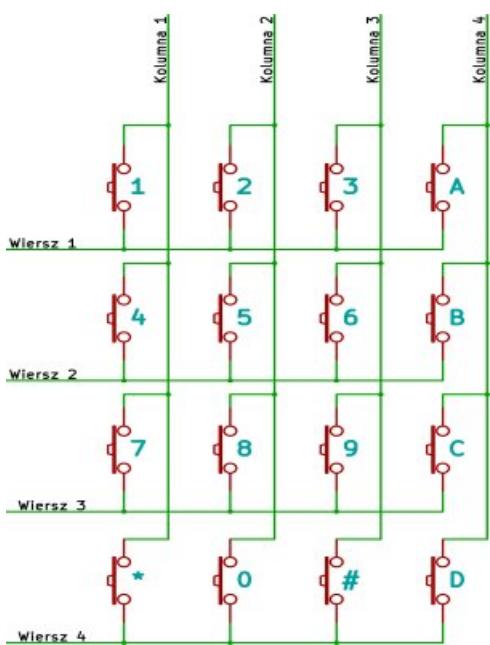
Aby zabezpieczyć się przed nieoprawnym odczytem stanu klawisza należy wprowadzić opóźnienia w trakcie odczytywania. W poniższym fragmencie kodu po **naciśnięciu przycisku** następuje **odczekanie przez 5 ms**, po którym iskrzenie praktycznie już nie występuje. Identyczna sytuacja ma miejsce przy zwalnianiu przycisku.

```

KB_READ
    btfsc  PORTB, RB0      ; czy naciśnięty przycisk?
    goto   $-1              ; nie - wróć o jeden wiersz w góre
    call   delay_5ms        ; odczekaj 5 ms - likwidacja iskrzenia
    ; tutaj wykonujemy operacje po wciśnięciu przycisku
    btfss  PORTB, RB0      ; czy zwolniono przycisk?
    goto   $-1              ; nie - wróć o jeden wiersz w góre
    call   delay_5ms        ; odczekaj 5 ms - likwidacja iskrzenia
    goto   KB_READ          ; pętla odczytywania stanu klawisza

```

61. KLAWIATURA MARYCOWA. PROCEDURA OBSŁUGI KLAWIATURY MARYCOWEJ 4X4



Zwiększenie liczby przycisków wymaga zwiększenia liczby wyprowadzeń mikrokontrolera zajętych przez dołączone przyciski. Sytuacja taka powoduje znaczne zredukowanie liczby wyprowadzeń, które pozostają do wykorzystania przez inne układy.

Załóżmy, że system wymaga podłączenia 4 przycisków. Można to zrealizować analogicznie, jak zostało pokazane w powyższym przykładzie, jedynie przez dołączenie kolejnych przycisków do kolejnych wyprowadzeń mikrokontrolera. Następnie chcemy zwiększyć liczbę przycisków o kolejne 4. Gdyby podłączyć je do tych samych wyprowadzeń, co poprzednie przyciski, nie mielibyśmy możliwości rozróżnienia, który przycisk został wciśnięty.

Można to jednak zrobić dodając dwa dodatkowe sygnały służące do aktywacji odczytu jednego z dwóch zestawów przycisków. Jeśli teraz dodamy jeszcze dwa zestawy po 4 przyciski każdy to uzyskamy układ 16 klawiszy, który może być odczytywany za pomocą 8 (zamiast 16) linii mikrokontrolera (4 linie na aktywację każdego z zestawów oraz 4 linie do odczytu stanu przycisku w aktywnym zestawie). Układ taki nazywany jest klawiaturą matrycową.

Klawiatura podłączona jest do portu C w taki sposób, że do wyprowadzeń:

RC7, RC5, RC3, RC1 – podłączono wiersze klawiatury,

RC6, RC4, RC2, RC0 – kolumny.

Do odczytu stanu klawiatury służy procedura ReadKey. Na początku następuje aktywowanie odczytu stanu kolumny pierwszej (za pomocą makrodefinicji ACT_COL z parametrem B_COL1).

dla każdej kolumny k

 podaj zero logiczne na port kolumny k

 dla każdego wiersza w

 jeżeli na porcie wiersza w jest zero logiczne to

 klawisz wxk jest wciśnięty

 return

 end

end

żaden klawisz nie wciśnięty

return

Aktywacja polega na ustawieniu trybu wyprowadzenia połączonego z kolumną 1 na wyjście i ustawienie na nim niskiego poziomu. Następnie wywoływana jest procedura TST_ROW, która odczytuje z portu B jaki klawisz został naciśnięty i zamienia numer wiersza, w którym został naciśnięty klawisz na kod od 0x80 do 0x83.

W przypadku, gdy nie został naciśnięty żaden przycisk rejestr W zawiera wartość 0. Gdy naciśnięty został przycisk, to po odczytaniu jego kodu procedurą TST_ROW następuje zamiana (od etykiety RK_press) kodu klawisza na kod ASCII znaku, który został naciśnięty. Gdy w danej kolumnie nie został naciśnięty przycisk następuje sprawdzenie kolejnych kolumn. Gdy żaden przycisk nie został naciśnięty w rejestrze W jest zapisywana wartość 0xFF i następuje opuszczenie procedury.

; Program obsługi klawiatury matrycowej

```
KB_PORT equ PORTC ; port, do którego podłączona
KB_TRIS equ TRISC ; jest klawiatura
KB_ROW1 equ RC7 ; pierwszy wiersz
KB_ROW2 equ RC5 ; drugi wiersz
KB_ROW3 equ RC3 ; trzeci wiersz
KB_ROW4 equ RC1 ; czwarty wiersz
KB_COL1 equ RC6 ; pierwszą kolumnę
KB_COL2 equ RC4 ; drugą kolumnę
KB_COL3 equ RC2 ; trzecią kolumnę
KB_COL4 equ RC0 ; czwartą kolumnę
```

; Makro aktywujące odczyt naciśnięcia klawisza w wierszu 'ROW'

```
ACT_COL MACRO COL
    bsf STATUS, RP0 ; przejdź do banku 1
    movlw ~(1 << COL) ; ustaw tryb pracy wyprowadzenia;
    movwf KB_TRIS ; odpowiada za wybór kolumny na wyjście
    bcf STATUS, RP0 ; powrót do banku 0
    movwf KB_PDR ; ustaw na wyprowadzenie wyboru kol. stan niski
    goto $+1 ; dla ustalenia stanu wyjścia oczekaj 2 cykle
ENDM
RST CODE 0x000 ; wektor resetu procesora
    pagesel main ; wybór strony pamięci programu
    goto main ; skok do początku programu
PGM CODE
main
    banksel OPTION_REG
    bcf OPTION_REG, NOT_RBPU ; włączenie pull-up na PORTB
    banksel PORTA
    call readKey
    goto $ ; pętla bez końca
```

```

; Procedura odczytująca klawisz. Zwraca w W kod ASCII klawisza
readKey
    ACT_COL    KB_COL1      ; aktywuj kolumnę 1
    call       TST_ROW      ; zamień nr wiersza na kod 0x80-0x83
    iorlw     0x00          ; sprawdź, czy nie 0 (żaden nie wciśnięty)
    btfss     STATUS, Z    ;
    goto      RK_press      ; Z=0, W != 0, wciśnięto - zakończ testowanie
                           ; Z=1, W = 0,  nic nie wciśnięto - kolejna kolumna

RK_COL_2
    ACT_COL    KB_COL2      ; aktywuj kolumnę 2
    call       TST_ROW      ; zamień nr wiersza na kod 0x80-0x83
    iorlw     0x00          ; czy 0 (żaden nie wciśnięty)
    btfsc     STATUS, Z    ; nie - przeskocz
    goto      RK_COL_3      ; Z=1, W = 0,  nic nie wciśnięto - kolejna kolumna
    iorlw     B'00000100'   ; Z=0, W != 0, wciśnięto - dołącz numer kolumny
    goto      RK_press      ; i zakończ testowanie

RK_COL_3
    ACT_COL    KB_COL3      ; aktywuj kolumnę 3
    call       TST_ROW      ; zamień nr wiersza na kod 0x80-0x83
    iorlw     0x00          ; czy nie 0 (żaden nie wciśnięty)
    btfsc     STATUS, Z    ; nie - przeskocz
    goto      RK_COL_4      ; Z=1, W = 0,  nic nie wciśnięto - kolejna kolumna
    iorlw     B'00001000'   ; Z=0, W != 0, wciśnięto - dołącz nr kolumny
    goto      RK_press      ; i zakończ testowanie

RK_COL_4
    ACT_COL    KB_COL4      ; aktywuj kolumnę 4
    call       TST_ROW      ; zamień nr wiersza na kod 0x80-0x83
    iorlw     0x00          ; sprawdź czy nie 0 (żaden nie wciśnięty)
    btfsc     STATUS, Z    ; nie - przeskocz
    goto      RK_COL_STOP   ; Z=1, W = 0,  nic nie wciśnięto - koniec
    iorlw     B'00001100'   ; Z=0, W != 0, wciśnięto - dołącz numer kolumny
    goto      RK_press      ; i zakończ testowanie

RK_COL_STOP
    movlw     0xFF          ; nic nie naciśnięto
return
RK_press
    andlw     B'01111111'
    addwf     PCL, F        ; zamień na kod ASCII
    dt       "147*2580369#ABCD"
; Procedura spr., w którym wierszu klawiatury został wciśnięty klawisz
; Zwraca w 'W' numer wiersza
TST_ROW
    clrw

```

```

btfss KB_PORT, KB_ROW1 ; czy naciśnięto w wierszu 1?
movlw B'10000000' ; kod - 0x80
btfss KB_PORT, KB_ROW2 ; czy naciśnięto w wierszu 2?
movlw B'10000001' ; kod - 0x81
btfss KB_PORT, KB_ROW3 ; czy naciśnięto w wierszu 3?
movlw B'10000010' ; kod - 0x82
btfss KB_PORT, KB_ROW4 ; czy naciśnięto w wierszu 4?
movlw B'10000011' ; kod - 0x83
return

```

END

62.1. DOŁĄCZENIE KLAWIATURY KOMPUTERA PC DO MIKROKONTROLERA. PRZESYŁANIE DANYCH DO KLAWIATURY. PRZESYŁANIE DANYCH Z KLAWIATURY DO MIKROKONTROLERA

Klawiatura jest elementem często stosowanym w układach sterowania. Jednak styki klawiatury są elementami mechanicznymi dlatego styki podczas przełączania drgają, powodując powstanie serii impulsów. Drgania te mogą trwać od 100 us do 15 ms. Drganie musi być eliminowane tak, aby po naciśnięciu klawisza mógł być odczytany jego stabilny stan.

1. Algorytm obsługi klawiatury
2. Czekanie na puszczenie przycisku
3. Czekanie na przyciśnięcie przycisku
4. Odliczanie opóźnienia eliminującego drgania styków (ok. 20 ms)
5. Sprawdzenie stanu klawiatury. Jeżeli żaden przycisk nie jest wciśnięty, to powrót do punktu 2 (wykryto zakłócenie). Inaczej (wykryto wciśnięcie) generowanie kodu przyciśniętego klawisza.
6. Czekanie na puszczenie przycisku.

```

; Obsługa klawiatury PC
; Przyciski podłączone do linii RB4...RB7
; kod przycisku jest zwracany w rejestrze W
pom equ 0x000C ; zmienna pomocnicza
_klawiatura ; 1. Czekanie na puszczenie przycisku
    movf PORTB,W
    andlw 0xf0 ; znalezienie przyciśniętego klawisza
    xorlw 0xf0 ; inwersja wyniku
    btfss STATUS,Z ; jeżeli zero, to nie ma naciśniętych klawiszy
    goto _klawiatura
p_klawisz ; 2. Czekanie na przyciśnięcie klawisza
    movf PORTB, W

```

```

andlw 0x0
xorlw 0x0
btfsc STATUS, Z      ; jeżeli zero, klawisz naciśnięty
goto p_klawisz ;
; 3. Odliczanie opóźnienia eliminującego drgania styków (ok. 20 ms).
movlw .20
call delay_msek ; likwidacja drgań styków
; 4. Ponowne sprawdzenie klawiatury
movlw PORTB,W
andlw 0x0
xorlw 0x0
btfsc STATUS,Z      ; jeżeli zero, klawisz naciśnięty
goto p_klawisz ; czekaj na przyciśnięcie klawisza
; 5.1. generowanie kodu przyciśniętego klawisza
movf PORTB,W        ; pobranie wartości portu B
movwf pom            ; zachowujemy w pom
rrf    pom,f          ; przesunięcie w prawo o 4 bity
rrf    pom,f
rrf    pom,f
rrf    pom,f
comf  pom,f          ; negacja wyniku
; 5. Czekanie na przyciśnięcie klawisza
z_klawisz
    movf PORTB,W
    andlw 0x0
    xorlw 0x0
    btfss STATUS,Z      ; nie ma naciśniętych klawisz
    goto z_klawisz
; 6. Ładowanie wyniku do akumulatora
    movf pom,w
    andlw 0x0f           ; czyść w W starsze bity
return

; przykład zastosowania procedury odczytu stanu klawiatury
org 00h
banksel TRISB ; inicjalizacja portów
movlw TRISB
clrf  TRISA
banksel PORTA
clrf  PORTA
_et
    call _klawiatura ; czytanie klawiatury
    movwf PORTA
    goto _et

```

62.2. PROCEDURA ODCZYTANIA KODU PRZYCISKU KLAWIATURY PC

Przy wysyłaniu danych do mikrokontrolera sterownik klawiatury generuje sygnał zegarowy i dane pojawiają się synchronicznie na linii Data. Dane na linii Data muszą być stabilne w momencie opadającego zbocza na linii zegarowej Clock.

Sterownik może wysłać kod sterujący klawiatury, a klawiatura odsyła kod potwierdzenia. Po naciśnięciu przycisku klawiatura wysyła do sterownika kod przyciśniętego klawisza. Jeżeli przycisk jest cały czas trzymany, to po określonym czasie wysyłany jest ponownie jego kod, aż do momentu puszczenia klawisza.

Po puszczeniu klawisza wysyłany jest kod F0h. Jeżeli nie zależy nam na wysyłaniu poleceń do klawiatury, to można nie wysyłać do niej żadnych komend. Kody podstawowych klawiszy mają zawsze długość jednego bajtu, a każdy klawisz ma przypisany tylko jeden kod.

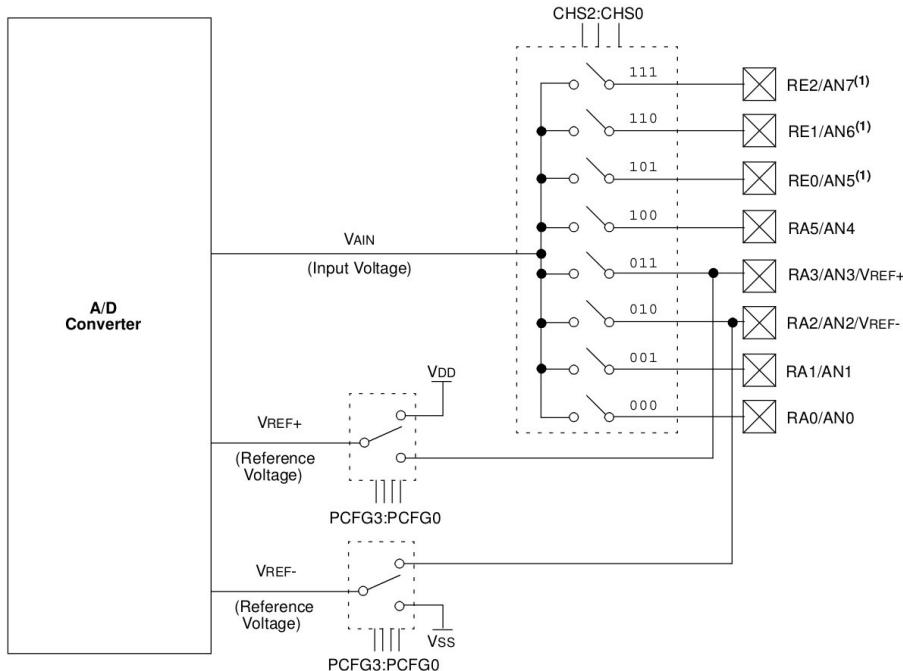
Rozróżnienie pomiędzy dużymi i małymi literami musi być realizowane programowe za pomocą sprawdzenia, czy przed naciśnięciem klawisza był przyciśnięty klawisz Shift lub Caps Lock.

Kontroler klawiatury obsługuje także przyciski z numerycznej i kursorowej części klawiatury. Kody tych przycisków mają długość 2 bajtów, z wyjątkiem trzech klawiszy: PrtScr, ScrLck i Break.

Po odebraniu kodu prefiksu E0h trzeba odebrać jeszcze jeden kod i dopiero te dwie wartości określają jednoznacznie wcisnięty przycisk.

Procedura odczytuje z klawiatury 11 bitów ramki wysyłanej z klawiatury i umieszcza je w rejestrach kod_0 i kod_1. Linia danych Data jest podłączona do wyprowadzenia RA2 mikrokontrolera, a linia zegarowa Clock – do wyprowadzenia RA3. Po odebraniu wszystkich jedenastu bitów są sprawdzane bity startu i stopu. Jeżeli bity ramki są prawidłowe, to do rejestru W wpisywany jest odebrany kod klawisza. W przeciwnym przypadku rejestr W jest zerowany.

63. Przetwornik analogowo-cyfrowy. Parametry przetwornika A/C. Schemat blokowy przetwornika A/C. Wynik przetwarzania



Note 1: Not available on 28-pin devices.

Układ przetwornika analogowo cyfrowego występuje tylko w mikrokontrolerach PIC16F87xA, nie występuje w PIC16F8x. Dla PIC16F877A przetwornik posiada **8 wejść analogowych**.

Wynikiem konwersji wejściowego sygnału analogowego jest 10-bitowa liczba binarna. Przetwornik ADC posiada wejścia napięcia odniesienia zarówno dla poziomu wysokiego jak i niskiego. Są one wybierane dla pewnych kombinacji V_{dd} , V_{ss} , wejść RA2 i RA3.

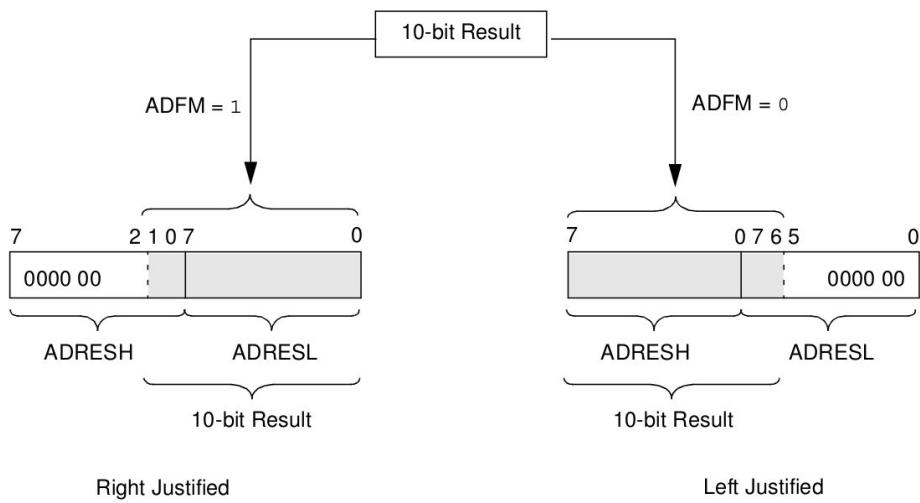
Przetwornik układów PIC16 posiada **unikatową cechę** możliwości pracy gdy mikrokontroler jest **uspiony**. W tym celu musi być taktowany **własnym wewnętrznym oscylatorem RC**.

Moduł przetwornika ADC operuje na **4 rejestrach**:

1. **ADRESH** - rejestr górnego bajtu wyniku
2. **ADRESL** - rejestr dolnego bajtu wyniku
3. **ADCON0** - rejestr kontrolny 0
4. **ADCON1** - rejestr kontrolny 1

Wynik przetwarzania

10-bitowy wynik przetwarzania umieszczany jest w dwóch rejestrach 8-bitowych. Istnieją dwie możliwości zapisu wyniku do tych rejestrów.



- wyjustowanie do prawej strony** - ADCON1<ADFM>=1, dwa najstarsze bity wyniku w ADRESH, młodszy bajt w ADRESL, używamy tego trybu jeśli mierzone wartości napięcia są relatywnie małe, a zależy nam na większej dokładności. Możemy wówczas odczytać tylko wartość z ADRESL, a dwa starsze bity w ADRESH pominąć.
- wyjustowanie do lewej strony** - ADCON1<ADFM>=0, starszy bajt wyniku umieszczany w ADRESH, najmłodsze dwa bity w ADRESL. Używamy go gdy nie zależy nam na tak bardzo na dokładności, lub gdy wynik pomiaru jest niestabilny. Możemy wówczas odczytać tylko wartość w ADRESH pomijając 2 młodsze bity w ADRESL

Oczywiście można też odczytać pełny 10-bitowy wynik jednak jego przetworzenie będzie już wymagało zazwyczaj napisania procedur operujących na liczbach 16-bitowych, gdyż mikrokontroler PIC16 nie posiada takich rozkazów.

64.1. Rejestry ADCON0 i ADCON1

Rejestr konfiguracyjny ADCON0

| R/W-0 ADCS1 | R/W-0 ADCS0 | R/W-0 CHS2 | R/W-0 CHS1 | R/W-0 CHS0 | R/W-0 GO/DONE | U-0 — | R/W-0 ADON |
|----------------|----------------|---------------|---------------|---------------|------------------|----------|---------------|
| bit 7 | | | | | | | bit 0 |

- bit 7-6: ADCS1:ADCS0: Wybór zegara taktującego układ przetwornika wspólnie z ADCON1<ADCS2>

| ADCON1<ADCS2> | ADCON0<ADCS1:ADCS0> | Częstotliwość |
|---------------|---------------------|--|
| 0 | 00 | $F_{osc}/2$ |
| 0 | 01 | $F_{osc}/8$ |
| 0 | 10 | $F_{osc}/32$ |
| 0 | 11 | F_{RC} (wewnętrzny oscylator układu ADC) |
| 1 | 00 | $F_{osc}/4$ |
| 1 | 01 | $F_{osc}/16$ |
| 1 | 10 | $F_{osc}/64$ |
| 1 | 11 | F_{RC} (wewnętrzny oscylator układu ADC) |

- bity 5-3: CHS2:CHS0: Wybór wejściowego kanału analogowego
 - 000 – kanał 0 (AN0)
 - 001 – kanał 1 (AN1)
 - 010 – kanał 2 (AN2)
 - 011 – kanał 3 (AN3)
 - 100 – kanał 4 (AN4)
 - 101 – kanał 5 (AN5)
 - 110 – kanał 6 (AN6)
 - 111 – kanał 7 (AN7)
- bit 2 – GO/DONE: Status konwersji ADC. Istotny gdy **ADON=1**
 - 1 – konwersja aktualnie trwa, ustawienie tego bitu rozpoczyna konwersje, bit jest automatycznie czyszczony przez układ gdy konwersja się kończy
 - 0 – brak aktywnej konwersji
- bit 0: ADON
 - 1 – układ ADC jest włączony
 - 0 – układ ADC wyłączony i nie pobiera żadnej mocy

Rejestr konfiguracyjny ADCON1

| R/W-0 | R/W-0 | U-0 | U-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 |
|-------|-------|-----|-----|-------|-------|-------|-------|
| ADFM | ADCS2 | — | — | PCFG3 | PCFG2 | PCFG1 | PCFG0 |
| bit 7 | | | | bit 0 | | | |

- bit 7 – ADFM – sposób zapisu wyniku konwersji
 - 0 – wyjustowanie do lewej strony. Sześć bardziej znaczących bitów w ADRESL odczytywane jako 0
 - 1 – wyjustowanie do prawej strony. Sześć bardziej znaczących bitów w ADRESH odczytywane jako 0
- bit 6 – ADCS2: Wybór zegara taktującego układ przetwornika wspólnie z ADCON0<ADCS1:ADCS0> (patrz: tabela na poprzedniej stronie)
- bity 3-0 – PCFG3:PCFG0 – konfiguracja portów wejściowych przetwornika

64.2. Skonfigurowanie przetwornika A/C. Określenie trybu pracy wejść

| PCFG <3:0> | AN7 | AN6 | AN5 | AN4 | AN3 | AN2 | AN1 | AN0 | VREF+ | VREF- | C/R |
|---------------|-----|-----|-----|-----|-------|-------|-----|-----|-------|-------|-----|
| 0000 | A | A | A | A | A | A | A | A | VDD | VSS | 8/0 |
| 0001 | A | A | A | A | VREF+ | A | A | A | AN3 | VSS | 7/1 |
| 0010 | D | D | D | A | A | A | A | A | VDD | VSS | 5/0 |
| 0011 | D | D | D | A | VREF+ | A | A | A | AN3 | VSS | 4/1 |
| 0100 | D | D | D | D | A | D | A | A | VDD | VSS | 3/0 |
| 0101 | D | D | D | D | VREF+ | D | A | A | AN3 | VSS | 2/1 |
| 011x | D | D | D | D | D | D | D | D | — | — | 0/0 |
| 1000 | A | A | A | A | VREF+ | VREF- | A | A | AN3 | AN2 | 6/2 |
| 1001 | D | D | A | A | A | A | A | A | VDD | VSS | 6/0 |
| 1010 | D | D | A | A | VREF+ | A | A | A | AN3 | VSS | 5/1 |
| 1011 | D | D | A | A | VREF+ | VREF- | A | A | AN3 | AN2 | 4/2 |
| 1100 | D | D | D | A | VREF+ | VREF- | A | A | AN3 | AN2 | 3/2 |
| 1101 | D | D | D | D | VREF+ | VREF- | A | A | AN3 | AN2 | 2/2 |
| 1110 | D | D | D | D | D | D | D | A | VDD | VSS | 1/0 |
| 1111 | D | D | D | D | VREF+ | VREF- | D | A | AN3 | AN2 | 1/2 |

A – wejście analogowe D – cyfrowe wejście/wyjście

C/R – # wejściowych kanałów analogowych / # napięć odniesienia dla ADC

64.3. Częstotliwość taktowania układu przetwornika A/C. Wybór wejścia, które zostanie poddane przetwarzaniu

Czas konwersji pojedynczego bitu oznaczany jest jako T_{AD} . Dla rozdzielczości 10-bitowej minimalny czas konwersji wynosi $12 T_{AD}$. Źródło taktowania przetwornika ADC jest wybierane programowo. Istnieje siedem możliwych wartości T_{AD} :

- $2 T_{osc}, 4 T_{osc}, 8 T_{osc}, 16 T_{osc}, 32 T_{osc}, 64 T_{osc}$
- przy użyciu wewnętrznego oscylatora RC układu ADC: $2 - 6 \mu s$

Dla uzyskania poprawnego wyniku konwersji czas T_{AD} powinien wynieść minimum $1.6 \mu s$.

TABLE 11-1: TAD vs. MAXIMUM DEVICE OPERATING FREQUENCIES (STANDARD DEVICES (F))

| AD Clock Source (TAD) | | Maximum Device Frequency T_{osc} |
|-------------------------|-------------------|---------------------------------------|
| Operation | ADCS2:ADCS1:ADCS0 | |
| 2 Tosc | 000 | 1.25 MHz |
| 4 Tosc | 100 | 2.5 MHz |
| 8 Tosc | 001 | 5 MHz |
| 16 Tosc | 101 | 10 MHz |
| 32 Tosc | 010 | 20 MHz |
| 64 Tosc | 110 | 20 MHz |
| RC ^(1, 2, 3) | x11 | (Note 1) |

Note 1: The RC source has a typical TAD time of $4 \mu s$ but can vary between $2-6 \mu s$.

- 2: When the device frequencies are greater than 1 MHz, the RC A/D conversion clock source is only recommended for Sleep operation.
- 3: For extended voltage devices (LF), please refer to **Section 17.0 “Electrical Characteristics”**.

Konfiguracja analogowych wejść portów

Rejestry ADCON1 oraz rejesty TRISx kontrolują działanie wejść analogowych. Wejścia analogowe powinny mieć ustawione odpowiednie bity TRISx na 1 (tryb wejściowy). Jeżeli port jest w trybie wyjściowym konwersji zostanie poddany wyjściowy stan logiczny na tym porcie (V_{OH} lub V_{OL})

Przy próbie programowego **odczytania** stanu pinów **analogowych** portów zostanie **zwrócone zawsze 0**.

Podanie napięcia **analogowego** na wejścia skonfigurowane jako **cyfrowe** może spowodować, że przez wejściowe bufory przepłynie **prąd który przekracza dopuszczalny**.

```

/* Przykład konfiguracji przetwornika ADC */
; Częstotliwość taktowania ADC: Fosc/8
; wejście analogowe RA0/ANO
; GO/DONE = 0 (nie przetwarza)
; ADON=1, przetwornik włączony
banksel  ADCONO
movlw    b'01000001'
movwf    ADCONO

; Częstotliwość taktowania ADC: Fosc/8
; ADFM=0, starsza część wyniku w ADRESH, młodsze dwa bity w ADRESL
; PCFG3...PCFG0 = 1110 - RA7...RA1 - cyfrowe, RA0 - analogowy,
; standardowe napięcia odniesienia (Vdd i Vss)
banksel  ADCON1
movlw    b'00001110'
movwf    ADCON1

; porty RA7...RA1 w trybie wyjściowym (cyfrowe)
; port RA0 w trybie wejściowym (analogowy)
banksel  TRISA
movlw    b'00000001'
movwf    TRISA

; Konfiguracja przerwań
banksel  PIE1
bsf      PIE1, ADIE          ; włączenie przerwania od przetwornika ADC
banksel  PIR1
bcf      PIR1, ADIF          ; wyczyszczenie flagi wystąpienia przerwania ADC

banksel  PORTA              ; przełączenie na bank 0
bsf      INTCON, PEIE        ; zezwolenie na przerwania od urządzeń peryferyjnych
bsf      INTCON, GIE         ; globalne zezwolenie na przerwania

read_loop:
; uruchomienie konwersji i oczekiwanie na jej zakończenie
banksel  ADCONO
bsf      ADCONO, GO_DONE     ; ustawienie bitu GO/DONE
btfsr  ADCONO, GO_DONE
goto   $-1                   ; czekaj dopóki GO_DONE=1
banksel  PIR1
bcf      PIR1, ADIF          ; wyczyszczenie flagi wystąpienia przerwania ADC
banksel  ADRESH
movf    ADRESH, w            ; przenieś górny bajt wyniku do W
banksel  PORTA

```

65.1. UKŁADY LOGIKI PROGRAMOWALNEJ. ZALETY LOGIKI PROGRAMOWALNEJ. CECHY SYSTEMÓW WBUDOWANYCH NA UKŁADACH PROGRAMOWALNYCH

Ostatnio do budowy szybkich systemów wbudowanych najczęściej wykorzystuje się układy logiki programowalnej: FPGA, CPLD i SoPC. Dzięki dużej mocy funkcjonalnej i dużej liczbie wyprowadzeń zewnętrznych, układy te pozwalają, drogą konfiguracji swojej struktury wewnętrznej, realizować na jednym układzie scalonym całe systemy mikroprocesorowe z dużą liczbą urządzeń peryferyjnych.

Inną właściwością układów programowalnych jest to, że pozwalają one rozwiązywać specjalne zadania systemów wbudowanych nie programowo, za pomocą wykonywania szeregu rozkazów, a sprzętowo. Inaczej mówiąc, specjalne zadania przedstawiane są w postaci układu logicznego, który jest realizowany sprzętowo w układzie programowalnym. Przy czym, szybkość rozwiązania zadania, w porównaniu z mikroprocesorami i mikrokontrolerami, jest kilka rzędów wyższa.
ZALETY:

Inną ważną cechą układów logiki programowalnej jest duża liczba wyprowadzeń zewnętrznych ogólnego przeznaczenia. Dzięki temu w wielu przypadkach można zrezygnować z multipleksowania sygnałów i wykorzystywania różnych standardowych interfejsów.

Oprócz tego, układy logiki programowalnej wyróżniają się niskim poborem mocy, małymi gabarytami i ciężarem.

Ostatnio pojawiła się tendencja do realizacji w układzie scalonym, razem z logiką programowalną, różnych urządzeń peryferyjnych, często wykorzystywanych przy budowie systemów wbudowanych:

- bloków pętli sprzężenia fazowego (PLL),
- przetworników ADC i DAC,
- komparatorów sygnałów analogowych,
- bloków mnożących itp.

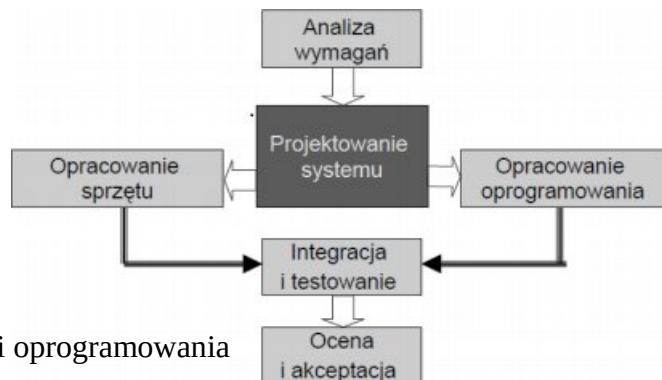
65.2. POZIOMY PROJEKTOWANIA SYSTEMÓW WBUDOWANYCH. METODYKA PROJEKTOWANIA SYSTEMÓW WBUDOWANYCH. SYNTEZA CZĘŚCI SPRZĘTOWEJ. KOMPILACJA CZĘŚCI PROGRAMOWEJ

Projektowanie niezawodnych systemów sterujących opartych na mikroprocesorach jako urządzeniach zastępujących klasyczne regulatory wymaga **systematycznego podejścia** do procesu projektowania. Charakterystyczną cechą procesu projektowania komputerowych systemów sterujących jest możliwość **równoległego opracowania** części **sprzętowej i programowej**.

1. Analiza wymagań

Pierwszy kluczowy etap powstawania projektu

- podstawowe funkcje
- żądana dokładność
- szybkość i niezawodność działania
- ograniczenia dot. stosowanego sprzętu i oprogramowania
- szacowane koszty projektu
- spodziewane efekty wdrożenia



Dokument: specyfikacja wymagań

Etap kończy: opinia recenzentów.

2. Projektowanie systemu

- określenie architektury
- podział na podsystemy
- przypisanie funkcji poszczególnym podsystemom
- zdefiniowanie reguł współpracy podsystemów
- plan testów systemu

Etap kończy: opinia recenzentów

3. Projektowanie i implementacja podsystemów

- współbieżna i niezależna realizacja sprzętu i oprogramowania podsystemów
- wynikiem etapu jest zbiór programowych i sprzętowych komponentów systemu

4. Dokumentacja komponentów systemu

- **funkcjonalna** – czy wszystkie wymagania wymienione w specyfikacji są spełnione
- **fizyczna** – czy komponenty są spójne i w pełni udokumentowane

Etap kończy: REWIZJA

5. Integracja, testowanie i ocena

5.1. Testowanie i integracja systemu

- połączenie wszystkich komponentów systemu i testowanie zgodnie z planem testów
- przegląd kwalifikacyjny

Wynik: raport testowania

5.2. Ocena i akceptacja – etap realizowany przez użytkowników systemu.

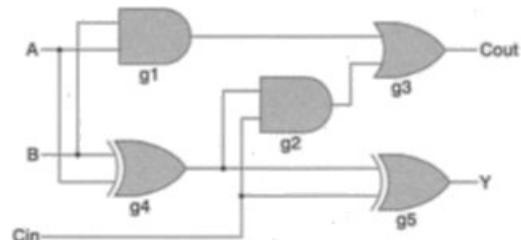
66.1. JĘZYK VERILOG. PIERWSZY PROJEKT W JĘZYKU VERILOG (SUMATOR JEDNOBITOWY). OPIS STRUKTURY. OPIS RÓWNAŃ LOGICZNYCH. OPIS ZA POMOCĄ INSTRUKCJI PROCEDURALNYCH

Początki języka Verilog sięgają wczesnych lat 80. ubiegłego stulecia. Głównym twórcą pierwotnej wersji języka był Phil Moorby – pracownik amerykańskiej firmy Gateway Design Automation, zajmującej się wytwarzaniem oprogramowania CAE. Firma ta w roku 1984 rozpoczęła sprzedaż symulatora układów cyfrowych pod nazwą Verilog (nieco później Verilog XL). W tamtym okresie nie istniał jeszcze żaden standard języka opisu sprzętu (VHDL również dopiero się rozwijał i nie stanowił jeszcze standardu IEEE), dlatego też Gateway stworzył własny język i nazwał go Verilog HDL. Język Verilog przeznaczony był wówczas stosowany wyłącznie do symulacji układów cyfrowych.

```
/* Sumator 1-bitowy (równania logiczne) */
module sum1_1(input A, B, Cin, output Y, Cout);
    wire g1_o, g2_o, g3_o, g4_o;

    and g1(g1_o, A, B);
    xor g4(g4_o, A, B);
    and g2(g2_o, Cin, g4_o);
    or g3(Cout, g1_o, g2_o);
    xor g5(Y, Cin, g4_o);

    endmodule
```



```
/* Sumator 1-bitowy (równania logiczne) */
module sum1_2(input cin, a, b, output s, cout);
    assign s = a^b ^ cin;
    assign cout = a & b | (a^b) & cin;
endmodule
```

```
/* Sumator 1-bitowy (proceduralnie) */
module sum1_3(input cin, a, b, output reg s, cout);
    always@(cin,a,b) begin
        if(a & b | cin & a | cin & b) cout = 1;
        else cout = 0;
        s = a^b^cin;
    end
endmodule
```

```
/* Sumator 1-bitowy (strukturalnie) */
module sum1_4(input cin, a, b, output s, cout);
    wire g1_o, g2_o, g3_o;
    and g1(g1_o, a, b);
    xor g2(g2_o, a, b);
    and g3(g3_o, g2_o, cin);
    xor g4(s, g2_o, cin);
    or g5(cout, g3_o, g1_o);
endmodule
```

66.2. HIERARCHICZNY OPIS SUMATORA 4-BITOWEGO

```
module sum4 (cin, x3, x2, x1, x0, y3, y2, y1, y0, s3, s2, s1, s0, cout);
    input cin, x3, x2, x1, x0, y3, y2, y1, y0;
    output s3, s2, s1, s0, cout;

    sum1 stage0 (cin, x0, y0, s0, c1);
    sum1 stage1 (c1, x1, y1, s1, c2);
    sum1 stage2 (c2, x2, y2, s2, c3);
    sum1 stage3 (c3, x3, y3, s3, cout);
endmodule
```

67. Jednostka testowa dla 1-bitowego sumatora

```
/* Moduł jednostki testowej sumatora jednobitowego
 * brak portów wejściowych i wyjściowych - jednostka testowa
 * nie może mieć portów */
module test_add_1();

    /* deklaracja zmiennych dla wektorów wejściowych
     * na wejścia cin, a, b */
    reg      tcin, ta, tb;

    /* deklaracja zmiennych dla funkcji wyjściowych */
    wire     ts, tcout;
```

```

/* utworzenie egzemplarza 1-bitowego sumatora
 * wraz z przyporządkowaniem zmiennych do bloku sumatora */
add_1_1 sum(tcin, ta, tb, ts, tcout);

/* Początek nazwanego bloku proceduralnego (musi posiadać nazwę),
 * wykonywanego tylko RAZ */
initial begin: test
    /* deklaracja zmiennej lokalnej */
    integer i;

    /* wywołanie funkcji systemowej */
    $display("Wynik symulacji 1-bitowego sumatora");

    /* deklaracja formatu czasu dla $display i $monitor */
    $timeformat(-9, 1, "hs", 8);

    /* monitor uruchamiany jest przy jakiejkolwiek zmianie
     * tcin, ta, tb, ts, tcout
     * wszystkich zmiennych na poniższej liście parametrów
     * %b - format binarny */
    $monitor($time, "cin=%b, a=%b, b=%b, s=%b, cout=%b",
             tcin, ta, tb, ts, tcout);

    /* Pętla for - podczas symulacji każda jej iteracja
     * wykonywana RÓWNOLEGLE */
    for (i = 0; i < 8; i = i+1) /* nie ma operatora inkrementacji, jak w C */
begin
    #10;           /* instrukcja opóźnienia na 10 jednostek czasowych */
    {tcin, ta, tb} = i;      /* tworzenie wektora wejściowego
                                * {tcin, ta, tb} = 000, 001, ..., 111 (bo i = 0...7)
                                * operacja konkatenacji { } */
end /* koniec pętli for */
end /* koniec nazwanego bloku proceduralnego */
end /* koniec modułu */

/* Wynik symulacji:
t    cin a  b s cout
#0      x  x  x  x  x
#10     0  0  0  0  0
#20     0  0  1  1  0
#30     0  1  0  1  0
#40     0  1  1  0  1
#50     1  0  0  1  0
#60     1  0  1  0  1
#70     1  1  0  0  1
#80     1  1  1  1  1

```

*/

List. 2.8. Kompletny kod jednostki testowej dla modułu sumatora pełnego – wersja 1

```
'timescale 1ns / 1ps
module jednostka_testowa;
    reg A;
    reg B;
    reg Cin;
    wire Y;
    wire Cout;

    suml uut (
        .A(A),
        .B(B),
        .Cin(Cin),
        .Y(Y),
        .Cout(Cout)
    );
    initial
    begin
        $monitor($time,": A=%b B=%b Cin=%b Y=%b Cout=%b",
            A,B,Cin,Y,Cout);
        #10; A = 0; B = 0; Cin = 0;
        #10; A = 0; B = 0; Cin = 1;
        #10; A = 0; B = 1; Cin = 0;
        #10; A = 0; B = 1; Cin = 1;
        #10; A = 1; B = 0; Cin = 0;
        #10; A = 1; B = 0; Cin = 1;
        #10; A = 1; B = 1; Cin = 0;
        #10; A = 1; B = 1; Cin = 1;
    end
endmodule
```

W opisie modułu jednostki testowej z listingu 2.8, wewnątrz instrukcji *initial* i bloku *begin...end* zastosowano zwykłą sekwencję instrukcji opóźnienia i instrukcji przypisania proceduralnego. W ten sposób kolejno odbywa się oczekiwanie 10 jednostek czasowych, po czym zmiennym *A*, *B*, *Cin* (zmienne te stanowią sygnały wejściowe dla testowanego modułu) jest nadawana odpowiednia wartość. W następnych linijkach kodu specyfikowane są kolejne kombinacje wartości tych trzech zmiennych.

Na uwagę zasługuje tu jeszcze jedna instrukcja, a dokładnie zadanie systemowe *\$monitor*, o którym do tej pory nie wspominaliśmy. Zadanie *\$monitor* wyświetla podczas symulacji wartości zmiennych, które znajdują się na jego liście parametrów – ale tylko wtedy, gdy nastąpi jakakolwiek zmiana wartości którejkolwiek z tych zmiennych (wyjątkiem jest wystąpienie *\$time* na liście parametrów). Sposób użycia *\$monitor* jest zasadniczo taki sam jak w przypadku *\$display* – można dokładnie tak samo formułować listę parametrów. Zaleta stosowania *\$monitor* w porównaniu z *\$display* jest taka, że zadanie *\$monitor* wystarczy wywołać (z odpowiednią listą parametrów) jednokrotnie na początku danego bloku *initial* i gdy tylko nastąpi zmiana wartości zmiennych z listy parametrów, wówczas zostanie wypisana ich wartość. Zadanie *\$display* wyświetla tylko bieżącą wartość zmiennych – taką, jaka istniała w momencie wywołania *\$display*.

68. POJĘCIA PODSTAWOWE. SŁOWA KLUCZOWE. IDENTYFIKATORY. ZNAKI BIAŁE. KOMENTARZE

68.1. Identyfikatory

Identyfikatory są nazwami, które nadawane są modułom, instancjom, zmiennym, funkcjom itd. Identyfikatory w języku Verilog mogą zawierać wszystkie duże i małe litery alfabetu (a-z, A-Z), cyfry (0-9), znak podkreślenia (_) oraz **znak dolara (\$)**. Inne znaki ASCII mogą być użyte w połączeniu z tzw. Identyfikatorem ominięcia (escaped identifier), czyli **poprzedzone znakiem lewego ukośnika (\)** i zakończone znakiem białym. Identyfikator musi rozpoczynać się od znaku litery lub znaku podkreślenia. **Długość** identyfikatora może sięgać aż **1024 znaków**. Bardzo istotne jest to, że język Verilog - w przeciwieństwie do np. VHDL – **odróżnia duże i małe litery alfabetu**.

68.2. Znaki białe

Znaki białe (white space) czynią kod bardziej czytelnym dla człowieka. Podobnie jak w językach programowania, znakami białymi są znaki spacji, tabulacja oraz znak nowej linii. Język Verilog nie jest językiem wrażliwym na białe znaki. Pisząc kod w tym języku, możemy wstawiać białe znaki praktycznie w dowolnym miejscu. Wyjątek stanowi tekst zawarty pomiędzy znakami cudzysłowu, który nie może być przedzielony znakiem nowej linii.

68.3. Komentarze

W języku Verilog istnieją dwa formaty dla linii kodu, które mają stanowić komentarz: format jednoliniowy i format blokowy. Komentarz jednoliniowy rozpoczyna się znakiem „//” (podwójny ukośnik) i rozciąga się do końca danej linii. Komentarz blokowy rozpoczyna się znakiem „/*”, kończy znakiem „*/” i może obejmować wiele linii kodu. Komentarzy blokowych nie można zagnieżdżać.

68.4. Słowa kluczowe

Słowo kluczowe (angielskie reserved word, keyword) to słowo (najczęściej pochodzenia angielskiego) lub jego fragment o znaczeniu określonym w definicji języka programowania, któremu nie można w programie nadać innego sensu. Przykłady słów zarezerwowanych: **begin, end, for, if, or, xor, module**.

69. WARTOŚĆ LOGICZNA BITU. REPREZENTACJA LICZB. PRZYKŁADY REPREZENTACJI LICZB

Czteropoziomowa wartość logiczna bitu

W języku Verilog każdy bit (zmienna, sieć logiczna, wyjście bramki itp.) może przyjmować 4 wartości. Są one następujące:

1. 0 – logiczne 0 lub fałsz,
2. 1 – logiczne 1 albo prawda,
3. x – wartość nieznana, nieistotna (don't care): jedna z wartości 0, 1 lub też w stanie zmiany,
4. z- stan wysokiej impedancji (np. na niepodłączonym porcie wejściowym)

W przypadku, gdy **wartość z** jest obecna na **wejściu bramki** lub gdy **pojawia się w wyrażeniu**, wówczas efekt jest zazwyczaj taki sam, jak gdyby była to **wartość x**.

Reprezentacja liczb

W Verilog istnieje specjalna notacja dla reprezentacji liczb o określonej podstawie. Poniżej pokazano ogólną formę takiej reprezentacji:

<rozmiar>'<podstawa><wartość>

Gdzie rozmiar jest liczbą bitów (podawaną w kodzie dziesiętnym), postawa reprezentuje symbol podstawy systemu, w którym jest zapisana liczba o wartości wartość. Poniżej wyszczególniono symbole dla reprezentacji liczb o różnych podstawach:

- ‘b lub ‘B – postawa binarna,
- ‘d lub ‘D – postawa dziesiętna,
- ‘h lub ‘H – postawa szesnastkowa,
- ‘o lub ‘O – postawa ósemkowa,

Liczba bitów oraz symbol postawy są opcjonalne. Domyślną postawą jest postawa dziesiętna. Domyślona liczba bitów zależy od implementacji, ale zazwyczaj wynosi 32 bity.

Jeżeli wartość ma **mniej bitów** niż podany rozmiar jest **uzupełniana zerami na najbardziej znaczących** bitach, chyba że **ostatnim** występującym **bitem** jest **x lub z**, wtedy jest uzupełniane **tą wartością**.

| liczba | wartość |
|--------|------------|
| 8'b0 | 0000 0000 |
| 8'bx | xxxx xxxx |
| 8'b1x | 0000 001x |
| 8'b0x | 0000 000x |
| 8'hx | xxxx xxxx |
| 8'b1 | 0000 0001 |
| 8'hz1 | zzzz 0001 |
| 8'bx1 | xxxx xxxx1 |
| 8'bx0 | xxxx xxxx0 |
| 8'hz | zzzz zzzz |
| 8'h0z | 0000 zzzz |

Przykłady

8'b1011011

16'habcd

121

W pierwszym przypadku mamy do czynienia z 8-bitową liczbą podaną w zapisie binarnym. W drugim przypadku jest to 16-bitowa liczba zapisana w kodzie szesnastkowym. Ostatnia liczba podana jest domyślnie w kodzie dziesiętnym.

70.1. MODUŁ I LISTA PORTÓW WEJŚCA-WYJŚCIA. MODUŁ. DEFINICJA MODUŁU

Moduł - podstawowa jednostka projektu w języku Verilog, opisywany w tym języku system cyfrowy składa się ze zbioru modułów. Oprócz nazwy, ma tzw. Porty we-wy, czyli pewne sygnały umożliwiające komunikację z otoczeniem. Każdy moduł ma interfejs do innych modułów i opis architektury.

Reprezentuje logiczną jednostkę, która może zostać opisana poprzez specyfikację wewnętrznej struktury logicznej (bramki) lub przez opis sposobu zachowania (język programowania). Moduły mogą być łączone ze sobą przez sieci (nets), co umożliwia ich wzajemną komunikację.

Moduły przypominają nieco procedury lub funkcje w językach programowania oprogramowania.

Ogólna definicja modułu

```
module nazwa [lista portów] ;  
    [deklaracje i instrukcje]  
endmodule
```

gdzie **lista_portów** ma postać:

```
{port {,port} }
```

70.2. PORTY. PRZYKŁADY DEKLARACJI PORTÓW. OPIS MODUŁU Z NOWYM STYLEM DEKLARACJI PORTÓW. OPIS MODUŁU ZE STARYM STYLEM DEKLARACJI PORTÓW

Port może być pewnym wyrażeniem, najczęściej jednak wygląda następująco:

```
typ_portu [zakres] nazwa_portu {,nazwa_portu}
```

gdzie **typ_portu** może być jednym ze słów kluczowych:

- **input** (wejście)
- **output** (wyjście)
- **inout** (port dwukierunkowy)

za którym występuje nazwa typu danych:

- reg (tylko dla output)
- wire (domyślny)

Dodatkowo pomiędzy typem portu a listą nazw sygnałów może wystąpić **specyfikator zakresu** (dany port może być **pojedynczym bitem**, może też być **wektorem – magistralą** o zadanej liczbie bitów). Specyfikatora zakresu używa się również podczas definicji zmiennych (sygnałów-magistral) dla danego modułu. Forma użycia specyfikatora zakresu jest następująca:

[msb:lsb]

gdzie:

- msb – pozycja najbardziej znaczącego bitu
- lsb – pozycja najmniej znaczącego bitu

Oba muszą być wyrażeniami stałymi, nieujemnymi. Nawiasy kwadratowe stanowią część opisu składni i nie oznaczają elementu opcjonalnego.

Przykład (**nowy styl deklaracji modułu – jak w C**)

```
module sum1 (input A, B, Cin,
              output Y, Cout);
    wire g1_o,g2_o,g3_o,g4_o;

    and g1(g1_o,A,B);
    xor g4(g4_o,A,B);
    and g2(g2_o,Cin,g4_o);
    or g3(Cout,g1_o,g2_o);
    xor g5(Y,Cin,g4_o);
endmodule
```

Zdefiniowaliśmy 5 portów we-wy: 3 porty wejściowe o nazwie A, B, Cin oraz dwa porty wyjściowe o nazwach Y oraz Cout. Nie określiliśmy tu jawnie typu danych sygnałów (wire czy reg), a jedynie ich kierunek (we/wy).

Przykład (**stary styl deklaracji modułu**)

```
module sum1(A, B, Cin, Y, Cout);

    input     A, B, Cin;
    output    Y, Cout;
    reg      Y, Cout;
    wire     A, B, Cin;
(...)
```

Czyli po prostu w nawiasie tylko nazwy a specyfikacja typów niżej.

71. INSTANCJE MODUŁÓW. SKŁADNIA UTWORZENIA INSTANCJI MODUŁU. TERMINALI. NOTACJI POPRZEZ NAZWĘ. NOTACJI POZYCYJNE. HIERARCHIA

Instancja (konkretyzacja) modułu – użycie tego modułu, utworzenie rzeczywistego egzemplarza modułu i połączenie jego portów z innymi komponentami opisywanego system cyfrowego. Sam opis modułu stanowi tylko definicję jego działania (pełnionej funkcji) albo jest specyfikacją struktury (lub jedno i drugie).

Dopiero utworzenie instancji tego modułu z wnętrza innego modułu powoduje, że podczas procesu syntezy logicznej utworzony zostanie konkretny egzemplarz naszego modułu i zostaną mu przydzielone określone zasoby logiczne.

Składnia utworzenia modułu jest następująca

`identyfikator_modułu [przypisania_wartości_parametrów] instancja_modułu {, instancja_modułu};`

gdzie:

identyfikator_modułu – oznacza nazwę modułu, którego instancję tworzymy

a **instancja_modułu** ma postać:

`nazwa_instancji ([lista_połączeń_portów])`

z kolei **lista_połączeń_portów** jest jednym z poniższych:

1. `uporządkowane_połączenie_portu {, uporządkowane_połączenie_portu}`
2. `nazwane_połączenie_portu {, nazwane_połączenie_portu}`

Użycie unikalnej nazwy instancji modułu jest obowiązkowe. Ogólnie lista_połączeń_portów może być traktowana jako lista tzw. terminali. Poszczególne terminale instancji dołączane są do odpowiednich portów modułu. Terminale dołączane do portów wejściowych modułu mogą być dowolnymi wyrażeniami. Terminale dołączane do portów wyjściowych lub dwukierunkowych mogą być identyfikatorami, wektorami, jedno- lub wielobitowymi fragmentami wektorów lub sklejeniem takich fragmentów lub pojedynczych bitów. Istotne jest to, że liczba bitów terminala musi być identyczna z liczbą bitów oczekiwana przez dany port modułu (w przypadku niezgodności generowane są ostrzeżenia, nie są to błędy więc nie zatrzymują procesu syntezy – może to powodować nieprawidłowe działanie układu).

W przypadku użycia jako terminalu **niezadeklarowanego** identyfikatora (zmiennej) terminal ten jest niejawnie deklarowany jako skalar (**sygnał 1-bitowy**) typu **wire** i może być użyty wszędzie tam, gdzie dopuszczalne jest użycie sygnałów typu wire.

W celu określenia sposobu połączenia terminali instancji z poszczególnymi portami modułu można posłużyć się notacją:

- **pozycyjną** – istotna jest kolejność terminali na liście połączeń, pierwszy terminal z listy jest dołączany do pierwszego portu modułu, itd.
- **przez nazwę** – w sposób jawnym podajemy, który port modułu należy połączyć z którym terminaliem instancji. Wówczas nazwane połaczenie portu ma postać:
`.identyfikator_portu ([wyrażenie])`
gdzie
identyfikator_portu - nazwa portu modułu, do którego dołączamy wyrażenie instancji modułu

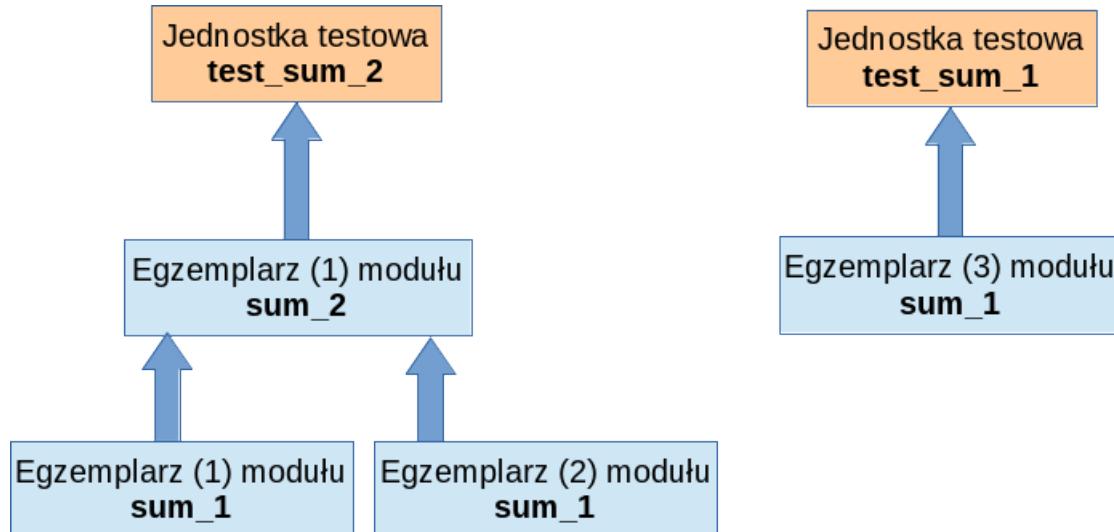
Hierarchia modułów

W środowisku projektowym Quartus II w momencie dodania nowego modułu automatycznie tworzona jest hierarchia modułów.

Modułem nadrzędnym jest moduł tworzący egzemplarz innego modułu, który jest jego modułem podrzędnym:

Jednostka testowa jest zawsze modułem nadrzędnym.

Przykład (sum_2 – sumator 2-bitowy, sum_1 – sumator 1-bitowy)



72. PARAMETRY. NOTACJA DEFINICJI MODUŁU Z PARAMETRAMI. PRZYPISANIE WARTOŚCI PARAMETRÓW

Parametry pozwalają na wprowadzenie nazw symbolicznych dla wartości i wyrażeń, które będą stosowane w opisie modułu. Technika stosowania parametrów umożliwia definiowanie sparametryzowanych modułów, które mogą być używane w różnych sytuacjach.

Istnieje możliwość nie tylko ponownego użycia definicji modułu w innych projektach (np. z różną liczbą bitów dla portów wejścia-wyjścia), ale również można definiować parametry, których wartość można zmieniać podczas tworzenia instancji danego modułu.

Ogólnie specyfikacja parametru jest częścią definicji modułu. Notacja definicji modułu z parametrami jest następująca:

```
module nazwa [lista_parametrów] [lista_portów];  
    deklaracje_i_instrukcje  
endmodule
```

gdzie lista_portów ma taką samą postać jak w przypadku definicji modułu bez parametrów, a lista_parametrów wygląda następująco:

```
#(deklaracja_parametru {, deklaracja_parametru})
```

Z kolei deklaracja_parametru ma postać jedną z poniższych:

1. parameter [signed] [zakres] lista_przypisań_parametrów;
2. parameter integer [zakres] lista_przypisań_parametrów;
3. parameter real [zakres] lista_przypisań_parametrów;
4. parameter realtime [zakres] lista_przypisań_parametrów;
5. parameter time [zakres] lista_przypisań_parametrów;

Występująca w powyższym zapisie lista_przypisań_parametrów może być określona następująco:

```
nazwa_parametru=wartość {, nazwa_parametru=wartość}
```

Ogólna postać utworzenia instancji modułu wraz z redefinicją wartości parametru jest taka, jak wyżej, gdzie przypisania_wartości_parametrów wyglądają następująco:

```
#{lista_przypisań_parametru}
```

A lista_przypisań_parametru jest jednym z poniższych:

1. Uporządkowane_przypisanie_parametru {, i uporządkowane_przypisanie_parametru}
2. Nazwane_przypisanie_parametru {, nazwane_przypisanie_parametru}

Podczas tworzenia instancji modułu, w celu zmiany domyślnej wartości parametrów można posłużyć się albo notacją pozycyjną, albo opartą na nazwach – identycznie jak dla terminali modułu

Parametry mogą być również definiowane wewnątrz opisu modułu. Stanowią one wówczas rodzaj stałych symbolicznych, wykorzystywanych wewnątrz kodu. Każda instancja modułu może redefiniować (zmieniać) wartości parametrów.

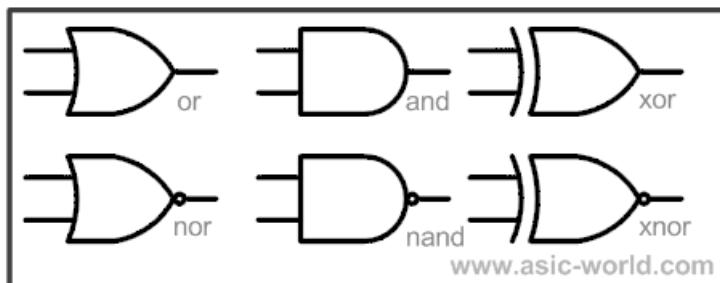
Jeżeli chcemy uniknąć możliwości zmiany wartości parametru w sposób bezpośredni (np. za pomocą defparam z poziomu innego modułu), co może być przydatne chociażby w przypadku, gdy definiujemy nazwy symboliczne poszczególnych stanów automatu sekwencyjnego, to podczas definicji parametru zamiast słowa kluczowego parameter należy użyć localparam. Składnia definicji parametru ze słowem kluczowym localparam jest identyczna jak dla parameter.

73. PODSTAWOWE ELEMENTY LOGICZNE. BRAMKI LOGICZNE. SKŁADNIA INSTANCJI BRAMEK

Do celów modelowania strukturalnego w języku Verilog mamy do dyspozycji **26 podstawowych elementów logicznych (primitives)**. Do tych elementów zaliczane są **bramki logiczne** i **przełączniki** (switches). Bardziej szczegółowo zbiór podstawowych elementów logicznych można podzielić na **trzy kategorie**.

Kategorie wraz z nazwami poszczególnych elementów:

1. **Bramki logiczne:** and, nand, or, nor, xor, xnor
2. **Bufory:** buf, not, bufif0, bufif1, notif0, notif1, pullup, pulldown
3. **Tranzystory:**
 - a) nmos, pmos, cmos,
 - b) rnmos, rpmos, rcmos,
 - c) tran, rtran
 - d) tranif0, tranif1,
 - e) rtranif0, rtranif1



Modelując układ cyfrowy na poziomie bramek logicznych posługujemy się instancjami. Utworzenie instancji (instantiation) elementów takich jak bramki logiczne lub moduły oznacza ich użycie, utworzenie konkretnego egzemplarza i podłączenie z pozostałą częścią opisywanego układu (innymi bramkami, modułami, sygnałami itp.).

Sposób podłączenia podstawowych elementów logicznych wykorzystuje tzw. notację pozycyjną. Dla kategorii **bramek logicznych** **pierwszy z terminali** listy połączeń reprezentuje **wyjście** danej bramki, **pozostałe to wejścia** (bramki mogą mieć wiele wejść). Składnia utworzenia instancji bramek logicznych jest następująca:

```
typ_bramki [siła] [opóźnienie] n_instancja_bramki {, n_instancja_bramki };
```

gdzie:

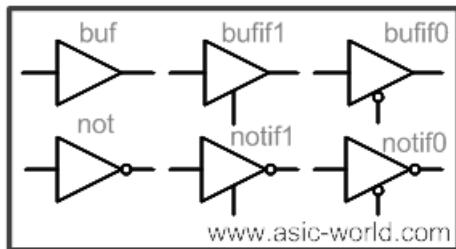
- **typ_bramki** jest jednym z symboli wymienionych w kategorii bramki logiczne,
- **siła** jest opcjonalnym elementem charakteryzującym pewne właściwości elektryczne obwodu wyjściowego (drivera) bramki (stosowane głównie do modelowania na poziomie przełączników),
- **opóźnienie** określa wartość czasu propagacji sygnału przez bramkę (stosowane do celów symulacji),

n_instancja_bramki ma postać:

```
[nazwa_instancji] (terminal_wyjsciowy, terminal_wejsciowy {, terminal_wejsciowy} )
```

74. BUFORY. BUFORY TRÓJSTANOWE. ELEMENTY LOGICZNE DEFINIOWANE PRZEZ UŻYTKOWNIKA

Bufory: buf, not, bufif0, bufif1, notif0, notif1, pullup, pulldown



Składnia utworzenia instancji buforów jest następująca:

```
typ_bufora [siła] [opóźnienie] n_instancja_bufora {, n_instancja_bufora};
```

gdzie

- **typ_bufora** jest jednym z symboli wymienionych w kategorii bufora,
- **siła** jest opcjonalnym elementem charakteryzującym pewne właściwości elektryczne obwodu wyjściowego (drivera) bufora (stosowane głównie do modelowania na poziomie przełączników),
- **opóźnienie** określa wartość czasu propagacji sygnału przez bufor (stosowane do celów symulacji),

n_instancja_bufora ma postać:

```
[nazwa_instancji] (terminal_wyjsciowy, terminal_wyjsciowy {, terminal_wejsciowy} )
```

Sposób podłączenia terminali bazuje na notacji pozycyjnej. Dla buforów **not i buf ostatni** (prawy skrajny) terminal reprezentuje **wejście**, a **pozostałe** terminale to **wyjścia** (bufory not i buf mogą mieć wiele wyjść).

Bufory bfif0, bufif1, notif0, notif1 to bufory trójstanowe. Mają one dokładnie 3 terminale.:

1. ostatni terminal (prawy skrajny) reprezentuje **wejście sterujące**,
2. terminal środkowy stanowi **wejście sygnałowe**,
3. pierwszy z lewej terminal to **wyjście**.

Bufory typu bufif0 i notif0 są aktywne, gdy wejście sterujące znajduje się w stanie niskim (notif0 ma dodatkowo zanegowane wyjście), zaś bufif1 i notif1, gdy jest on wysoki (notif1 ma zanegowane wyjście). Bufory pulldown i pullup mają tylko jeden terminal reprezentujący wyjście.

Każdy z terminali wymienionych buforów może być tylko 1-bitowym sygnałem lub wyrażeniem.

Podstawowe elementy logiczne definiowane przez użytkownika – UDP (user defined primitives) to elementy, które może zdefiniować i dołączyć samodzielnie. Język Verilog dostarcza specjalny mechanizm definicji i opisu takich elementów.

Można je również definiować w ramach jednostek projektowych (np. w module) i następnie używać ich w projekcie, tworząc odpowiednie instancje tych modułów. Jeśli używamy zintegrowanego środowiska projektowego (np. Xilinx ISE), mamy do dyspozycji szereg bibliotek elementów (m. in. multiplekserów, przerzutników, komparatorów itp.) zorganizowanych jako UDP.

75. TYPY DANYCH. TYP WIRE I JEGO POCHODNE. TYP REG. TYPY LICZBOWE

W języku Verilog występują dwa **podstawowe** typy danych: **wire** i **reg**.

Wire reprezentuje fizyczne połączenia (przewody) w opisywanym układzie. Dane (sygnały) zadeklarowane jako wire często nazywane są sieciami (nets). Dane tego typu, w przeciwieństwie do zmiennych typu reg, nie przechowują swojej wartości, lecz za pomocą sygnałów zadeklarowanych jako wire dokonuje się połączenie międzyinstancjami elementów takich jak bramki logiczne oraz instancjami modułów. Dane zadeklarowane jako wire mogą być sterowane na dwa sposoby:

1. poprzez dołączenie sygnałów tego typu do wyjścia bramki lub modułu,
2. poprzez przypisanie wartości za pomocą tzw. przypisania ciągłego (continuous assignments).

Danych typu wire nie można przypisać wartości z wnętrza funkcji lub bloku begin... end (w tym z wnętrza bloku always), wartość tę można jednak odczytać.

Typ wire jest w języku Verilog typem domyślnym. Oznacza to, że nie ma konieczności jawnego deklarowania danych skalarnych jako wire (deklarację możemy zwyczajnie pominąć).

W języku Verilog występuje jeszcze kilka typów **pochodnych** od typu **wire**. Są to **wand**, **wor**, **tri**, **triand**, **trior**, **tri1**, **tri0**, **trireg**, **supply1**, **supply0**.

- Typ wand oraz triand reprezentują funkcję logiczną AND na przewodzie (wired AND). Użycie danych tego typu przypomina nieco logikę układów z otwartym kolektorem- jeżeli którykolwiek z sygnałów ma wartość 0, rezultatem jest również 0.

- Typ danych `wor` oraz `trior` reprezentują z kolei logikę typu OR na przewodzie: jeżeli którykolwiek z sygnałów ma wartość 1, rezultatem jest również 1.
- Typy `tri1` i `tri0` stosowane są w przypadku, gdy ważna jest domyślna wartość sygnału, który nie jest sterowany przez żadnego wyjście. Domyślną wartością sygnału `tri1` jest 1, a sygnału `tri0` jest 0.
- Typ `supply1` oraz `supply0` modelują linie zasilania.
- Typ `trireg` jest stosowany w modelu na poziomie przełączników (switch level) i nie ma istotnego znaczenia dla celów syntezy logicznej.

Typ danych **`reg`** reprezentuje zmienne w języku Verilog. Danych tego typu używa się wewnętrz bloków proceduralnych (w tym bloku `always`). Użycie danych typu `reg` nie zawsze musi oznaczać, że modelowane będą rejestrzy (przerzutniki typu flip-flop lub zatrzaski). Ten typ danych jest stosowany również do modelowania układów czysto kombinacyjnych.

Zarówno danego typ `wire` jak i `reg` mogą mieć postać sygnałów skalarnych (o szerokości 1 bitu) oraz mogą być wektorami (wielobitowymi magistralami).

Oprócz opisanych wyżej typów danych w języku Verilog występują jeszcze dwa typy- bardzo podobne do typów często spotykanych w popularnych językach programowania. Są to typy **`integer`** i **`real`**.

Zmienne zadeklarowane jako `integer` przechowują wartości numeryczne – liczby całkowite ze znakiem. Zmienne tego typu mają rozmiar 32 bitów i pod pewnym względem podobne są do 32-bitowych wektorów typu `reg`.

Z kolei typ danych `real` reprezentuje 32-bitowe zmiennoprzecinkowe liczby rzeczywiste. Te dwa typy danych znajdują ogromne zastosowanie do celów syntezy logicznej (szczególnie typ `real`)- bardziej zaś przydane są w opisie zachowania otoczenia projektowanego systemu cyfrowego (symulacje). Do celów symulacji stosowane są również typy `time` oraz `realtime`. Reprezentują one czas symulacji.

76. ŁĄCZENIE WYJŚĆ BRAMEK ZA POMOCY TYPÓW `WAND` I `WOR`. TYP SIECI `WAND`. TYP SIECI `WOR`. WARTOŚCI SYGNAŁÓW W SIECI

Sieć (net) reprezentuje fizyczne połączenia pomiędzy elementami. Nie jest elementem pamiętającym (wyjątek: `trireg`). Wartość jest ustalana na podstawie sygnałów zasilających. Domyślna wartość portów we/wy: `wire`. Jeżeli nie ma **połączenia zasilającego**, to jest w stanie **wysokiej impedancji z (!)**.

Typy sieci:

- `wire`, `tri` – węzeł, węzeł trójstanowy
- `supply0`, `supply1` – stała wartość logiczna
- `wand`, `wor` – iloczyn, suma na drucie
- `trior`, `triand`, `tri0`, `tri1`, `trireg`

połączenie WAND / TRIAND

```
module wand_triand_module(input A, B, C, output wand Y);
    assign Y = A & B; // przypisanie ciągłe do typu sieciowego
    assign Y = B | C; // przypisanie ciągłe do typu sieciowego
endmodule
```

połączenie WOR / TRIOR

```
module wor_trior_module(input A, B, C, output wor Y);
    assign Y = A & B;
    assign Y = B | C;
endmodule
```

Połączenia WAND i WOR umożliwiają nam użycie na połączenie, dowolnych funkcji wynikowych za pomocą bramki AND lub OR np.:

```
// Dla funkcji WOR
wor Y;
assign Y = F & G;
assign Y = B | C;
// lub:
assign Y = (F & G) | (B | C);
```

```
// Dla funkcji WAND
wand Y;
assign Y = F & G;
assign Y = B | C;
// lub:
assign Y = (F & G) & (B | C);
```

77. SELEKCJA BITÓW. TABLICE INSTANCJI (ARRAY OF INSTANCES)

Selekcja bitów to wybór pojedynczego bitu lub części bitów zmiennych lub sygnałów, które są wektorami. Składnia selekcji bitu lub jego częściowej selekcji jest następująca (nawias kwadratowy jest częścią opisu składni i nie oznacza elementu opcjonalnego) :

identyfikator [wyrażenie_zakresu]

gdzie **wyrażenie_zakresu** może być jednym z poniższych:

1. wyrażenie
2. wyrażenie_stale_msb : wyrażenie_stale_lsb
3. wyrażenie_podstawy + : wyrażenie_stale_szerokość
4. wyrażenie_podstawy - : wyrażenie_stale_szerokość

W pierwszym przypadku, gdy wyrażenie_zakresu jest zwykłym wyrażeniem, mamy do czynienia z selekcją **pojedynczego bitu** o numerze, który jest wartością tego wyrażenia (zwróćmy uwagę, że wyrażenie to nie musi być stałe).

W drugim przypadku mamy do czynienia z **częściową selekcją bitów** wektora o **numerach** zawartych **pomiędzy** wartościami wyrażeń **wyrażenie_stale_lsb** a **wyrażenie_stale_msb**.

W dwóch ostatnich przypadkach jako **wyrażenie_podstawy** podaje się numer **bitu początkowego**, a jako **wyrażenie_stale_szerokość** jest podawana **liczba bitów**. Liczba bitów jest następnie dodawana (jeżeli przed znakiem dwukropka znajduje się znak +) lub odejmowana (jeżeli przed znakiem dwukropka znajduje się znak -) od wyrażenie_podstawy tworząc konstrukcję analogiczną do częściowej selekcji bitu.

Tablice instancji znajdują swoje zastosowanie w metodach specyfikacji, gdzie numeracja kolejnych instancji różni się w sposób kontrolowany. Przykładowo bufif0 b[1:8] (out, in, oe) zastępuje długi opis 8-bitowego modułu bufora trójstanowego zrealizowanego z użyciem jednego z podstawowych elementów logicznych.

Specyfikacja z tablicą instancji jest tworzona przy użyciu opcjonalnego specyfikatora zakresu występującego tuż za nazwą instancji. Nie występują tutaj istotne ograniczenia wartości granicznych specyfikatora zakresu, z wyjątkiem tego, że obie liczby muszą być całkowite. Nie jest ważne, czy jedna jest większa od drugiej – w przypadku, gdy są równe, tworzona jest tylko jedna instancja. Przy zakresie zdefiniowanym przez dwie wartości msb i lsb, liczba wygenerowanych instrukcji wynosi: $\text{abs(msb} - \text{lsb)} + 1$.

78. PRZYPISANIA CIĄGŁE. PRZYPISANIA CIĄGŁE O RÓŻNEJ LICZBIE BITÓW

W przypadku, gdy nie znamy struktury układu kombinacyjnego, który chcemy opisać lub nie chcemy modelować układu na poziomie bramek logicznych, ale znamy równania algebry Boole'a, które ten układ opisują, do jego opisu możemy zastosować tzw. przypisania ciągłe (continuous assignment).

Jednobitowy sumator pełny opisany jest następującymi równaniami (można je również bezpośrednio wyprowadzić ze schematu):

$$Y = A \wedge B \wedge Cin; Cout = A \cdot B + (A \wedge B) \cdot Cin$$

gdzie „ \cdot ”, „ $+$ ” oraz „ \wedge ” to operatory odpowiednio: iloczynu logicznego, sumy logicznej oraz operacji XOR.

Opis modułu sumatora z użyciem przypisania ciągłego

```
module sumator1(input A, B, Cin, output Y, Cout);  
    assign Y      = A^B^Cin;                      // (1)  
    assign Cout = (A & B) | ((A^B) & Cin);       // (2)  
endmodule
```

Przypisania ciągłe występują w liniach oznaczonych jako (1) oraz (2) i rozpoczynają się od słowa kluczowego (instrukcji) **assign**. W języku Verilog, identycznie jak w języku C, operatory sumy, iloczynu bitowego oraz bitowej różnicy symetrycznej mają postać odpowiednio „|”, „&”, „^”

Ogólnie, użycie przypisań ciągłych jest bardzo podobne do specyfikacji układu logicznego za pomocą równań algebry Boole'a, z tym jednym wyjątkiem, że w języku Verilog istnieje dużo więcej operatorów, które można wykorzystać. Przypisania ciągłe pozwalają opisać układ kombinacyjny bez zajmowania się implementacją jego struktury.

Przypisań ciągłych można użyć w dwojaki sposób:

1. Stosując jawne przypisanie za pomocą instrukcji **assign**
2. Specyfikując przypisanie w tej samej linii, w której występuje deklaracja **wire**.

Lewa strona przypisania ciągłego może być sygnałem typu **wire** (i **pochodnym**, jak **wand**, **wor**, **tri**), **selekcją bitu** lub **częściową selekcją** oraz **sklejeniem (konkatenacją)** jednego lub więcej bitów wektora. **Nie można zastosować** przypisania ciągłego do zmiennych typu **reg**.

Prawa strona operatora przypisania ciągłego może być wyrażeniem zawierającym dowolny z operatorów dostępnych w języku Verilog oraz wcześniej zadeklarowanych zmiennych (sygnałów) oraz funkcji.

Przypisania ciągłe są wykonywane z uwzględnieniem kolejności bitów: najmniej znaczący bit prawej strony przypisania jest przypisywany do najmniej znaczącego bitu lewej strony itd. Jeżeli liczba bitów **prawej** strony jest **większa** niż liczba bitów **lewej** strony, to **najbardziej znaczące bity** prawej strony są **odrzucane**.

Jeżeli liczba bitów **lewej** strony jest **większa** niż liczba bitów **prawej** strony, wówczas operand po **prawej** stronie jest **uzupełniany zerami** na najbardziej znaczących pozycjach bitu.

Zilustrujemy teraz praktycznie dwie wspomniane wyżej właściwości przypisania ciągłego. Najpierw wykonamy **przypisanie ciągłe niejawnie**, bez użycia instrukcji **assign** - w tej samej linii, w której występuje deklaracja **wire**. Przypadek taki występuje na listingu 1.13 w linii oznaczonej jako **(1)**. Zadeklarowano tutaj dodatkowy sygnał **a_xor_b**, przypisując mu bitową różnicę symetryczną wejść **A** i **B**.

1.13. Równoważna wersja opisu modułu sumatora

```
module sumator_1a(input A, B, Cin, output Y, Cout);  
    wire    a_xor_b    = A^B;          // (1)  
    assign Y          = a_xor_b^Cin;  
    assign Cout      = (A & B) | (a_xor_b & Cin);  
endmodule
```

1.14. Opis modułu sumatora z użyciem operatora dodawania

```
module sumator_1b(input A, B, Cin, output Y, Cout);
    assign {Cout,Y} = A+B+Cin; // maksymalny wynik po prawej = 3 (dwa bity wyniku)
endmodule
```

1.15. Opis modułu 8-bitowego sumatora

```
module sumator_1c(input [7:0] A, B, input Cin, output Cout, output [7:0] Y);
    assign {Cout,Y}=A+B+Cin;
endmodule
```

Druga właściwość przypisania ciągłego, którą tu zilustrujemy, będzie dotyczyła przypisań sygnałów o różnej liczbie bitów. Wiemy, że nasz układ kombinacyjny z rysunku 1.1 jest sumatorem - realizuje operację sumy arytmetycznej.

W języku Verilog występuje operator sumy arytmetycznej (i co ważne, jest on powszechnie implementowany przez narzędzia syntez), którego możemy użyć w przypisaniu ciągłym.

Przypadek taki zilustrowano na listingu 1.14. Występuje tam tylko jedna linijka kodu z przypisaniem ciągłym. Liczba bitów po prawej stronie operatora przypisania ciągłego wynosi 2, gdyż tyle bitów wymaga wynik sumy arytmetycznej sygnałów 1-bitowych (jeżeli uwzględnimy bit przeniesienia). Po lewej stronie operatora przypisania ciągłego zastosowano sklejenie (konkatenację) dwóch sygnałów Cout oraz Y, używając do tego celu operatora sklejania (`{}`). W ten sposób wynik sumy przypisany jest do sygnału (wyjścia) Y, a wartość bitu przeniesienia trafia do sygnału (wyjścia) Cout.

Taka sama sytuacja mogłaby mieć miejsce w przypadku, gdyby nasz sumator wykonywał operacje nie na pojedynczych bitach, lecz na wektorach. Na listingu 1.15 przedstawiono opis sumatora 8-bitowego. Jak widać, modyfikacji uległa jedynie deklaracja portów wejścia-wyjścia (dodano specyfikator zakresu).

79. OPERATORY (STATEMENTS). TABLICA OPERATORÓW JĘZYKA VERILOG. OPERATORY ARYTMETYCZNE

Operatory identyfikują rodzaj operacji wykonywanej na operandach w celu uzyskania odpowiedniej wartości wyrażenia. Operandy są to argumenty operatorów. W języku Verilog operandami mogą być:

1. liczby,
2. sieci,
3. zmienne reg (także dane innych typów, jak. np. integer),
4. selekcja bitu,
5. częściowa selekcja
6. wywołania funkcji.

Z kolei wyrażenie składa się z jednego lub wielu operandów przedzielonych operatorami. Wyrażenia używane są wszędzie tam, gdzie wymagana jest wartość w języku Verilog.

Większość występujących w języku Verilog operatorów jest operandami jedno- lub dwuargumentowymi. Wyjątek stanowi **operator warunkowy (trójargumentowy)** oraz **operator sklejania (wieloargumentowy)**.

Operatory arytmetyczne

Realizują podstawowe operacje arytmetyczne, jak:

- dodawanie (+)
- odejmowanie (-)
- mnożenie (*)
- dzielenie (/)
- potęgowanie (**)
- reszta z dzielenia (%)

Operandy typu **reg** oraz **sieci** (wire, wand, wor, itd.) są traktowane jako **liczby bez znaku**. Jeżeli mamy do czynienia z operandami typu **real** lub **integer**, to **mogą** być one traktowane jako **liczby ze znakiem**. Operator ‘-‘ może być dwuargumentowy lub jednoargumentowy. W tym drugim przypadku użycie tego operatora oznacza zmianę znaku operandu. Operator potęgowania działa zgodnie z zależnością: wynik=podstawa**wykładnik. Jeżeli jakikolwiek bit operandu operatorów arytmetycznych jest **nieistotny** – ma wartość x – **don't care lub z – wysoka impedancja**) wówczas **wynik jest również nieistotny (wartość x)**.

| kategoria | operator | opis |
|------------------------|----------------------------------|--|
| Operatory arytmetyczne | + - * / ** % | suma, różnica, mnożenie, dzielenie potęgowanie reszta z dzielenia |
| Operatory relacji | > >= < <= | relacje |
| Operatory porównania | == != === !== | logiczna równość logiczna nierówność logiczna(literalna) równość logiczna(literalna) nierówność |
| Operatory logiczne | ! && | negacja logiczna NOT iloczyn logiczny AND suma logiczna OR |
| Operatory bitowe | ~ & ^ ^~ ^~ | negacja bitowa NOT iloczyn bitowy AND suma bitowa OR bitowa różnica symetryczna XOR XNOR |
| Operatory redukcji | & ~& ~ ^ ^~ ^~ | redukция AND redukция OR redukция NAND redukция NOR redukция XOR redukция XNOR |
| Operatory przesunięcia | << >> <<< >>> | przesunięcie bitowe w lewo przesunięcie bitowe w prawo przesunięcie bitowe ze znakiem w lewo przesunięcie bitowe ze znakiem w prawo |
| Operator warunkowy | ?: | operator warunkowy |
| Operator sklejania | {} | sklejenie |

80. OPERATORY RELACJI. OPERATORY PORÓWNANIA. OPERATORY LOGICZNE

Operatory relacji

1. większy ($>$)
2. większy lub równy (\geq)
3. mniejszy ($<$)
4. mniejszy lub równy (\leq)

Wynikiem działania operatorów relacji są zawsze dwie wartości: 0 (gdy dana relacja jest fałszywa) i 1 (gdy relacja jest prawdziwa). Identycznie jak w przypadku operatorów arytmetycznych, operandy typu **reg** oraz **sieci** są traktowane jako **liczby bez znaku**. Operandy typu **real** i **integer** mogą być **liczbami ze znakiem**. Jeżeli jakikolwiek **bit** operandu jest **nieistotny** (x lub z), to **wynik relacji jest również nieistotny**.

Moduł realizujący dodawanie lub odejmowanie

```
module komparator
#(parameter rozmiar=8)
  (input [rozmiar-1:0] A, B, output Y);

  assign Y=(A>B);
endmodule
```

Operatory porównania

- równy ($==$)
- różny ($!=$)
- literalnie równy ($==>$)
- literalnie różny ($!=>$)

Wynikiem działania operatorów porównania są również tylko dwie wartości, 0 (fałsz) lub 1 (prawda). Dla operatorów typu $==$ oraz $!=$ w przypadku, gdy **którykolwiek** z bitów operandów jest **nieistotny**, wynik porównania jest również **nieistotny**.

Literalne operatory porównania wprowadzono wraz ze standardem Verilog-2001 i w ich przypadku **porównanie** następuje **również z wartościami x oraz z**. Aby wynik porównania był prawdziwy, wszystkie bity operandów muszą być identyczne.

Różnica w znaczeniu literalnych operatorów porównania oraz zwykłych operatorów porównania jest istotna w przypadku symulacji. **Podczas syntezy wszystkie literalne operatory porównania zamieniane są na zwykłe odpowiedniki.**

Identycznie jak w przypadku operatorów arytmetycznych i operatorów relacji, operandy typu reg oraz sieci są traktowane jako liczby bez znaku. Operandy typu real i integer mogą być liczbami ze znakiem.

Operatory logiczne

- negacja logiczna (!)
- suma logiczna (||)
- iloczyn logiczny (&&)

Wyrażenia z operatorami logicznymi generują wartości 1 albo 0 (pojedynczy bit) w zależności od tego, czy wyrażenie jest prawdziwe czy też fałszywe.

Operator negacji logicznej przekształca niezerową wartość (prawda) w wartość równą 0 (fałsz), natomiast wartość 0 przekształca w 1. Operator iloczynu logicznego generuje wartość 1 tylko wtedy, gdy obydwa operandy są prawdziwe, w przeciwnym wypadku daje wartość 0. W przypadku operatora sumy logicznej, daje on wartość 1 gdy którykolwiek z operandów jest prawdziwy.

W przypadku, gdy operandy operatorów logicznych **zawierają bity**, które przyjmują wartość **nieistotną (x lub z)**, a pozostałe **bity** są **wyłącznie zerami**, wtedy **wynik** operacji logicznej jest **nieistotny**.

Dla operatorów logicznych operandy typu reg oraz sieci są traktowane jako liczby bez znaku. Operandy typu real i integer mogą być liczbami ze znakiem.

81. OPERATORY PRZESUNIĘCIA BITOWEGO. OPERATORY BITOWE. OPERATORY REDUKCJI

81.1. Operatory bitowe

W odróżnieniu od operatorów logicznych, operatory bitowe zwracają wartość, która jest wektorem o rozmiarze równym rozmiarowi operandów lub rozmiarowi sygnału/zmiennej, do której wartość wyrażenia jest przypisywana - w zależności od tego, który rozmiar jest większy. Operatorami bitowymi w języku Verilog są:

- negacja bitowa (~),
- iloczyn bitowy (&),
- suma bitowa (|),
- bitowa różnica symetryczna XOR (^),
- XNOR ($\sim\wedge$ lub $\wedge\sim$ [takie samo działanie]).

Tab. 1.3. Tablica prawdy dla operatorów bitowych

| Operandy | | Wartość wyrażenia | | | |
|----------|---|-------------------|-------|-------|--------|
| a | b | a & b | a b | a ^ b | a ~^ b |
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 | 1 |
| 0 | x | 0 | x | x | x |
| x | 0 | 0 | x | x | x |
| x | x | x | x | x | x |
| 1 | x | x | 1 | x | x |
| x | 1 | x | 1 | x | x |

Operatory bitowe realizują poszczególne operacje na swoich operandach bit po bicie (na bitach o odpowiadającej sobie pozycji). Działanie operatorów bitowych szczegółowo ilustruje tabela 1.3. Z użyciem tych operatorów mieliśmy już do czynienia np. w przypadku opisu modułu z listingu 1.12 (moduł sumatora pełnego).

81.2. Operatory redukcji

Operatory redukcji są specjalnym przypadkiem operatorów bitowych. Są to operatory jednoargumentowe. W Verilogu operatorami redukcji są:

- redukcja AND (&),
- redukcja OR (|),
- redukcja NAND (~&),
- redukcja NOR (~|),
- redukcja XOR (^),
- redukcja XNOR (~^ lub ^~ [takie samo działanie]).

Operatory redukcji przekształcają wielobitowy operand do wartości jednobitowej. Na przykład dla operatora redukcji AND wykonywana jest operacja iloczynu bitowego (**AND**) na kolejnych bitach operandu, by w wyniku dać **wartość 1, gdy wszystkie bity operandu są jedynkami**, i 0 w przeciwnym przypadku.

Operatory redukcji od operatorów bitowych odróżniane są poprzez składnię (tylko jeden operand z prawej strony). W tabeli 1.4 pokazano przykładowe wartości liczbowe (w kodzie binarnym) oraz wynik zastosowania operatorów redukcji dla tych wartości.

Na listingu 1.22 przedstawiono opis bardzo prostego modułu, w którym zastosowano operatory redukcji w celu wyliczenia bitu parzystości 8-bitowego sygnału wejściowego oraz sygnału informującego o tym, że w sygnale wejściowym wszystkie bity mają wartość 1.

Tab. 1.4. Przykładowe wartości wyrażeń z operatorami redukcji

| Operand | Wartość wyrażenia | | |
|-----------|-------------------|-----------|-----------|
| a | &a | a | ^a |
| 0000 0000 | 0 | 0 | 0 |
| 1111 1111 | 1 | 1 | 0 |
| 1010 1101 | 0 | 1 | 1 |
| 1100 11zz | 0 | 1 | x |
| 1111 111x | x | 1 | x |

List. 1.22. Przykład zastosowania operatorów redukcji

```
module sprawdz_wejscie (input [7:0] in, output parity, all_ones);
    assign parity = ^ in;
    assign all_ones = & in;
endmodule
```

81.3. Operatory przesunięcia bitowego

Operatory przesunięcia są operatorami dwuargumentowymi. W języku Verilog występują cztery takie operatory:

1. przesunięcie bitowe w prawo (`>>`),
2. przesunięcie bitowe w lewo (`<<`),
3. przesunięcie bitowe ze znakiem w prawo (`>>>`),
4. przesunięcie bitowe ze znakiem w lewo (`<<<`).

Operatory te powodują przesunięcie w danym kierunku wszystkich bitów pierwszego operandu o liczbę pozycji określoną wartością drugiego operandu. Po wykonaniu przesunięcia brakujące bity wypełniane są zerami.

Wyjątek stanowi operator przesunięcia w prawo ze znakiem (`>>>`), dla którego brakujące pozycje z lewej strony wypełniane są **kopią bitu znaku (najstarszego bitu** lewego operandu).

Operator przesunięcia bitowego w lewo ze znakiem (`<<<`) działa identycznie jak zwykły operator `<<`

Na listingu 1.23 pokazano opis modułu realizującego operację mnożenia przez 10, który wykorzystuje m.in. operatory przesunięcia bitowego.

List. 1.23. Przykład zastosowania operatorów przesunięcia bitowego

```
module pomnoz_x10 (input [7:0] A, output [11:0] Y);  
    wire[8:0] Ax2 = A << 1;  
    wire[10:0] Ax8 = A << 3;  
    assign Y = Ax2 + Ax8;  
endmodule
```

82.1. OPERATOR WARUNKOWY. OPERATOR SKLEJANIA (KONKATENACJI)

Operator warunkowy

Operator warunkowy (? :) jest operatorem trójargumentowym, działającym podobnie jak instrukcja warunkowa if-then-else. Używany jest, w połączeniu z trzema operandami, w następujący sposób: wyrażenie_warunkowe ? wyrażenie_1 : wyrażenie_2.

Jeżeli wyrażenie_warunkowe jest prawdziwe (wartość różna od zera), to obliczana i zwracana jest wartość wyrażenie_1. W przeciwnym przypadku, gdy wartość wyrażenie_warunkowe jest fałszywa, obliczana i zwracana jest wartość wyrażenie_2.

Gdy wartość **wyrażenie_warunkowe** jest **nieistotna** (zawiera bity x i/lub z, a pozostałe bity są zerami), wówczas wartością całego wyrażenia jest **specyficzna wartość złożona z wartości obydwu wyrażeń wyrażenie_1 oraz wyrażenie_2** w następującym porządku bitowym:

jeżeli bity na tych samych pozycjach (o tych samych wagach) w obydwu wyrażeniach mają identyczną wartość, to zwracana jest ta wartość bitów (na danych pozycjach), jeżeli bity różnią się, to na tych pozycjach zwracana jest wartość x.

Wyrażenia z operatorem warunkowym mogą być zagnieźdzane, np. tak, jak to zilustrowano na listingu 1.24. Na tym listingu jest pokazany opis bardzo prostej jednostki arytmetyczno-logicznej wykonującej 5 podstawowych operacji arytmetycznych i bitowych.

List. 1.24. Przykład użycia zagnieźdzonego operatora warunkowego

```
module minIALU(input [7:0] a,b, input [2:0] op, output [7:0] result);  
    parameter ADD=3'h0,SUB=3'hi,AND=3'h2, OR=3'h3, XOR=31h4;  
    assign result = ((op == ADD)  
                    ? a+b  
                    : ((op== XOR)  
                        ? a^b  
                        : (a) ))));  
endmodule
```

Operator sklejania (konkatenacji)

Dzięki temu operatorowi możliwe jest złożenie (sklejenie) jednego lub kilku wyrażeń w celu uformowania wektora o większej liczbie bitów. Operator sklejania może być zastosowany zarówno po prawej, jak i po lewej stronie operatora przypisania (tak ciągłego, jak i proceduralnego). Użycie tego operatora polega na wylistowaniu wszystkich sklejanych wyrażeń przedzielonych znakiem przecinka i ujęcie ich w nawiasy klamrowe: {wyr_1, wyr_2, ... , wyr_n}. Sklejane wyrażenia mogą być dowolnymi wyrażeniami z wyjątkiem stałych, dla których nie jest podany ich rozmiar bitowy.

Pewnym **szczególnym** przypadkiem operatora **sklejania** jest **operator replikacji „{{}}”**. Dla tego operatora używa się dwóch par nawiasów klamrowych oraz liczby określającej, ile razy dana wartość ma być powtórzona: {liczba{wyr_1, wyr_2, ... , wyr_n}}.

Działanie operatora sklejania bardziej szczegółowo wyjaśnimy na przykładzie (List. 1.25)

List. 1.25. Fragment kodu ilustrujący użycie operatora sklejania

```
wire[3:0] a,b;  
wire[7:0] c,d;  
wire[11:0] e,f;  
  
assign c = {a, b};      // (1)  
assign e = {b,a,b};    // (2)  
assign f = {3 {a}};    // (3)  
assign b = {4{e==f}};  // (4)  
assign f = {a,d};      // (5)  
assign e = {2{1'b1,a,1'b0}}; // (6)  
assign {a, b} = d;     // (7)  
assign {a,b,f} = {e,d]+1; // (8)
```

W linii (1) po prawej stronie operatora przypisania, dzięki użyciu operatora sklejania, otrzymaliśmy wektor o rozmiarze równym sumie rozmiarów wektorów a i b. Czyli w tym przypadku 8. Tyle samo wynosi rozmiar wektora, do którego przypisujemy sklejenie wektorów a i b. Najstarsze bity wektora c (o numerach od 4 do 7) otrzymują wartość bitów wektora a, z kolei młodszym bitom wektora c nadawana jest wartość wektora b.

Podobna sytuacja ma miejsce w linii (2), z tym że 12-bitowemu wektorowi przypisujemy sklejenie trzech wektorów 4-bitowych.

W linii (3) zastosowaliśmy operator sklejenia z powtórzeniem (replikacją). Równoważny zapis tej linii byłby następujący:

```
assign f = {a,a,a}; // (3)
```

Podobnie linię (4) moglibyśmy zapisać jako:

```
assign b = {(e==f), (e==f),(e==f),(e==f)}; // (4)
```

W linii (4) wszystkim bitom wektora b jest nadawana wartość wyniku porównania (1 bit) wektorów e i f.

Przypadek w linii (5) jest analogiczny do tych omawianych już w liniach (1) i (2).

W kolejnym przypisaniu - linia (6) - w wyniku otrzymujemy wektor, którego bity mają następującą wartość: 1 a₃a₂a ja(/)l a[^]a₂ a ia(P> gdzie ai oznacza i-ty bit wektora a.

W liniach (7) i (8) mamy do czynienia z użyciem operatora sklejania również po lewej stronie przypisania.

82.2. PRIORYTET OPERATORÓW. ROZMIAR BITOWY WYRAŻENIA

Priorytet operatorów

W przypadku, gdy mamy do czynienia ze złożonymi wyrażeniami zawierającymi pewną liczbę wymienionych wyżej operatorów, istotny jest sposób, w jaki zostanie wyliczona wartość tego wyrażenia. Czyli ważna jest informacja, który operator zostanie wykonany jako pierwszy, a który jako następny itd. O kolejności wykonywania operatorów decyduje ich priorytet. Podczas tworzenia wyrażeń z operatorami znajomość ich priorytetów nie jest warunkiem koniecznym. Bez informacji o priorytetach można się zawsze obejść, stosując nawiasy okrągłe.

Jednak wiedza na temat priorytetów czasem bywa bardzo przydatna. Poniższa tabela przedstawia priorytety operatorów języka Verilog, **pierwszy wiersz ma priorytet najwyższy, ostatni - najniższy**. Operatory znajdujące się w tej samej linii mają ten sam priorytet

| Operator | Opis |
|-------------------|---|
| [] | Selekcja bitu, częściowa selekcja |
| () | nawias okrągły |
| ! ~ | negacja logiczna i bitowa |
| & ~& ~ ^ ~^ ~~ | operatorы redukcji |
| + - | jednoargumentowe operatorы zmiany znaku |
| { } | operator sklejania |
| ** | potęgowanie |
| * / % | operatorы arytmetyczne (dwuargumentowe) |
| + - | operatorы arytmetyczne (dwuargumentowe) |
| << >> <<< >>> | przesunięcia bitowe |
| < <= > >= | operatorы relacji arytmetycznych |
| == != === !== | operatorы porównania |
| & | iloczyn bitowy |
| ^ ~^ ~~ | XOR bitowy, XNOR bitowy |
| | suma bitowa |
| | suma logiczna |
| ? : | operator warunkowy (trójargumentowy) |

Wszystkie operatory w języku Verilog, z wyjątkiem operatora warunkowego, są łączne lewostronnie (w uproszczeniu: obliczane są od lewej strony do prawej). Jedynie operator warunkowy ma łączność prawostronną.

Rozmiar bitowy wyrażenia

Rozmiar (liczba bitów) wyrażenia jest zależna od rozmiaru operandów oraz typu operacji (zastosowanych operatorów). W ramach podsumowania zagadnień związanych z operatorami, w tabeli 1.6 przedstawiono rozmiar wyrażenia w zależności od typu operatora. Oznaczenie **L(i)** oznacza rozmiar (liczbę bitów) operandu i.

Tab. 1.6. Rozmiar bitowy wyrażenia

| Wyrażenie | Rozmiar | Komentarz |
|-----------|-----------------------------|-----------------------------|
| i OP j | $\max(L(i), L(j))$ | OP: |
| OPi | $L(i)$ | OP: |
| i OP j | 1 bit | OP: , !, , !, &&, , <, <, |
| i OP j | $L(i)$ | OP: >>, <<, **, >>>, <<< |
| i?j:k | $\max(L(j), L(k))$ | |
| {i...1} | $L(i) + \dots + L(j)$ | |
| {i(j..k)} | $i^k (L(j) + \dots + L(k))$ | |

Tabela 1.6