Univerza v Ljubljani
Fakulteta za elektrotehniko

Danjel Vončina

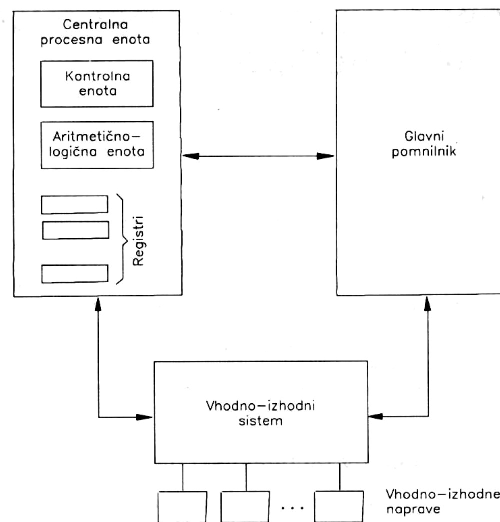
Digitalno procesiranje v mehatroniki I
2. del

Zapiski predavanj

Ljubljana, 2014

1. Von Neumannov računalniški model

Večina današnjih računalnikov (vključno z mikroprocesorskimi sistemi) je zgrajenih po vzoru von Neumannovega računalniškega modela. Imenujejo se po ameriškem matematiku Johnu von Neumannu (1903-1957), ki je leta 1945 napisal predlog za gradnjo novega računalnika. V tem predlogu je von Neumann uvedel idejo o računalniku s shranjenim programom, ki je bil bistveno drugačen kot tedaj že delujoči prvi elektronski računalnik ENIAC



Slika 1: Von Neumannov računalniški model

Na sliki 1 vidimo, da tipičen von Neumannov računalnik sestavljajo naslednje enote:

1. Centralna procesna enota (CPE). V von Neumannovem računalniku se večina dogajanja odvija v CPE. Sestavljajo jo: krmilna enota, aritmetična-logična enota (ALE) in registri.
2. Glavni pomnilnik: V njem so shranjeni ukazi in operandi, ki jih uporablja CPE. Sestavljen je iz pomnilniških besed. Vsaka pomnilniška beseda ima svoj naslov.
3. Vhodno-izhodni (V/I) sistem. V CPE in v glavnem pomnilniku je informacija shranjena v obliki, ki zunanjemu svetu ni dostopna. Vsak računalnik ima zato del, ki ga imenujemo vhodno-izhodni sistem in omogoča prenos informacij v računalnik in iz računalnika v zunanje enote (npr. tipkovnica, tiskalniki, prikazovalniki, magnetni diski, telefonske linije), ki pretvarjajo informacijo iz oblike, ki jo uporablja CPE, v obliko, ki je primerna za človeka ali kakega drugega uporabnika (npr. stroje, merilne instrumente).

V von Neumannovem računalniku je celotno dogajanje pod nadzorom CPE, ki jemlje iz glavnega pomnilnika ukaze in jih izvršuje. Obdelava enega programskega ukaza poteka v dveh korakih:

- branje in dekodiranje ukaza iz glavnega pomnilnika (fetch),

- izvajanje v prvem koraku prevzetega ukaza (execute).

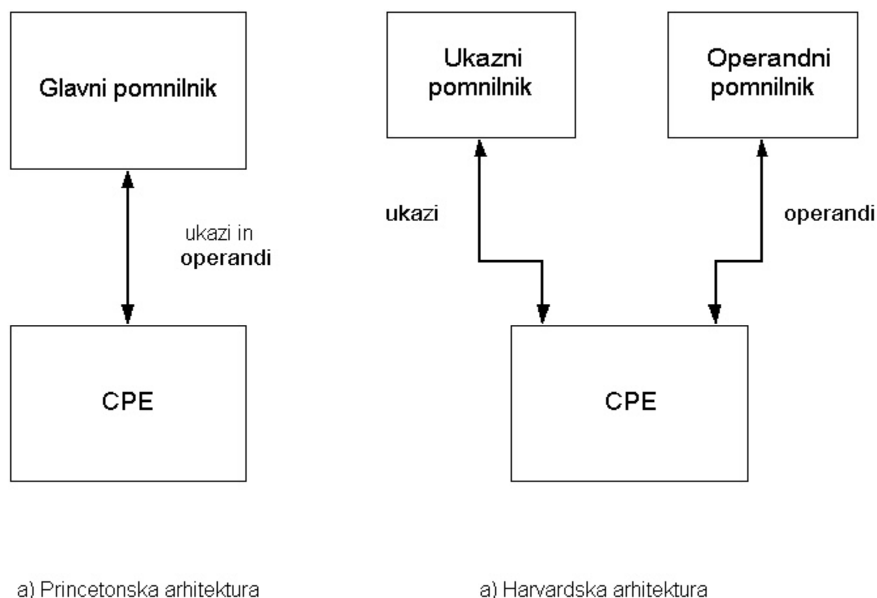
V von Neumannovem računalniku je glavni pomnilnik namenjen shranjevanju ukazov in operandov. Naredi samo tisto kar od njega zahteva CPE in V/I sistem. Zmogljivost računalnika je določena tudi s količino podatkov, ki se prenaša iz glavnega pomnilnika v CPE in obratno. Pot med CPE in glavnim pomnilnikom je ozko grlo, ki je znano tudi pod imenom von Neumannovo ozko grlo.

2. Harvardska arhitektura

Eden od načinov za razširitev tega ozkega grla je, da glavni pomnilnik razdelimo na dva dela. V prvem delu so ukazi, v drugem pa operandi. Branje ukaza in operanda lahko poteka istočasno. Če je CPE izdelana tako, da traja branje ukaza približno enako dolgo kot njegova izvedba (v večini operacij se izvaja branje ali pisanje enega operanda), potem dosežemo s tako razdeljenim pomnilnikom dvakrat večjo hitrost izvrševanja instrukcije. Računalnikom s tako razdeljenim glavnim pomnilnikom pravimo, da imajo Harvardsko arhitekturo. Ime so dobili od računalnikov Harvard Mark I do Mark IV, ki so imeli podobno razdeljen pomnilnik.

Računalnikom, ki nimajo razdeljenega glavnega pomnilnika pravimo tudi, da imajo Princetonsko arhitekturo. Ime so dobili po računalniku IAS, ki je bil izdelan v Princetonu. Ta arhitektura pa je inačica von Neumannove.

Primer obeh arhitektur vidimo na sliki 2.1.

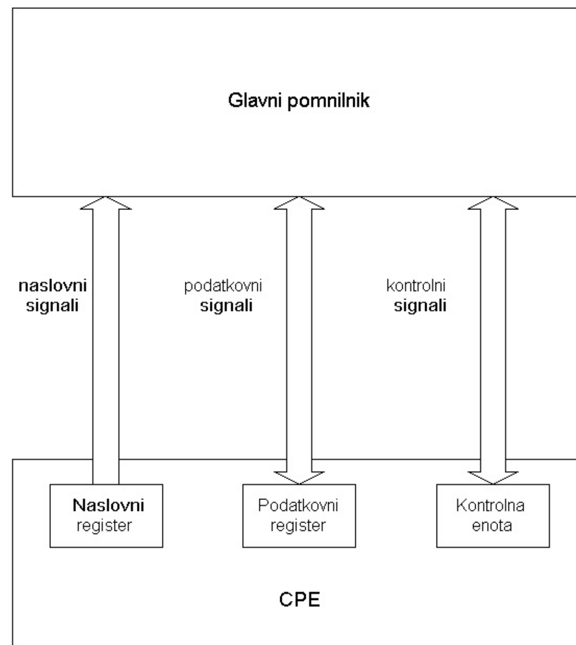


Slika 2.1: Princetonska in Harvardska arhitektura

Večina današnjih računalnikov uporablja Princetonsko arhitekturo, Harvardski pa so v manjšini. Podobno je tudi pri mikroprocesorjih. Harvardsko arhitekturo srečamo predvsem pri signalnih procesorjih in pri nekaterih RISC mikroprocesorjih.

Von Neumannovo ozko grlo lahko razširimo tudi drugače, npr. s predpomnilnikom ali s pomnilniškim prepletanjem. Ker pri večini računalnikov traja izvrševanje ukaza dalj kot pa traja prevzem ukaza, lahko s temi načini organizacije spomina dosežemo boljše rezultate. Harvardska arhitektura je razmeroma pogosta na nivoju predpomnilnika, kadar je ta vgrajen v CPE. Z ločenim predpomnilnikom za ukaze in operande je možno izkoristiti njene prednosti ne da bi se spremenila zgradba glavnega predpomnilnika.

Pri obeh vrstah von Neumannove arhitekture je glavni pomnilnik videti kot enodimenzionalno zaporedje pomnilniških besed, od katerih ima vsaka svoj enoličen naslov (pri Harvardski velja to za oba pomnilnika). Pomnilniška beseda, ki ji včasih pravimo tudi pomnilniška lokacija, je običajno sestavljena iz več pomnilniških celic. Vsaka pomnilniška celica je sposobna hraniti en bit informacije. Število celic v besedi imenujemo dolžina pomnilniške besede.
V von Neumannovem računalniku je povezava med CPE in glavnim pomnilnikom najpogosteje realizirana tako, kot je prikazano na sliki 2.2. CPE je sestavljena iz dveh registrov (naslovni in podatkovni) ter iz krmilne enote. V naslovnem registru se nahaja naslov pomnilniške besede, do katere se želi dostopiti, v podatkovni register pa se vpiše iz pomnilnika prebrana vrednost. Povezave med naslovnim registrom in glavnim pomnilnikom potekajo preko naslovnih signalov, med glavnim pomnilnikom in podatkovnim registrom pa potekajo preko podatkovnih signalov. Smer prenosa podatkov določajo krmilni signali (branje/pisanje).



Slika 2.2: Povezava med CPE in glavnim pomnilnikom

Pri tem velja omeniti še eno značilnost von Neumannovih računalnikov. Iz vsebine pomnilniške besede v splošnem ni mogoče videti, ali je v njej ukaz ali operand. Pri Harvardski arhitekturi je to trivialno vprašanje.

3. Predpomnilnik (Cache)

Delovanje von Neumannovega računalnika je s stališča glavnega pomnilnika videti kot zaporedje naslovov, preko katerih je zahtevan dostop do podatkov. Vsak program tvori svoje zaporedje naslovov. Značilno za ta računalnik je, da zaporedje naslovov ni naključno-verjetnost za pojav enih naslovov je večja kot verjetnost pojava drugih. Če merimo pogostost pojava posameznih naslovov v N naslovov dolgem zaporedju vidimo, da se pri praktično vsakem programu nekateri naslovi pojavijo večkrat. Poleg tega si naslovi zelo pogosto sledijo po določenem vzorcu. Pojav imenujemo lokalnost pomnilniških dostopov. Različni programi seveda tvorijo različna zaporedja naslovov in zato nimajo enake lokalnosti. Je pa lokalnost vedno prisotna in je posledica načina delovanja von Neumannovega računalnika ter načina pisanja programov. Tako poznamo prostorsko in časovno lokalnost.

1. *Prostorska lokalnost*

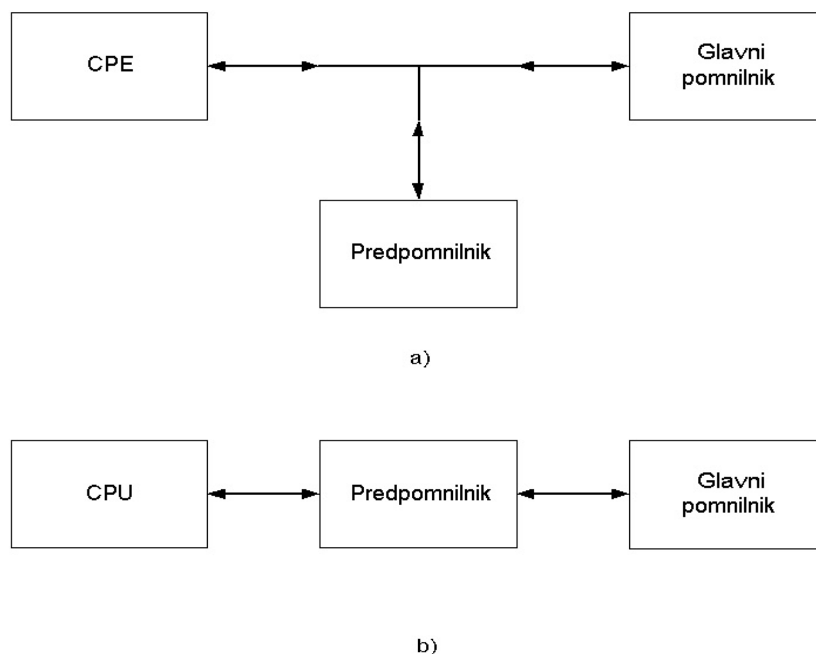
S tem označujemo pojav, ki nam podaja verjetnost, da se po naslovu A(i), pojavi naslednji naslov A(i+n) blizu A(i). Vzroki za prostorsko lokalnost so v glavnem trije:

- če ni skoka, CPU jemlje ukaze enega za drugim po naraščajočih naslovih
- v programih pogosto nastopajo podatkovne strukture kot so polja, ki se največkrat uporabljajo zaporedoma po naraščajočih in padajočih indeksih
- program je pogosto razdeljen na procedure ali podprograme

2. *Časovna lokalnost*

S tem označujemo pojav, ko program ob času *t* pogosto tvori naslove, ki jih je tvoril malo pred trenutkom *t*, in ki jih bo tvoril tudi nekoliko po trenutku *t*. Glavni vzrok za ta pojav so programske zanke in začasne spremenljivke. Tako se zaradi zank isto zaporedje ukazov in s tem isti naslovi, velikokrat ponovijo, prav tako pa se zaradi začasnih spremenljivk naslovi nekaterih operandov pojavijo večkrat.

Lokalnost pomnilniških dostopov lahko izkoristimo pri gradnji računalnika. Osnovna dva načina za izkoriščanje lokalnosti sta predpomnilnik (cache) in navidezni pomnilnik (virtual memory).



Slika 3.1: Dva načina za vključitev predpomnilnika v Von Neumannov računalnik

Predpomnilnik (cache) je majhen in hiter pomnilnik, ki ga priključimo na CPE na dva možna načina. Na sliki 3.1 vidimo, da lahko priključimo predpomnilnik paralelno k glavnemu pomnilniku, ali pa med CPE in glavni pomnilnik. V drugem primeru ima CPE dostop do

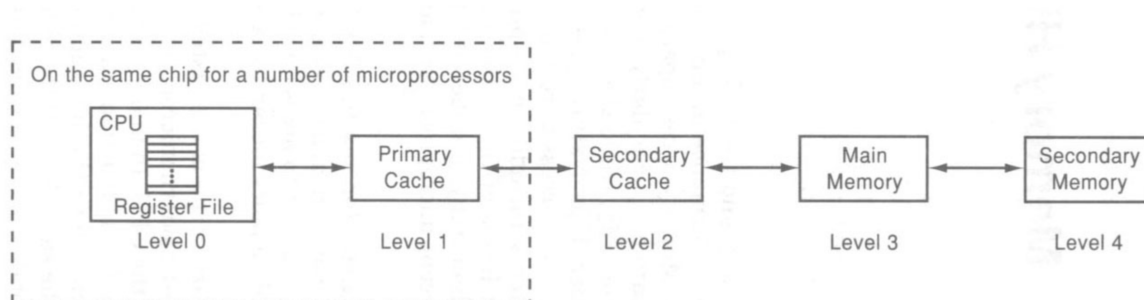
glavnega pomnilnika samo preko predpomnilnika, v prvem pa ima neposreden dostop do obeh. Razlika med obema oblikama je običajno majhna.

Namesto enega predpomnilnika pa imamo lahko en predpomnilnik za ukaze in drugi predpomnilnik za operande. V tem primeru dobimo inačico Harvardske arhitekture. Ta način je razmeroma pogost, še posebno kadar je vgrajen v CPE.

Hitrost predpomnilnika je običajno 5-10 krat večja od hitrosti glavnega pomnilnika. Če gledamo iz CPE je videti, kot da bi bil celoten glavni pomnilnik skoraj tako hiter kot predpomnilnik. Predpomnilnik je običajno veliko manjši od glavnega pomnilnika, je seveda dosti cenejši kot enako hiter glavni pomnilnik. V tem je tudi osnovni razlog za uporabo predpomnilnikov. Z uporabniške strani je delovanje predpomnilnika nevidno. Razlika je le v hitrosti. Sestavlja ga več predpomnilniških blokov. Vsak blok pa ima svoj kontrolni del, v katerem je informacija, ki poleg drugega vsebuje pomnilniški naslov besed, ki so trenutno v bloku. Pri vsakem dostopu do pomnilnika mora predpomnilnik vedeti ali je z naslovom podana beseda v predpomnilniku ali ne. Poleg tega mora imeti vgrajene algoritme, ki določajo, kdaj se neka beseda prenese iz glavnega pomnilnika v predpomnilnik in kdaj se prenese nazaj vanj. Temu pravimo polnilna ali zamenjevalna strategija predpomnilnika.

3.1 Primarni in sekundarni predpomnilnik

Povezave med pomnilnikom in CPE igrajo zelo pomembno vlogo pri hitrosti delovanja mikrokrmilnika. Dostopi do spomina so običajno počasnejši kot so notranje operacije v CPE, še posebno tam, kjer je CPE ločena od glavnega pomnilnika, ki je lahko razdeljen na več zunanjih čipov. Zato imajo novejši računalniški sistemi razdeljen spomin na več nivojev. Ponavadi so ti nivoji določeni od 0 – 4. To kaže slika 3.2.



Slika 3.2: Nivojska zgradba spomina

Nivo 0.

To je najbližji register s katerim razpolaga CPE, je del CPE in do njega ima CPE tudi najlažji dostop.

Nivo 1.

To je primarni predpomnilnik, ki je tudi najhitrejši pomnilnik, ki se nahaja zunaj CPE. V novejših izvedbah mikroprocesorjev se običajno nahaja na istem čipu, skupaj s CPE. V veliki večini mikroprocesorskih sistemov je primarni predpomnilnik razdeljen na dva dela-torej ima Harvardsko arhitekturo. Njegova velikost pa je omejena s ceno.

Nivo 2

To je sekundarni predpomnilnik. Nahaja se zunaj CPE čipa in je večji od primarnega predpomnilnika. Ta predpomnilnik ni deljen in vsebuje tako ukaze kot operande skupaj.

Nivo 3

Glavni pomnilnik. V njem so shranjeni ukazi in operandi tekočega programa.

Nivo 4

Sekundarni pomnilnik. Je veliko večji od glavnega.

V veliki večini RISC mikroprocesorjev in v nekaterih CISC mikroprocesorjih je primarni predpomnilnik razdeljen na ukazni in operandni predpomnilnik – Harvardska arhitektura, medtem, ko sta sekundarni predpomnilnik in glavni pomnilnik združena.

(RISC - reduced instruction set computer, CISC - complex instruction set computer)

4. Mikrokrmilniki s Harvardsko arhitekturo

4.1 PIC (angl. Peripheral Interface Controller)

To so mikrokrmilniki podjetja Microchip Technology Inc. Delijo se na tri družine, in sicer po velikosti pomnilniške besede (word):

1. Base-Line: 12-bitna instrukcijska beseda, 8-bitno podatkovno vodilo - PIC 12xxx
2. Mid-Range: 14-bitna instrukcijska beseda, 8-bitno podatkovno vodilo - PIC 16xxx
3. High-End: 16-bitna instrukcijska beseda, 8-bitno podatkovno vodilo - PIC 18xxx
4. Enhanced: 16-bitna instrukcijska beseda, 8-bitno podatkovno vodilo - PIC 18xxx
5. dsPIC30F: 24- bitna instrukcijska beseda, 16-bitno podatkovno vodilo

V manj zahtevnih aplikacijah se najpogosteje uporablja družina mikrokrmilnikov srednjega razreda 16xxx z dolžino instrukcijske besede 14 bitov. V teh mikrokrmilnikih je na nivoju predpomnilnika uporabljena Harvardska arhitektura. Pri tej arhitekturi sta programski in podatkovni pomnilnik fizično ločena. Na ta način se poenostavlja in pospešuje dostop do podatkov. PIC mikrokrmilnik so v splošnem sestavljeni iz jedra, perifernih enot in posebnih enot.

V naslednjem poglavju je opisana zgradba 8 - bitnega mikrokrmilnika PIC18FXX20.

4.2 Mikrokrmilnik PIC18FXX20

PIC18FXX2 se uvrščajo v razred zmogljivejših 8-bitnih mikrokrmilnikov proizvajalca Microchip. Osnovni podatki o obsegu spominskih enot nekaterih izvedb mikrokrmilnikov iz te družine so prikazani v spodnji tabeli:

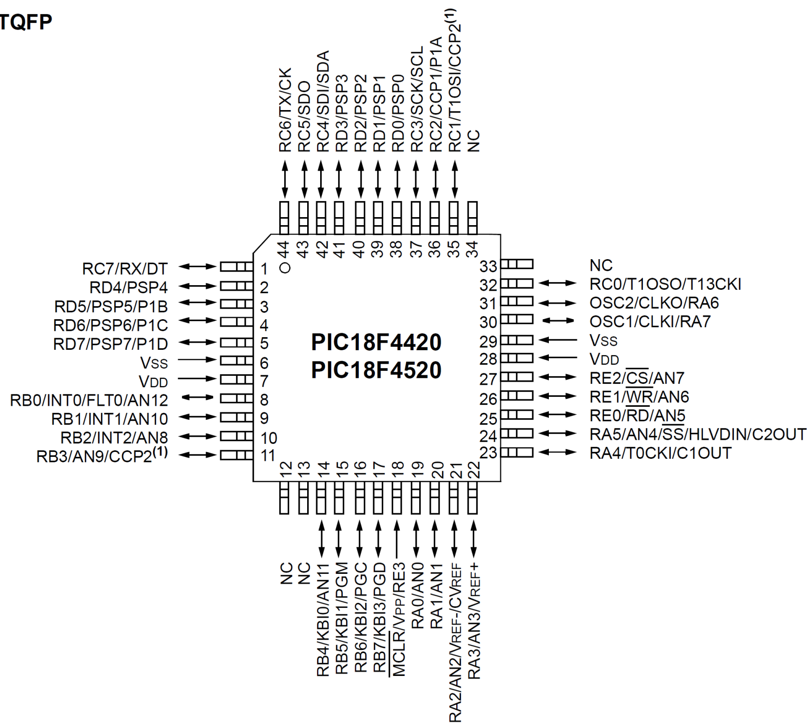
Tabela 4.1

Device	Program Memory		Data Memory		I/O	10-Bit A/D (ch)	CCP/ ECCP (PWM)	MSSP		EUSART	Comp.	Timers 8/16-Bit
	Flash (bytes)	# Single-Word Instructions	SRAM (bytes)	EEPROM (bytes)				SPI	Master I^2C™			
PIC18F2420	16K	8192	768	256	25	10	2/0	Y	Y	1	2	1/3
PIC18F2520	32K	16384	1536	256	25	10	2/0	Y	Y	1	2	1/3
PIC18F4420	16K	8192	768	256	36	13	1/1	Y	Y	1	2	1/3
PIC18F4520	32K	16384	1536	256	36	13	1/1	Y	Y	1	2	1/3

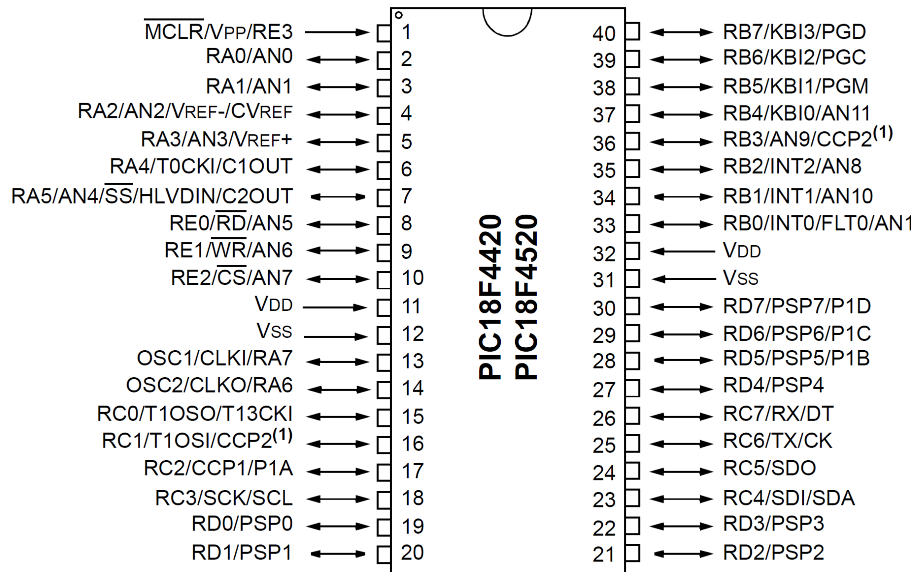
© 2008 Microchip Technology Inc. DS39631E-page 1

Posamezne izvedenke mikrokrmilnikov PIC18FXX20 se razlikujejo po številu priključkov oz. v številu perifernih enot in zato jih tudi vgrajujejo v različna ohišja. Na spodnji sliki je prikazani izvedbi PIC18F4520 v TQFP ohišju s 44 priključki in v klasičnem PDIP ohišju s 40 priključki.

44-pin TQFP



40-Pin PDIP



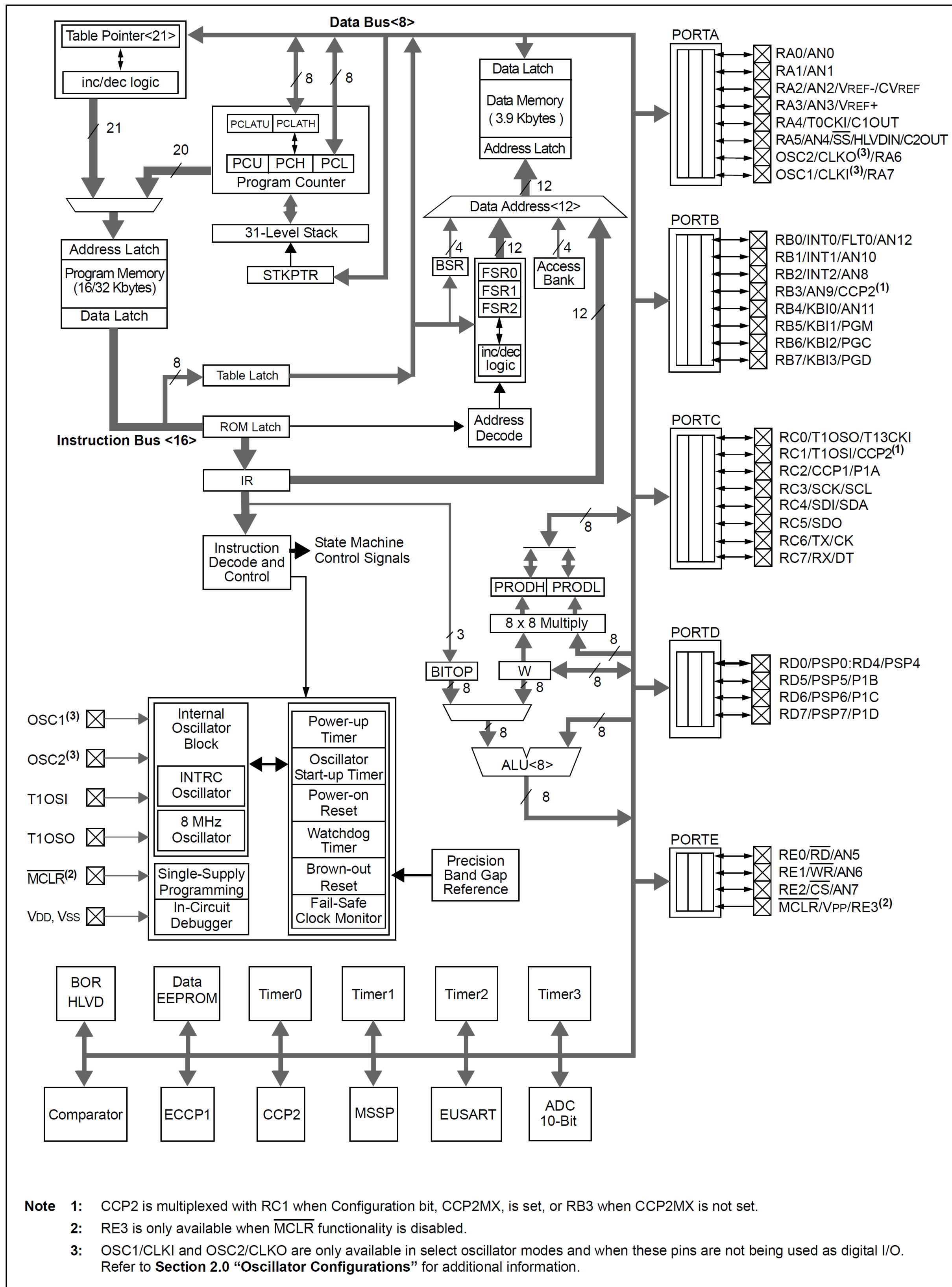
Slika 4.1: Mikrokrmilnika PIC18F4420 in PIC18F4520 v PLCC in PDIP ohišju

Najpomembnejši podatki o zmogljivosti in o vgrajenih perifernih enotah v posameznih izvedenkah mikrokrmilnikov PIC18FXX20 so zbrani v spodnji tabeli:

Tabela: 4.2

Features	PIC18F2420	PIC18F2520	PIC18F4420	PIC18F4520
Operating Frequency	DC – 40 MHz	DC – 40 MHz	DC – 40 MHz	DC – 40 MHz
Program Memory (Bytes)	16384	32768	16384	32768
Program Memory (Instructions)	8192	16384	8192	16384
Data Memory (Bytes)	768	1536	768	1536
Data EEPROM Memory (Bytes)	256	256	256	256
Interrupt Sources	19	19	20	20
I/O Ports	Ports A, B, C, (E)	Ports A, B, C, (E)	Ports A, B, C, D, E	Ports A, B, C, D, E
Timers	4	4	4	4
Capture/Compare/PWM Modules	2	2	1	1
Enhanced Capture/Compare/PWM Modules	0	0	1	1
Serial Communications	MSSP, Enhanced USART	MSSP, Enhanced USART	MSSP, Enhanced USART	MSSP, Enhanced USART
Parallel Communications (PSP)	No	No	Yes	Yes
10-Bit Analog-to-Digital Module	10 Input Channels	10 Input Channels	13 Input Channels	13 Input Channels
Resets (and Delays)	POR, BOR, RESET Instruction, Stack Full, Stack Underflow (PWRT, OST), $\overline{\text{MCLR}}$ (optional), WDT	POR, BOR, RESET Instruction, Stack Full, Stack Underflow (PWRT, OST), $\overline{\text{MCLR}}$ (optional), WDT	POR, BOR, RESET Instruction, Stack Full, Stack Underflow (PWRT, OST), $\overline{\text{MCLR}}$ (optional), WDT	POR, BOR, RESET Instruction, Stack Full, Stack Underflow (PWRT, OST), $\overline{\text{MCLR}}$ (optional), WDT
Programmable High/Low-Voltage Detect	Yes	Yes	Yes	Yes
Programmable Brown-out Reset	Yes	Yes	Yes	Yes
Instruction Set	75 Instructions; 83 with Extended Instruction Set Enabled	75 Instructions; 83 with Extended Instruction Set Enabled	75 Instructions; 83 with Extended Instruction Set Enabled	75 Instructions; 83 with Extended Instruction Set Enabled
Packages	28-Pin SPDIP 28-Pin SOIC 28-Pin QFN	28-Pin SPDIP 28-Pin SOIC 28-Pin QFN	40-Pin PDIP 44-Pin QFN 44-Pin TQFP	40-Pin PDIP 44-Pin QFN 44-Pin TQFP

Blokovna shema mikrokrmilnika PIC18F4520 je prikazana na spodnji sliki:



Slika 4.2: Blokovna shema mikrokrmilnika PIC18F4520

4.2.1 Organizacija spomina

Spominski prostor je razdeljen v tri medsebojno ločene enote:

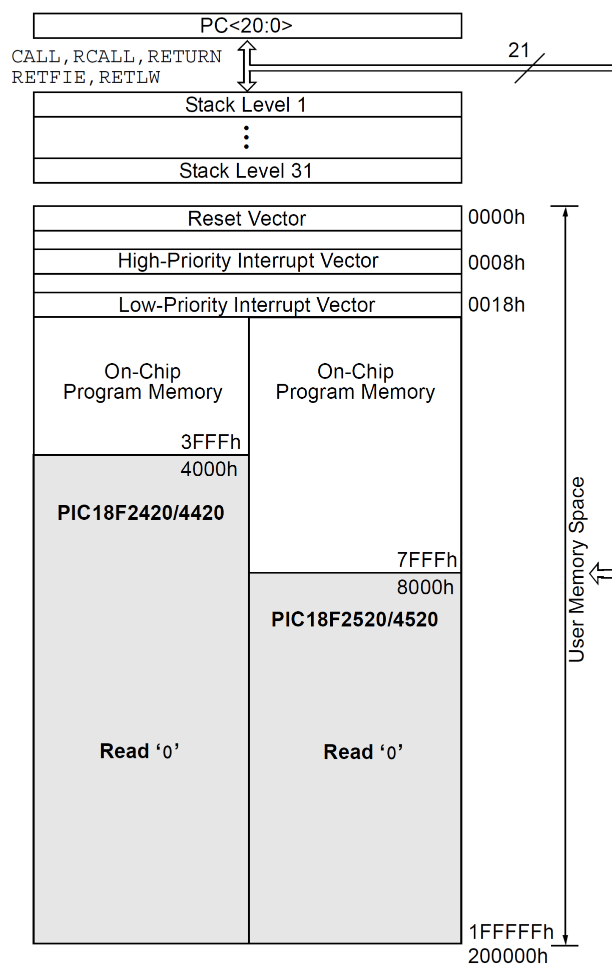
- programski spomin,
- podatkovni RAM spomin in
- EEPROM spomin za shranjevanje podatkov

Podatkovni in programski spomin imata ločeni vodili, ki omogočata neodvisen dostop do njunih vsebin.

4.2.1.1 Organizacija programskega spomina

Programski števec je 21 bitni in tako omogoča naslavljanje spominskega prostora v obsegu 2 Mbajt-ov. Pri morebitnem naslavljanju neuporabljenega dela naslovnega prostora dobimo vrednost 0, kar ustreza instrukciji NOP.

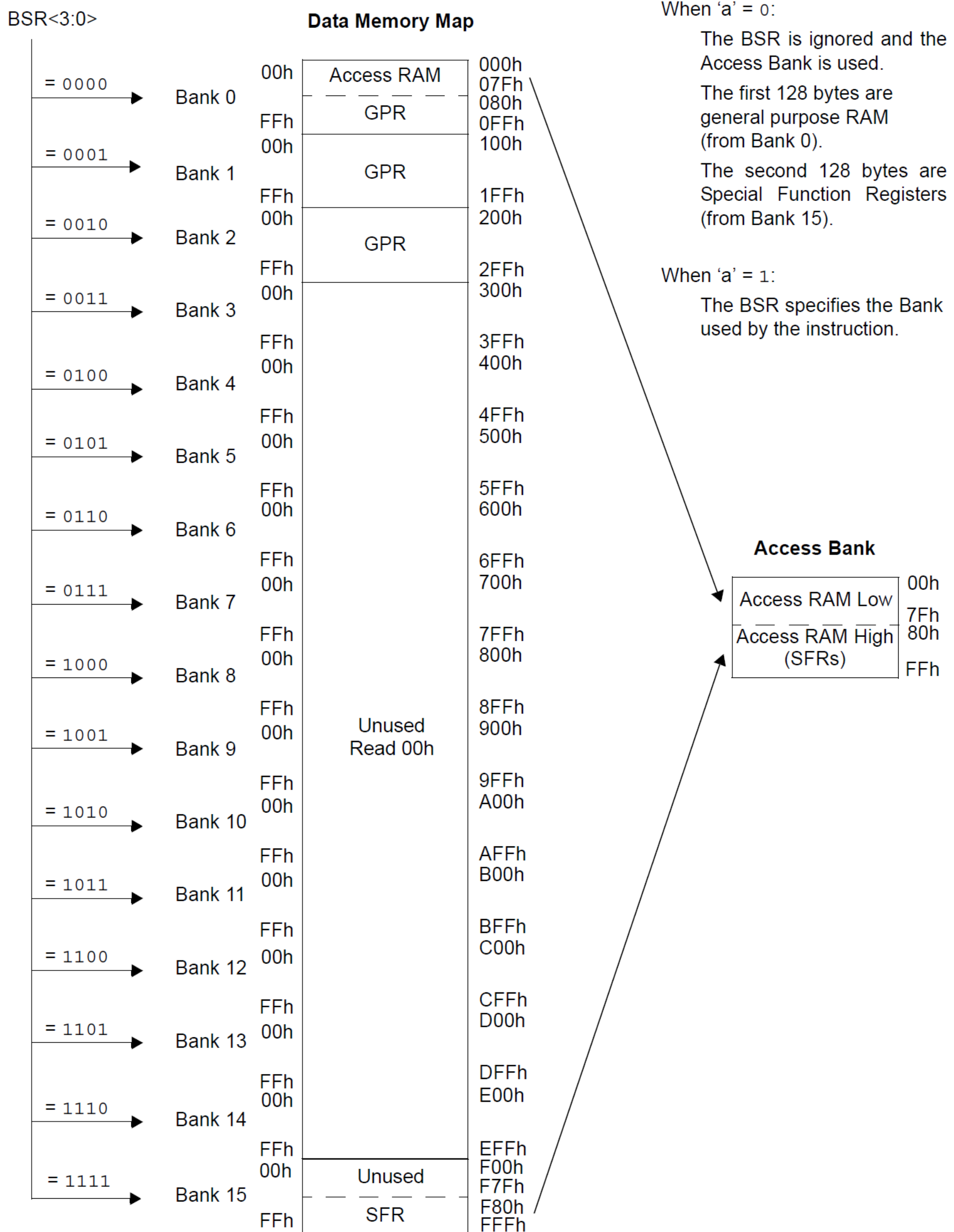
PIC18F4520 ima vgrajenih 32 Kbajt-ov »bliskovnega« EEPROM spomina, ki omogoča shranjevanje 16K enobesednih instrukcij. Reset vektor je na lokaciji 0000h, medtem ko sta dva prekinitvena vektorja na lokacijah 0008h in 0018h. Na sliki 4.3 je prikazana razporeditev programskega spominskega prostora.



Slika 4.3: Razporeditev programskega spominskega prostora

4.2.1.2 Organizacija podatkovnega spomina

Za shranjevanje podatkov imajo mikrokrmilniki PIC18FXX20 vgrajen statični RAM. Vsak register v podatkovnem spominu ima 12-bitni naslov, kar skupaj omogoča shranjevanje 4096 bajtov podatkov. Zgradba podatkovnega spomina pri mikrokrmilnikih PIC18FXX20 je prikazana na sliki 4.4.



Slika 4.4: Zgradba podatkovnega spomina mikrokrmilnika PIC18F4520

Celoten podatkovni spomin je razdeljen na 16 strani (ang. bank) in vsaka stran vsebuje 256 bajtov. Spodnji štirje biti registra BSR določajo stran, na katero želimo dostopati. Zgornji trije biti tega registra niso pomembni in ne sodelujejo pri naslavljanju strani.

V podatkovnem spominu imamo ti. posebne funkcijske registre (ang. **S**pecial **F**unction **R**egister - SFR) in splošno namenske registre (ang. **G**eneral **P**urpose **R**egister - GPR). SFR registri se uporabljajo za krmiljenje in za ugotavljanje stanja mikrokrmilnika oz. perifernih funkcij. GPR registri pa se v glavnem uporabljajo za shranjevanje podatkov.

SFR registri se nahajajo na zadnji strani (15) in sicer od zadnje lokacije (0xFFF) na tej strani proti vedno nižjim vrednostim naslova (Slika 4.5). Neuporabljen del spomina, ki ga ne zasedajo SFR registri, lahko uporabljamo kot GPR registre. GPR registri so razporejeni v spominu od prve lokacije na strani 0 naprej. Če beremo neuporabljene lokacije, dobimo vedno vrednost 0.

Do celotnega spominskega prostora lahko dostopamo posredno ali neposredno. Pri neposrednem naslavljanju moramo ustrezno postaviti bite v BSR registru. Pri posrednem naslavljanju moramo uporabiti "Izbirni" register (ang. **F**ile **S**elect **R**egister - FSRn) in ustrezni operand, ki je zapisan v registru INDFn (ang. INDirect File – INDF) . Vsak FSR vsebuje 12 bitni naslov, ki ga lahko uporabimo za dostop do lokacije v spominu brez predhodne izbire strani.

Arhitektura mikrokrmilnika in nabor ukazov omogočata izvajanje operacij na vseh straneh. To dosežemo s posrednim naslavljanjem ali z uporabo ukaza MOVFF. Slednja je dvobesedna instrukcija, ki se izvrši v dveh ciklih, in prenese vsebino iz enega registra v drugega.

Address	Name	Address	Name	Address	Name	Address	Name
FFFh	TOSU	FDFh	INDF2[3]	FBFh	CCPR1H	F9Fh	IPR1
FFEh	TOSH	FDEh	POSTINC2[3]	FBEh	CCPR1L	F9Eh	PIR1
FFDh	TOSL	FDDh	POSTDEC2[3]	FBDh	CCP1CON	F9Dh	PIE1
FFCh	STKPTR	FDCh	PREINC2[3]	FBCh	CCPR2H	F9Ch	—
FFBh	PCLATU	FDBh	PLUSW2[3]	FBBh	CCPR2L	F9Bh	—
FFAh	PCLATH	FDAh	FSR2H	FBAh	CCP2CON	F9Ah	—
FF9h	PCL	FD9h	FSR2L	FB9h	—	F99h	—
FF8h	TBLPTRU	FD8h	STATUS	FB8h	—	F98h	—
FF7h	TBLPTRH	FD7h	TMR0H	FB7h	—	F97h	—
FF6h	TBLPTRL	FD6h	TMR0L	FB6h	—	F96h	TRISE[2]
FF5h	TABLAT	FD5h	T0CON	FB5h	—	F95h	TRISD[2]
FF4h	PRODH	FD4h	—	FB4h	—	F94h	TRISC
FF3h	PRODL	FD3h	OSCCON	FB3h	TMR3H	F93h	TRISB
FF2h	INTCON	FD2h	LVDCON	FB2h	TMR3L	F92h	TRISA
FF1h	INTCON2	FD1h	WDTCON	FB1h	T3CON	F91h	—
FF0h	INTCON3	FD0h	RCON	FB0h	—	F90h	—
FEFh	INDF0[3]	FCFh	TMR1H	FAFh	SPBRG	F8Fh	—
FEEh	POSTINC0[3]	FCEh	TMR1L	FAEh	RCREG	F8Eh	—
FEDh	POSTDEC0[3]	FCDh	T1CON	FADh	TXREG	F8Dh	LATE[2]
FECh	PREINC0[3]	FCCh	TMR2	FACh	TXSTA	F8Ch	LATD[2]
FEBh	PLUSW0[3]	FCBh	PR2	FABh	RCSTA	F8Bh	LATC
FEAh	FSR0H	FCAh	T2CON	FAAh	—	F8Ah	LATB
FE9h	FSR0L	FC9h	SSPBUF	FA9h	EEADR	F89h	LATA
FE8h	WREG	FC8h	SSPADD	FA8h	EEDATA	F88h	—
FE7h	INDF1[3]	FC7h	SSPSTAT	FA7h	EECON2	F87h	—
FE6h	POSTINC1[3]	FC6h	SSPCON1	FA6h	EECON1	F86h	—
FE5h	POSTDEC1[3]	FC5h	SSPCON2	FA5h	—	F85h	—
FE4h	PREINC1[3]	FC4h	ADRESH	FA4h	—	F84h	PORTE[2]
FE3h	PLUSW1[3]	FC3h	ADRESL	FA3h	—	F83h	PORTD[2]
FE2h	FSR1H	FC2h	ADCON0	FA2h	IPR2	F82h	PORTC
FE1h	FSR1L	FC1h	ADCON1	FA1h	PIR2	F81h	PORTB
FE0h	BSR	FC0h	—	FA0h	PIE2	F80h	PORTA

Note 1: Unimplemented registers are read as '0'.
2: This register is not available on PIC18F2X2 devices.
3: This is not a physical register.

Slika 4.5: Seznam SFR registrov

File Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on POR, BOR	Details on page:
TOSU	—	—	—	Top-of-Stack upper Byte (TOS<20:16>)					---0 0000	37
TOSH	Top-of-Stack High Byte (TOS<15:8>)								0000 0000	37
TOSL	Top-of-Stack Low Byte (TOS<7:0>)								0000 0000	37
STKPTR	STKFUL	STKUNF	—	Return Stack Pointer					00-0 0000	38
PCLATU	—	—	—	Holding Register for PC<20:16>					---0 0000	39
PCLATH	Holding Register for PC<15:8>								0000 0000	39
PCL	PC Low Byte (PC<7:0>)								0000 0000	39
TBLPTRU	—	—	bit21[(2)]	Program Memory Table Pointer Upper Byte (TBLPTR<20:16>)					--00 0000	58
TBLPTRH	Program Memory Table Pointer High Byte (TBLPTR<15:8>)								0000 0000	58
TBLPTRL	Program Memory Table Pointer Low Byte (TBLPTR<7:0>)								0000 0000	58
TABLAT	Program Memory Table Latch								0000 0000	58
PRODH	Product Register High Byte								xxxx xxxx	71
PRODL	Product Register Low Byte								xxxx xxxx	71
INTCON	GIE/GIEH	PEIE/GIEL	TMR0IE	INT0IE	RBIE	TMR0IF	INT0IF	RBIF	0000 000x	75
INTCON2	RBPU	INTEDG0	INTEDG1	INTEDG2	—	TMR0IP	—	RBIP	1111 -1-1	76
INTCON3	INT2IP	INT1IP	—	INT2IE	INT1IE	—	INT2IF	INT1IF	11-0 0-00	77
INDF0	Uses contents of FSR0 to address data memory - value of FSR0 not changed (not a physical register)								n/a	50
POSTINC0	Uses contents of FSR0 to address data memory - value of FSR0 post-incremented (not a physical register)								n/a	50
POSTDEC0	Uses contents of FSR0 to address data memory - value of FSR0 post-decremented (not a physical register)								n/a	50
PREINC0	Uses contents of FSR0 to address data memory - value of FSR0 pre-incremented (not a physical register)								n/a	50
PLUSW0	Uses contents of FSR0 to address data memory - value of FSR0 (not a physical register). Offset by value in WREG.								n/a	50
FSR0H	—	—	—	—	Indirect Data Memory Address Pointer 0 High Byte				---- 0000	50
FSR0L	Indirect Data Memory Address Pointer 0 Low Byte								xxxx xxxx	50
WREG	Working Register								xxxx xxxx	n/a
INDF1	Uses contents of FSR1 to address data memory - value of FSR1 not changed (not a physical register)								n/a	50
POSTINC1	Uses contents of FSR1 to address data memory - value of FSR1 post-incremented (not a physical register)								n/a	50
POSTDEC1	Uses contents of FSR1 to address data memory - value of FSR1 post-decremented (not a physical register)								n/a	50
PREINC1	Uses contents of FSR1 to address data memory - value of FSR1 pre-incremented (not a physical register)								n/a	50
PLUSW1	Uses contents of FSR1 to address data memory - value of FSR1 (not a physical register). Offset by value in WREG.								n/a	50
FSR1H	—	—	—	—	Indirect Data Memory Address Pointer 1 High Byte				---- 0000	50
FSR1L	Indirect Data Memory Address Pointer 1 Low Byte								xxxx xxxx	50
BSR	—	—	—	—	Bank Select Register				---- 0000	49
INDF2	Uses contents of FSR2 to address data memory - value of FSR2 not changed (not a physical register)								n/a	50
POSTINC2	Uses contents of FSR2 to address data memory - value of FSR2 post-incremented (not a physical register)								n/a	50
POSTDEC2	Uses contents of FSR2 to address data memory - value of FSR2 post-decremented (not a physical register)								n/a	50
PREINC2	Uses contents of FSR2 to address data memory - value of FSR2 pre-incremented (not a physical register)								n/a	50
PLUSW2	Uses contents of FSR2 to address data memory - value of FSR2 (not a physical register). Offset by value in WREG.								n/a	50
FSR2H	—	—	—	—	Indirect Data Memory Address Pointer 2 High Byte				---- 0000	50
FSR2L	Indirect Data Memory Address Pointer 2 Low Byte								xxxx xxxx	50
STATUS	—	—	—	N	OV	Z	DC	C	---x xxxx	52
TMR0H	Timer0 Register High Byte								0000 0000	105
TMR0L	Timer0 Register Low Byte								xxxx xxxx	105
T0CON	TMR0ON	T08BIT	T0CS	T0SE	PSA	T0PS2	T0PS1	T0PS0	1111 1111	103

Legend: x = unknown, u = unchanged, - = unimplemented, q = value depends on condition

Note 1: RA6 and associated bits are configured as port pins in RCIO and ECIO Oscillator mode only and read '0' in all other Oscillator modes.

2: Bit 21 of the TBLPTRU allows access to the device configuration bits.

3: These registers and bits are reserved on the PIC18F2X2 devices; always maintain these clear.

Slika 4.6: Vsebina SFR registrov (a)

File Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on POR, BOR	Details on page:
OSCCON	—	—	—	—	—	—	—	SCS	---- ---0	21
LVDCON	—	—	IRVST	LVDEN	LVDL3	LVDL2	LVDL1	LVDL0	--00 0101	191
WDTCON	—	—	—	—	—	—	—	SWDTE	---- ---0	203
RCON	IPEN	—	—	$\overline{\text{RI}}$	$\overline{\text{TO}}$	$\overline{\text{PD}}$	$\overline{\text{POR}}$	$\overline{\text{BOR}}$	0--1 11qq	53, 28, 84
TMR1H	Timer1 Register High Byte								xxxx xxxx	107
TMR1L	Timer1 Register Low Byte								xxxx xxxx	107
T1CON	RD16	—	T1CKPS1	T1CKPS0	T1OSCEN	$\overline{\text{T1SYNC}}$	TMR1CS	TMR1ON	0-00 0000	107
TMR2	Timer2 Register								0000 0000	111
PR2	Timer2 Period Register								1111 1111	112
T2CON	—	TOUTPS3	TOUTPS2	TOUTPS1	TOUTPS0	TMR2ON	T2CKPS1	T2CKPS0	-000 0000	111
SSPBUF	SSP Receive Buffer/Transmit Register								xxxx xxxx	125
SSPADD	SSP Address Register in I^2C Slave mode. SSP Baud Rate Reload Register in I^2C Master mode.								0000 0000	134
SSPSTAT	SMP	CKE	D/$\overline{\text{A}}$	P	S	R/$\overline{\text{W}}$	UA	BF	0000 0000	126
SSPCON1	WCOL	SSPOV	SSPEN	CKP	SSPM3	SSPM2	SSPM1	SSPM0	0000 0000	127
SSPCON2	GCEN	ACKSTAT	ACKDT	ACKEN	RCEN	PEN	RSEN	SEN	0000 0000	137
ADRESH	A/D Result Register High Byte								xxxx xxxx	187,188
ADRESL	A/D Result Register Low Byte								xxxx xxxx	187,188
ADCON0	ADCS1	ADCS0	CHS2	CHS1	CHS0	GO/$\overline{\text{DONE}}$	—	ADON	0000 00-0	181
ADCON1	ADFM	ADCS2	—	—	PCFG3	PCFG2	PCFG1	PCFG0	00-- 0000	182
CCPR1H	Capture/Compare/PWM Register1 High Byte								xxxx xxxx	121, 123
CCPR1L	Capture/Compare/PWM Register1 Low Byte								xxxx xxxx	121, 123
CCP1CON	—	—	DC1B1	DC1B0	CCP1M3	CCP1M2	CCP1M1	CCP1M0	--00 0000	117
CCPR2H	Capture/Compare/PWM Register2 High Byte								xxxx xxxx	121, 123
CCPR2L	Capture/Compare/PWM Register2 Low Byte								xxxx xxxx	121, 123
CCP2CON	—	—	DC2B1	DC2B0	CCP2M3	CCP2M2	CCP2M1	CCP2M0	--00 0000	117
TMR3H	Timer3 Register High Byte								xxxx xxxx	113
TMR3L	Timer3 Register Low Byte								xxxx xxxx	113
T3CON	RD16	T3CCP2	T3CKPS1	T3CKPS0	T3CCP1	$\overline{\text{T3SYNC}}$	TMR3CS	TMR3ON	0000 0000	113
SPBRG	USART1 Baud Rate Generator								0000 0000	168
RCREG	USART1 Receive Register								0000 0000	175, 178, 180
TXREG	USART1 Transmit Register								0000 0000	173, 176, 179
TXSTA	CSRC	TX9	TXEN	SYNC	—	BRGH	TRMT	TX9D	0000 -010	166
RCSTA	SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D	0000 000x	167
EEADR	Data EEPROM Address Register								0000 0000	65, 69
EEDATA	Data EEPROM Data Register								0000 0000	69
EECON2	Data EEPROM Control Register 2 (not a physical register)								---- ----	65, 69
EECON1	EEPGD	CFGS	—	FREE	WRERR	WREN	WR	RD	xx-0 x000	66

Legend: x = unknown, u = unchanged, - = unimplemented, q = value depends on condition

Note 1: RA6 and associated bits are configured as port pins in RCIO and ECIO Oscillator mode only and read '0' in all other Oscillator modes.

2: Bit 21 of the TBLPTRU allows access to the device configuration bits.

3: These registers and bits are reserved on the PIC18F2X2 devices; always maintain these clear.

Slika 4.6: Vsebina SFR registrov (b)

File Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on POR, BOR	Details on page:
IPR2	—	—	—	EEIP	BCLIP	LVDIP	TMR3IP	CCP2IP	---1 1111	83
PIR2	—	—	—	EEIF	BCLIF	LVDIF	TMR3IF	CCP2IF	---0 0000	79
PIE2	—	—	—	EEIE	BCLIE	LVDIE	TMR3IE	CCP2IE	---0 0000	81
IPR1	PSPIP(3)	ADIP	RCIP	TXIP	SSPIP	CCP1IP	TMR2IP	TMR1IP	1111 1111	82
PIR1	PSPIF(3)	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	0000 0000	78
PIE1	PSPIE(3)	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	0000 0000	80
TRISE(3)	IBF	OBF	IBOV	PSPMODE	—	Data Direction bits for PORTE			0000 -111	98
TRISD(3)	Data Direction Control Register for PORTD								1111 1111	96
TRISC	Data Direction Control Register for PORTC								1111 1111	93
TRISB	Data Direction Control Register for PORTB								1111 1111	90
TRISA	—	TRISA6(1)	Data Direction Control Register for PORTA						-111 1111	87
LATE(3)	—	—	—	—	—	Read PORTE Data Latch, Write PORTE Data Latch			---- -xxx	99
LATD(3)	Read PORTD Data Latch, Write PORTD Data Latch								xxxx xxxx	95
LATC	Read PORTC Data Latch, Write PORTC Data Latch								xxxx xxxx	93
LATB	Read PORTB Data Latch, Write PORTB Data Latch								xxxx xxxx	90
LATA	—	LATA6(1)	Read PORTA Data Latch, Write PORTA Data Latch(1)						-xxx xxxx	87
PORTE(3)	Read PORTE pins, Write PORTE Data Latch								---- -000	99
PORTD(3)	Read PORTD pins, Write PORTD Data Latch								xxxx xxxx	95
PORTC	Read PORTC pins, Write PORTC Data Latch								xxxx xxxx	93
PORTB	Read PORTB pins, Write PORTB Data Latch								xxxx xxxx	90
PORTA	—	RA6(1)	Read PORTA pins, Write PORTA Data Latch(1)						-x0x 0000	87

Legend: x = unknown, u = unchanged, - = unimplemented, q = value depends on condition
Note 1: RA6 and associated bits are configured as port pins in RCIO and ECIO Oscillator mode only and read '0' in all other Oscillator modes.
2: Bit 21 of the TBLPTRU allows access to the device configuration bits.
3: These registers and bits are reserved on the PIC18F2X2 devices; always maintain these clear.

Slika 4.6: Vsebina SFR registrov (c)

4.3 Zgradba programa v zbirnem jeziku (Assembly Language)

V zbirnem jeziku uporabljamo ukaze, ki so simbolični zapisi strojne kode. Vsak ukaz je strukturiran tako, da omogoča neposreden prevod v strojno kodo.

Program v zbirnem jeziku je sestavljen iz več vrst stavkov:

- **direktive ali navodila**, ki krmilijo sam zbirnik. Predvsem gre za določanje vhodov, izhodov in umeščanje podatkov v spominu. Vsak program v zbirniku se mora začeti z direktivo START in končati z direktivo END. Vsi ukazi, ki so napisani za direktivo END, se ne izvajajo.
- **ukazi** v zbirnem jeziku. Mikrokrmilniki PIC18 imajo v naboru 77 različnih ukazov.
- **komentarji.** Uporabljamo dva načina pisanja komentarjev. Prvi način uporabljamo za razlago posameznega ukaza oz. direktive. Drug način pisanja komentarjev pa uporabljamo za razlago vloge skupine inštrukcij oz direktiv.

Za pisanje programa lahko uporabljamo katerikoli urejevalnik besedila v ASCII formatu. Vsaka vrstica v programu je lahko sestavljena iz štirih delov, ki si sledijo po vrsti od leve proti desni:

- ime stavka (ang. label),
- ukaz (ang. mnemonic),
- operand(i) in
- komentar.

Vrstni red je pomemben. Ime stavka se mora začeti v prvi koloni. Ukaz se lahko začne v drugi koloni ali pa v naslednjih kolonah. Med posameznimi deli ukaza mora biti vsaj en presledek. V eni vrstici je lahko največ 256 znakov.

4.3.1 Ime stavka (label)

Ime stavka se mora začeti v prvi koloni. Po imenu mora biti presledek ali skok tabulatorja ali podpičje ali znak za konec vrstice (EOL). Začeti se mora s črkovnim znakom ali s podčrtajem, vsebuje pa lahko alfanumerične znake podčrtaj in vprašaj. Vsebuje lahko največ 32 znakov. V osnovi je ime občutljivo na male in velike črke. Če pri definiciji imen uporabljamo podpičje, se smatra kot operator in ne kot del imena stavka.

Primeri:

```
zanka       addwf 0x20,F,A
_ponovi     addlw 0x03
c?xy        andlw 0x7F
maj2_junij  bsf   0x07,4
```

V naslednjih primerih so imena stavkov uporabljena nepravilno:

```
 jevecji    btfsc 0x15,7       ;Ime se začne v drugi koloni
3ali5       clrf  0x16,A       ;Ime se začne s številko
tri-stiri   cpfsgt 0x14,A      ;Ime vsebuje nedovoljen znak (-)
```

4.3.2 Polje za pisanje ukaza (Mnemonic)

V tem polju lahko pišemo ukaz v zbirniku ali pa zbirniško direktivo, ki se mora začeti v drugi koloni ali pa za imenom ukaza. Med imenom in ukazom mora biti dvopičje ali pa vsaj en presledek.

Primeri:

```
Napaka      equ   0        ;equ je zbirniška direktiva
            goto  start    ;goto je ukaz
zanka:      incf  0x20,W,A    ;incf  je ukaz
```

4.3.3 Polje za pisanje operanda

Ukazu sledi operand, zbirniški direktivi pa argument. Tudi med operandom in ukazom mora biti vsaj en presledek. Če ukaz vsebuje več operandov, jih ločimo z vejico.

Primeri:

```
cpfseq      0x20,A      ;0x20 je operand
movff       0x30,0x65   ;0x30 in 0x65 sta operanda
```

4.3.4 Polje za pisanje komentarja

Komentar pišemo po lastni presoji. Vedno se začne s podpičjem. Vsi znaki za podpičjem do konca vrstice se v programu ne upoštevajo. V naslednjih dveh primerih sta prikazana dva načina pisanja komentarja:

```
     decf  0x20,F,A     ; zmanjšaj vrednost za ena
; Celotna vrstica je komentar
```

4.4 Zbirniške direktive

Zbirniške direktive se na prvi pogled ne razlikujejo od običajnih ukazov. Večina direktiv ima drugo vlogo kot jo imajo ukazi. Direktive v bistvu določajo kako naj se izvajajo zbirniški ukazi. Poleg tega določajo konstante in rezervirajo spominski prostor za dinamične spremenljivke. Vsak zbirnik ima svoj nabor direktiv in ni nujno, da so med seboj enake. V MPASM imamo pet različnih vrst direktiv:

- ***Krmilne direktive.*** Te direktive omogočajo izvajanje pogojenih programskih sekvenc.
- ***Podatkovne direktive.*** Te direktive rezervirajo spominski prostor in omogočajo poimenovanje posameznih spominskih lokacij.
- ***Izpisne direktive.*** Te direktive omogočajo določitev formata izpisne datoteke (Listing File). Omogočajo določitev naslova, označevanje strani in ostale krmilne funkcije izpisa.
- ***Makroji.*** Te direktive omogočajo izvedbo in lokacijo podatkov znotraj makrojev.

– **Objektne direktive**. Te direktive se uporabljajo samo pri ustvarjanju objektne datoteke.

4.4.1 Krmilne direktive

Najpogosteje uporabljene krmilne direktive so zbrane v tabeli 4.4.1.

Tabela 4.4.1 Krmilne direktive v MPASM

Direktiva	Opis	Sintaksa
#DEFINE	Definiraj tekstovno nadomestilo vrednosti	#define <ime>[<vrednost>]
END	Konec programskega bloka	end

```
#define <name>[<string>]  (originalno)
#define <ime>[<vrednost>]
```

Ta direktiva definira tekstovni nadomestek vrednosti (stringa). Kadarkoli je zaznano <ime> v zbrirnem jeziku, se ga nadomesti z <vrednostjo> (s stringom)

Primer:

```
        #define     lenght 20
        #define     config 0x17,7,A
        #define     sum3(x,y,z) (x + y + z)
                .
                .
Test            dw  sum3(1,length,200)     ;postavi (1 + 20 + 200) na to lokacijo
                bsf config                 ;postavi bit 7 registra 0x17 na 1
```

```
end
```

Direktiva `end` določa konec programa. Primer:

```
List p=xxxx         ;xxxx je oznaka mikrokrmilnika (npr. PIC18F4520)
-                   ;izvršni del programa
-
end                 ;konec programa
```

4.4.2 Podatkovne direktive

Podatkovne direktive v MPASM so zbrane v tabeli 4.4.2.

Direktiva	Opis
CBLOCK	Določitev niza konstant
CONSTANT	Deklaracije simbola konstante
ENDC	Konec niza konstant
EQU	Določitev konstante v zbirniku

Direktiva CBLOCK določi listo poimenovanih konstant. Vsaka konstanta je na zapisana v spominu na lokaciji, ki je za ena višja od predhodne. Lista se konča z direktivo ENDC. Primer:

```
cblock 0x50
test1
test2
test3 : 2
test4
endc        ;Konstante so na naslovih od 0x50 do 0x54
```

Direktiva CONSTANT določi simbol, ki se nato uporablja v zbirniških ukazih. Primer:

```
constant      duty_cycle = .50
```

Ukaz določa, da se bo uporabila vrednost 50_D na vseh mestih, kjer je zapisano ime »duty_cycle«.

Direktiva EQU definira konstante. Kadarkoli se ime konstante uporabi v programskem ukazu, jo bo zbirnik nadomestil z vsebino. Primer:

```
Prav  equ 1
Narobe equ 0
Stiri equ 4
```

4.5 Predloga za pisanje programa v zbirniku

Uporabniški program mora biti napisan tako, da se izvede neposredno po izvedbi RESET-a. Pri tem se moramo držati naslednjega pravila:

```
            org   0x0000
            goto  start
            org   0x08
            goto  prek1           ;Prekinitveni vektor z višjo prioriteto (opcija)

            org   0x18
            goto  prek2           ;Prekinitveni vektor z nižjo prioriteto (opcija)

start       ....
            ....                  ;Uporabniški program, ki se izvaja ciklično
zanka1      movlw b'10101100'
            movwf PORTC
            movff sprem1,sprem2
            decf  Stevec
            btfsc status,Z
            goto  naprej
            movlw .50
            movwf Stevec
            btg   PORTB,2
naprej      goto  zanka

prek1       nop                   ;Prekinitveni podprogram 1
            retfie

prek2       nop                   ;Prekinitveni podprogram 2
            retfie

            end
```

4.6 Vrste naslavljanja

PIC18F4520 omogoča naslednje vrste naslavljanja:

- neposredno naslavljanje registrov (register direct)
- takojšnje (immediate)
- bitno (bit-direct)
- posredno (indirect)

Kot smo videli v poglavju 4.5, uporabljamo v zbirniku različne direktive, ki uporabniku omogočajo poimenovanje spominskih lokacij. Tako je zbirniški program bolj pregleden in lažje berljiv. V naslednjih primerih so za ponekod za lažje razumevanje uporabljena imena lokacij oz. simboli.

4.6.1 Neposredno naslavljanje registrov

PIC18 uporablja 8-bitno vrednost za določitev podatkovnega registra kot operanda. Register se lahko nahaja na prvi strani (access bank) ali na drugih straneh podatkovnega spomina. V slednjem moramo določiti tudi vsebino BSR registra. Poglejmo nekaj primerov:

```
movwf 0x1A,BANKED
```

Pri tem ukazu se kopira vsebina delovnega registra W v spomin na naslov 0x1A in sicer na strani, ki je določena z vrednostjo v BSR registru. Oznaka BANKED pa sporoča zbirniku, da mora pri določitvi naslova podatka vključiti še vrednost BSR registra.

```
movwf 0x45,A
```

Pri tem ukazu se vsebina delovnega registra prenese na lokacijo 0x45 na strani 0 v podatkovnem spominu (access bank).

```
movff reg1,reg2
```

Pri tem ukazu se vsebina registra reg1 prenese v register reg2. Pri tem se vrednost BSR registra ne upošteva.

4.6.2 Takojšnje naslavljanje

Pri tem načinu naslavljanja je operand že del ukaza in ni potrebe po dostopu do spomina. Vrednost, ki jo prenašamo, se imenuje »*literal*«. Poglejmo nekaj primerov:

```
addlw 0x20
```

Pri tem ukazu se vrednost 20_H prišteje vrednosti v delovnem registru W. Dobljena vsota je prav tako v delovnem registru W.

```
movlw 0x15
```

Pri tem ukazu se vrednost 15_H prenese v delovni register W.

```
movlb .3
```

Ukaz vpiše decimalno vrednost 3 v spodnje štiri bite BSR registra. Na ta način se določi stran 3 v BSR registru.

4.6.3 Neposredno bitno naslavljanje

PIC18 ima v naboru 5 ukazov, ki neposreno določajo stanja bitov.

```
bcf PORTB,3,A
```

Ta ukaz postavi bit 3 v registru PORTB na nič.

```
bsf PORTA,4,A
```

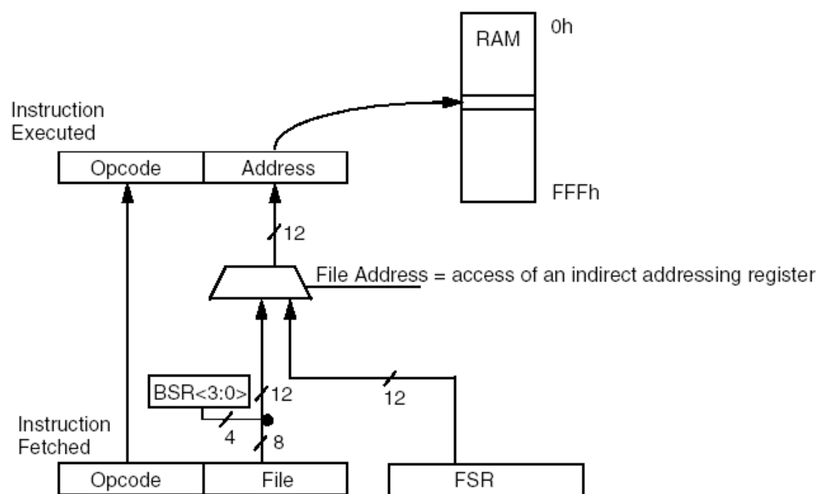
Ta ukaz pa postavi bit 4 v registru PORTA na ena.

4.6.4 Posredno naslavljanje z uporabo FSR in INDF registrov

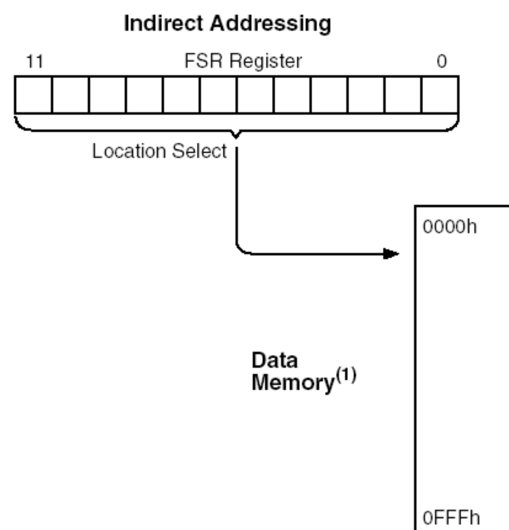
Pri posrednem naslavljanju naslov podatka, ki je zapisan v instrukciji, ni konstanta. Pri tem načinu naslavljanja uporabimo FSR register kot kazalec, ki kaže na določeno lokacijo, iz katere želimo brati ali pa vanjo vpisovati. Ker je vsebina tega vektorja v RAM-u, mu lahko

programsko spremenimo vsebino. To je zelo priročno predvsem pri delu s tabelami. Slika 4.7 kaže operacijo s posrednim naslavljanjem. Prikazan je princip naslavljanja lokacije v spominu, ki jo določa vsebina FSR registra.

Posredno naslavljanje je možno ob uporabi enega od INDF registrov. Vsaka instrukcija, ki uporablja INDF register omogoča dostop do registra oz. lokacije v podatkovnem spominu, ki jo kaže kazalec v FSR registru. Branje samega registra INDF (vsebina FSR = 0) bo omogočala branje vsebine na naslovu 00h. Vpis v INDF register je enakovreden operaciji NOP. FSR register torej vsebuje 12-bitni naslov, ki je prikazan na sliki 4.8



Slika 4.7: Posredno naslavljanje



Slika 4.8: Vloga FSR registra pri posrednem naslavljanju.

INDF je pravzaprav navidezen tip registra. Naslavljanje INDFn "registra" pomeni naslavljanje registra, katerega naslov vsebuje FSRn register (FSRn je torej kazalec). Spodnji primer kaže uporabo posrednega naslavljanja za brisanje RAM-a na strani 1 od lokacije 100h do 1FFh z minimalnim številom ukazov.

```
            LFSR FSR0, 0x100  ; Naloži register
NEXT        CLRF POSTINC0     ; Briši INDF in povečaj števec za 1
            BTFSS FSR0H, 1    ; Ali je na strani 1 (Bank1) vse zbrisano?
            GOTO NEXT         ; Ne, briši naslednjo lokacijo
CONTINUE                      ; Da, nadaljuj
```

Za posredno naslavljanje podatkovnega spomina v celotnem obsegu 4096 bajtov imamo na razpolago tri 12-bitne registre. Za shranjevanje 12-bitnega naslova potrebujemo dva osembitna registra in sicer:

1. FSR0 tvorita FSR0H in FSR0L
2. FSR1 tvorita FSR1H in FSR1L in
3. FSR2 tvorita FSR2H in FSR2L

Poleg tega imamo še registre INDF0, INDF1 in INDF2, ki pa fizično ne obstajajo. Branje oz. pisanje teh registrov omogoči posredno naslavljanje. Naslov je določen z zapisom v pripadajočem FSR registru. Če z instrukcijo vpišemo vrednost v INDF0, se ta vrednost zapiše na naslov, ki ga vsebujeta bajta FSR0H in FSR0L. Ravno tako pri branju registra INDF1 dobimo vrednost z naslova, ki ga tvorita bajta FSR1H in FSR1L. Treba je še omeniti, da posredna uporaba registrov INDF0, INDF1 in INDF2, ne vpliva na stanje bitov v statusnem registru (podobno kot operacija NOP).

Vsakemu FSR registru pripada INDF register, poleg njega pa še štirje naslovi registrov.Ti registri določajo kako se bo spreminjala vsebina FSR registrov pri izvajanju operacij s posrednim naslavljanjem. Ko dostopamo do podatka v enem od petih INDFn lokacij, se naslov FSRn registra:

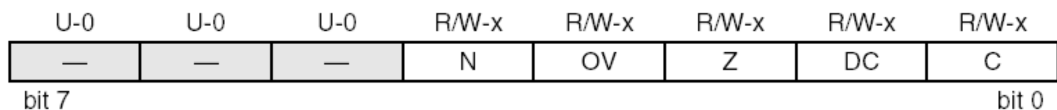
- ne spremeni (INDFn)
- avtomatsko zmanjša po operaciji (post-decrement) - POSTDECn
- avtomatsko poveča po operaciji (post-increment) - POSTINCn
- avtomatsko poveča pred operacijo (pre-increment) - PREINCn
- se uporabi vsebino WREG kot konstantni odmik (offset) FSRn registra po posrednem dostopu (no change) - PLUSWn

Če uporabimo možnost avtomatskega zmanjševanja ali povečevanja naslova, rezultat ne vpliva na bite statusnega registra. Če npr. vrednost FSR postane nič, se Z bit ne postavi. Pri avtomatskem povečevanju ali zmanjševanju vsebine FSR registra sodeluje vseh 12 bitov. Tako lahko FSRn registre dodatno uporabimo tudi kot kazalce sklada.

4.7 Statusni register

Statusni register, ki je prikazan na sliki 4.9, omogoča spremljanje stanja (rezultata) po opravljeni aritmetični-logični operaciji. Na splošno je statusni register lahko ciljni register za katerokoli instrukcij. V tem primeru so biti Z, DC C, OV in N bit onemgočeni. Ti biti se postavijo ali brišejo v skladu z vgrajeno logiko. Tako npr. rezultat pri operaciji s statusnim registrom kot ciljnim registrom ni vedno v skladu s pričakovanji. Npr. ukaz `CLRF STATUS` izbriše le zgornje štiri bite rgestra in postavi Z bitna 1. Stanje po operaciji je torej: 000n n1nnn, kjer n označuje nespremenjeno stanje. Proizvajalec priporoča, da se za spreminjanje vsebine statusnega registra uporablja le ukaze BCF, BSF, SWAPF, MOVFF in MOVWF. Te instrukcije ne vplivajo na stanja bitov Z, C, DC, OV in N.

Pomen bitov v statusnem registru je pojasnjen na sliki 4.9.



U-0	U-0	U-0	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
—	—	—	N	OV	Z	DC	C
bit 7							bit 0

bit 7-5 **Unimplemented:** Read as '0'

bit 4 **N:** Negative bit
This bit is used for signed arithmetic (2's complement). It indicates whether the result was negative (ALU MSB = 1).
1 = Result was negative
0 = Result was positive

bit 3 **OV:** Overflow bit
This bit is used for signed arithmetic (2's complement). It indicates an overflow of the 7-bit magnitude, which causes the sign bit (bit7) to change state.
1 = Overflow occurred for signed arithmetic (in this arithmetic operation)
0 = No overflow occurred

bit 2 **Z:** Zero bit
1 = The result of an arithmetic or logic operation is zero
0 = The result of an arithmetic or logic operation is not zero

bit 1 **DC:** Digit carry/$\overline{\text{borrow}}$ bit
For `ADDWF, ADDLW, SUBLW,` and `SUBWF` instructions
1 = A carry-out from the 4th low order bit of the result occurred
0 = No carry-out from the 4th low order bit of the result

Note: For $\overline{\text{borrow}}$, the polarity is reversed. A subtraction is executed by adding the two's complement of the second operand. For rotate (`RRF, RLF`) instructions, this bit is loaded with either the bit 4 or bit 3 of the source register.

bit 0 **C:** Carry/$\overline{\text{borrow}}$ bit
For `ADDWF, ADDLW, SUBLW,` and `SUBWF` instructions
1 = A carry-out from the Most Significant bit of the result occurred
0 = No carry-out from the Most Significant bit of the result occurred

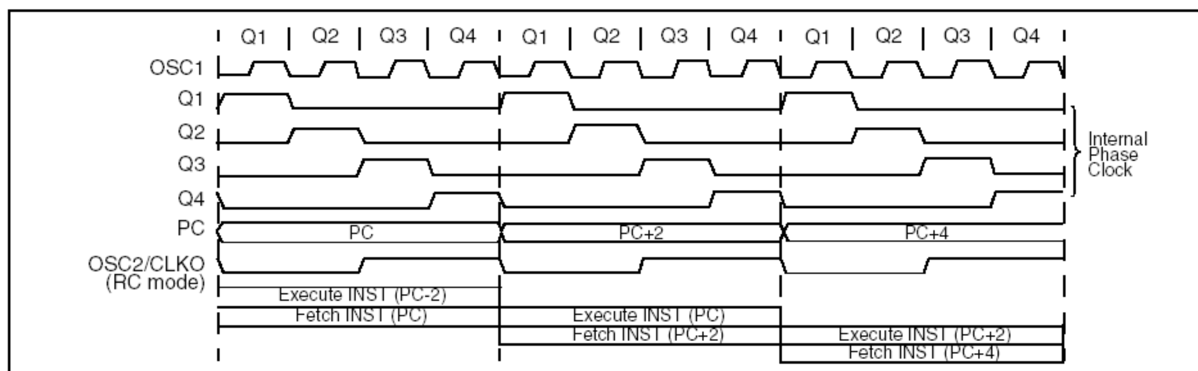
Note: For $\overline{\text{borrow}}$, the polarity is reversed. A subtraction is executed by adding the two's complement of the second operand. For rotate (`RRF, RLF`) instructions, this bit is loaded with either the high or low order bit of the source register.

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
- n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

Slika 4.9: Statusni register

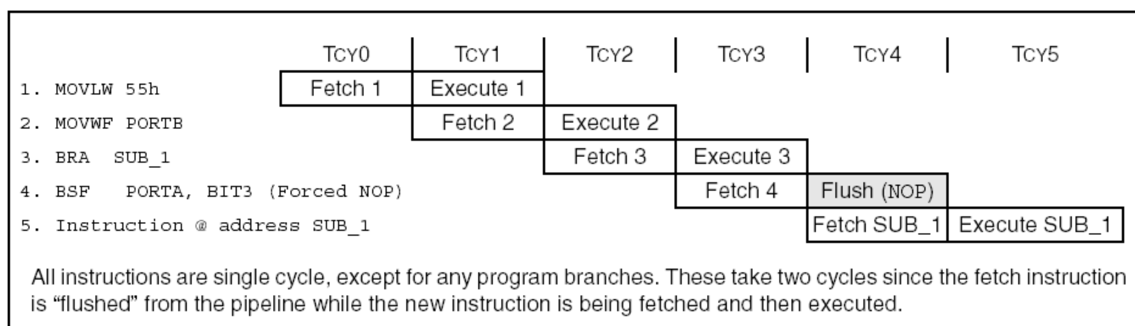
4.8 Izvajanje instrukcij (ang. Instruction flow-Pipelining)

Urin signal, ki je na vhodu mikrokrmilnika, se najprej deli s štiri. Tako dobimo štiri, medsebojno neprekrivajoče se takte Q1, Q2, Q3 in Q4. Programski števec poveča svojo vrednost ob vsakokratnem nastopu impulza Q1. V tem trenutku se začne zajem instrukcije iz programskega števca in se ob nastopu impulza Q4 prenese v instrukcijski register. Instrukcija se dekodira in izvede v naslednjem nizu impulzov od Q1 do Q4. Na spodnji sliki 4.10 je prikazan potek impulzov Q1 do Q4 in način izvajanja instrukcij.



Slika 4.10: Sistem internih imulzov Q1 – Q4 in izvajanje instrukcij

Na naslednji sliki vidimo prepletanje instrukcijskega in izvajalnega cikla pri mikrokrmilniku PIC18F452. V enem instrukcijskem ciklu se ukaz dekodira, v drugem ciklu pa se ukaz izvrši. Zaradi prepletanja obeh ciklov pa se celoten ukaz dejansko izvede v enem ciklu Tcy, kar je prikazano na spodnji sliki 4.11.



Slika 4.11: Sistem internih imulzov Q1 – Q4 in izvajanje instrukcij

Izjema so le vejitvene instrukcije (npr. GOTO), kjer se zaradi programskega skoka vsebina programskega števca spremeni. V takšnem primeru sta za izvedbo instrukcije potrebna dva cikla Tcy.

Programski spomin se naslavlja v bajtih. Posamezne instrukcije so shranjene programskem spominu kot dva ali pa štirje bajti. Manj pomembni bajt instrukcije je vedno shranjen na sodi lokaciji v spominu. Za preprečevanje napačnega branja vsebine programskega spomina pri besednih instrukcijah je najmanj pomemben bit programskega števca vedno 0.

4.9 Nabor instrukcij mikrokrmilnika PIC18FXX20

Večina instrukcij pri mikrokrmilnikih PIC18FXX2 obsega 16 bitov, le tri instrukcije pa obsegajo po dve lokaciji v programskem spominu. Vsaka instrukcija je sestavljena iz

operacijske kode in iz enega ali dveh operandov. Celoten nabor instrukcij lahko razdelimo v štiri skupine:

- bajtne operacije (ang. Byte-oriented operations)
- bitne operacije (ang. Bit-oriented operations)
- operacije s konstantami (ang. Literal operations)
- krmilne (nadzorne) operacije (ang. Control operations)

Večina bajtnih operacij ima tri operande:

- registrske operacije, ki so označene z »f« (ang. file register)
- operacije z določeno ciljno lokacijo rezultata z oznako »d« (ang. destination of the result)
- operacije za dostop do spomina z oznako »a« (ang. the accessed memory)

Z »f« je označen register do katerega z instrukcijo dostopamo. Z »d« označujemo register, v katerega se bo vpisal rezultat operacije. Če je d=0, potem se rezultat vpiše v delovni register WREG, pri vrednosti d=1 pa se rezultat vpiše v register, ki je določen s tekočo instrukcijo.

Vse bitne operacije imajo tri vrste operandov:

- register, ki ga označimo z »f«,
- določen bit v registru, ki ga označimo z »b« in
- lokacijo v spominu, ki jo označimo z »a«.

Z oznako »b« je oštevilčen bit v določenem registru »f«, na katerega neka operacija vpliva. Namesto številke lahko uporabimo kot operand tudi ime bita.
Operacije s konstantami (ang. literal) pa uporabljajo naslednje tipe operandov:

- konstanto, ki jo želimo vnesti v register (oznaka »k«),
- FSR, v katerga želimo vpisato neko konstanto (oznaka »f«) in
- operacije brez operanda (oznaka »-»).

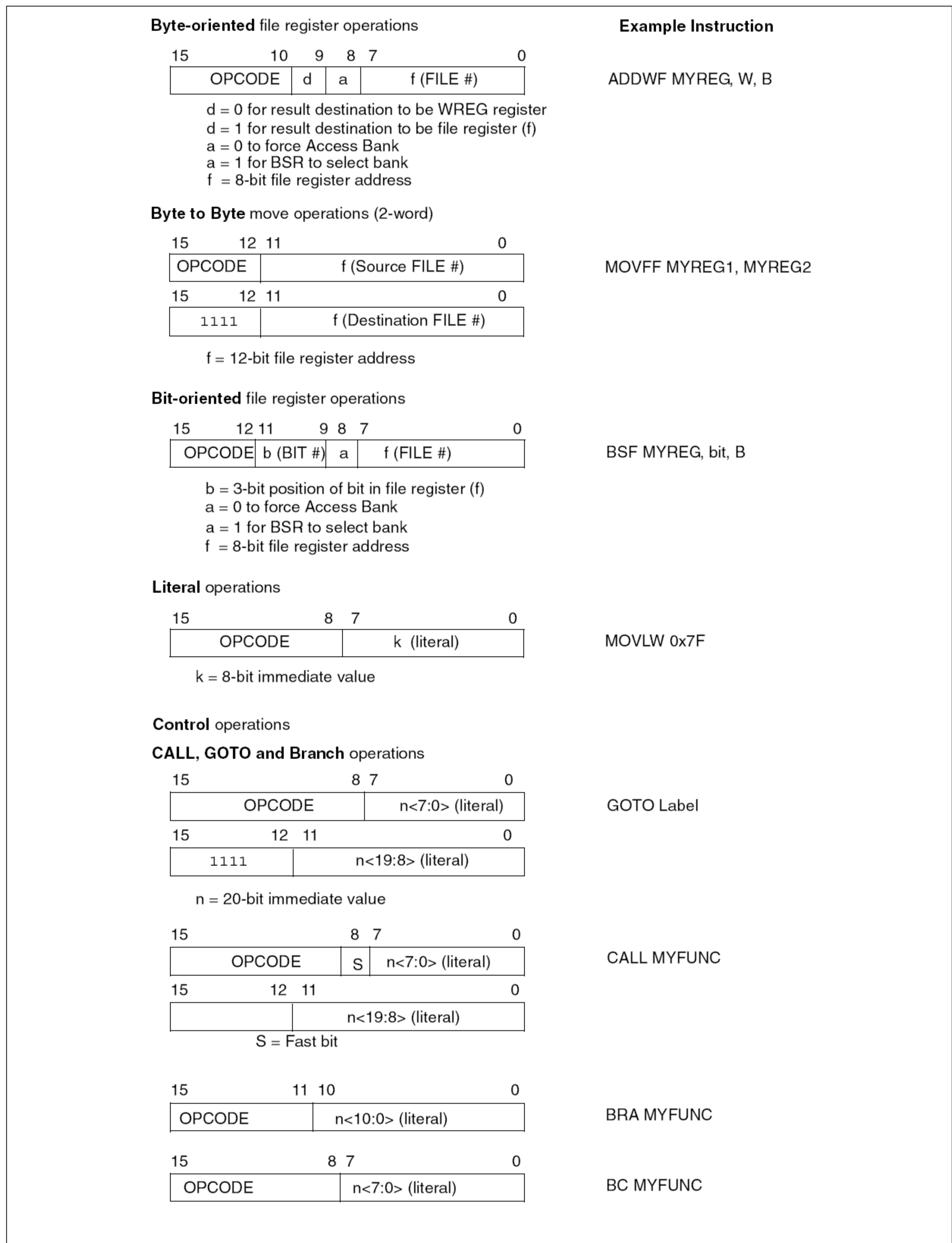
Pri krmilnih ali nadzornih operacijah uporabljamo naslednje operande:

- naslov lokacije v spominu (oznaka »n«),
- Call in Return instrukcija (oznaka »s«),
- način branja oz. vpisovanja tabel (oznaka »m«) in
- operacije brez operandov (oznaka »

Vse instrukcije se izvedejo v enem insrukcijskem ciklu, razen vejitvenih instrukcij, za katere sta potrebna dva instrukcijska cikla. Na sliki 4.12 so prikazane oznake, ki se uporabljajo v zbirnem jeziku pri mikrokrmilnikih PIC18FXX20

Field	Description
a	RAM access bit a = 0: RAM location in Access RAM (BSR register is ignored) a = 1: RAM bank is specified by BSR register
bbb	Bit address within an 8-bit file register (0 to 7)
BSR	Bank Select Register. Used to select the current RAM bank.
d	Destination select bit; d = 0: store result in WREG, d = 1: store result in file register f.
dest	Destination either the WREG register or the specified register file location
f	8-bit Register file address (0x00 to 0xFF)
fs	12-bit Register file address (0x000 to 0xFFF). This is the source address.
fd	12-bit Register file address (0x000 to 0xFFF). This is the destination address.
k	Literal field, constant data or label (may be either an 8-bit, 12-bit or a 20-bit value)
label	Label name
mm	The mode of the TBLPTR register for the Table Read and Table Write instructions. Only used with Table Read and Table Write instructions:
*	No Change to register (such as TBLPTR with Table reads and writes)
*+	Post-Increment register (such as TBLPTR with Table reads and writes)
*-	Post-Decrement register (such as TBLPTR with Table reads and writes)
+*	Pre-Increment register (such as TBLPTR with Table reads and writes)
n	The relative address (2's complement number) for relative branch instructions, or the direct address for Call/Branch and Return instructions
PRODH	Product of Multiply high byte
PRODL	Product of Multiply low byte
s	Fast Call/Return mode select bit. s = 0: do not update into/from shadow registers s = 1: certain registers loaded into/from shadow registers (Fast mode)
u	Unused or Unchanged
WREG	Working register (accumulator)
x	Don't care (0 or 1) The assembler will generate code with x = 0. It is the recommended form of use for compatibility with all Microchip software tools.
TBLPTR	21-bit Table Pointer (points to a Program Memory location)
TABLAT	8-bit Table Latch
TOS	Top-of-Stack
PC	Program Counter
PCL	Program Counter Low Byte
PCH	Program Counter High Byte
PCLATH	Program Counter High Byte Latch
PCLATU	Program Counter Upper Byte Latch
GIE	Global Interrupt Enable bit
WDT	Watchdog Timer
$\overline{TO}$	Time-out bit
$\overline{PD}$	Power-down bit
C, DC, Z, OV, N	ALU status bits Carry, Digit Carry, Zero, Overflow, Negative
[ ]	Optional
( )	Contents
→	Assigned to
< >	Register bit field
∈	In the set of
italics	User defined term (font is courier)

Slika 4.12: Pomembnejše oznake za programiranju PIC18FXX20 v zbirnem jeziku



Slika 4.13: Splošna oblika instrukcij pri mikrokrmilniku PIC18FXX20

Mnemonic, Operands		Description	Cycles	16-Bit Instruction Word				Status Affected	Notes
				MSb			LSb		
BYTE-ORIENTED FILE REGISTER OPERATIONS									
ADDWF	f, d, a	Add WREG and f	1	0010	01da0	ffff	ffff	C, DC, Z, OV, N	1, 2
ADDWFC	f, d, a	Add WREG and Carry bit to f	1	0010	0da	ffff	ffff	C, DC, Z, OV, N	1, 2
ANDWF	f, d, a	AND WREG with f	1	0001	01da	ffff	ffff	Z, N	1,2
CLRF	f, a	Clear f	1	0110	101a	ffff	ffff	Z	2
COMF	f, d, a	Complement f	1	0001	11da	ffff	ffff	Z, N	1, 2
CPFSEQ	f, a	Compare f with WREG, skip =	1 (2 or 3)	0110	001a	ffff	ffff	None	4
CPFSGT	f, a	Compare f with WREG, skip >	1 (2 or 3)	0110	010a	ffff	ffff	None	4
CPFSLT	f, a	Compare f with WREG, skip <	1 (2 or 3)	0110	000a	ffff	ffff	None	1, 2
DECF	f, d, a	Decrement f	1	0000	01da	ffff	ffff	C, DC, Z, OV, N	1, 2, 3, 4
DECFSZ	f, d, a	Decrement f, Skip if 0	1 (2 or 3)	0010	11da	ffff	ffff	None	1, 2, 3, 4
DCFSNZ	f, d, a	Decrement f, Skip if Not 0	1 (2 or 3)	0100	11da	ffff	ffff	None	1, 2
INCF	f, d, a	Increment f	1	0010	10da	ffff	ffff	C, DC, Z, OV, N	1, 2, 3, 4
INCFSZ	f, d, a	Increment f, Skip if 0	1 (2 or 3)	0011	11da	ffff	ffff	None	4
INFSNZ	f, d, a	Increment f, Skip if Not 0	1 (2 or 3)	0100	10da	ffff	ffff	None	1, 2
IORWF	f, d, a	Inclusive OR WREG with f	1	0001	00da	ffff	ffff	Z, N	1, 2
MOVF	f, d, a	Move f	1	0101	00da	ffff	ffff	Z, N	1
MOVFF	f_s, f_d	Move f_s (source) to 1st word f_d (destination) 2nd word	2	1100 1111	ffff ffff	ffff ffff	ffff ffff	None	
MOVWF	f, a	Move WREG to f	1	0110	111a	ffff	ffff	None	
MULWF	f, a	Multiply WREG with f	1	0000	001a	ffff	ffff	None	
NEGF	f, a	Negate f	1	0110	110a	ffff	ffff	C, DC, Z, OV, N	1, 2
RLCF	f, d, a	Rotate Left f through Carry	1	0011	01da	ffff	ffff	C, Z, N	
RLNCF	f, d, a	Rotate Left f (No Carry)	1	0100	01da	ffff	ffff	Z, N	1, 2
RRCF	f, d, a	Rotate Right f through Carry	1	0011	00da	ffff	ffff	C, Z, N	
RRNCF	f, d, a	Rotate Right f (No Carry)	1	0100	00da	ffff	ffff	Z, N	
SETF	f, a	Set f	1	0110	100a	ffff	ffff	None	
SUBFWB	f, d, a	Subtract f from WREG with borrow	1	0101	01da	ffff	ffff	C, DC, Z, OV, N	1, 2
SUBWF	f, d, a	Subtract WREG from f	1	0101	11da	ffff	ffff	C, DC, Z, OV, N	
SUBWFB	f, d, a	Subtract WREG from f with borrow	1	0101	10da	ffff	ffff	C, DC, Z, OV, N	1, 2
SWAPF	f, d, a	Swap nibbles in f	1	0011	10da	ffff	ffff	None	4
TSTFSZ	f, a	Test f, skip if 0	1 (2 or 3)	0110	011a	ffff	ffff	None	1, 2
XORWF	f, d, a	Exclusive OR WREG with f	1	0001	10da	ffff	ffff	Z, N	
BIT-ORIENTED FILE REGISTER OPERATIONS									
BCF	f, b, a	Bit Clear f	1	1001	bbba	ffff	ffff	None	1, 2
BSF	f, b, a	Bit Set f	1	1000	bbba	ffff	ffff	None	1, 2
BTFSC	f, b, a	Bit Test f, Skip if Clear	1 (2 or 3)	1011	bbba	ffff	ffff	None	3, 4
BTFSS	f, b, a	Bit Test f, Skip if Set	1 (2 or 3)	1010	bbba	ffff	ffff	None	3, 4
BTG	f, d, a	Bit Toggle f	1	0111	bbba	ffff	ffff	None	1, 2

Slika 4.14a: Nabor bajtnih instrukcij pri mikrokrmilniku PIC18FXX20

Mnemonic, Operands		Description	Cycles	16-Bit Instruction Word	Status Affected	Notes
				MSb LSb		
CONTROL OPERATIONS						
BC	n	Branch if Carry	1 (2)	1110 0010 nnnn nnnn	None	
BN	n	Branch if Negative	1 (2)	1110 0110 nnnn nnnn	None	
BNC	n	Branch if Not Carry	1 (2)	1110 0011 nnnn nnnn	None	
BNN	n	Branch if Not Negative	1 (2)	1110 0111 nnnn nnnn	None	
BNOV	n	Branch if Not Overflow	1 (2)	1110 0101 nnnn nnnn	None	
BNZ	n	Branch if Not Zero	2	1110 0001 nnnn nnnn	None	
BOV	n	Branch if Overflow	1 (2)	1110 0100 nnnn nnnn	None	
BRA	n	Branch Unconditionally	1 (2)	1101 0nnn nnnn nnnn	None	
BZ	n	Branch if Zero	1 (2)	1110 0000 nnnn nnnn	None	
CALL	n, s	Call subroutine1st word	2	1110 110s kkkk kkkk	None	
		2nd word		1111 kkkk kkkk kkkk		
CLRWDT	—	Clear Watchdog Timer	1	0000 0000 0000 0100	$\overline{TO}$, $\overline{PD}$	
DAW	—	Decimal Adjust WREG	1	0000 0000 0000 0111	C	
GOTO	n	Go to address1st word	2	1110 1111 kkkk kkkk	None	
		2nd word		1111 kkkk kkkk kkkk		
NOP	—	No Operation	1	0000 0000 0000 0000	None	
NOP	—	No Operation	1	1111 xxxx xxxx xxxx	None	4
POP	—	Pop top of return stack (TOS)	1	0000 0000 0000 0110	None	
PUSH	—	Push top of return stack (TOS)	1	0000 0000 0000 0101	None	
RCALL	n	Relative Call	2	1101 1nnn nnnn nnnn	None	
RESET		Software device RESET	1	0000 0000 1111 1111	All	
RETFIE	s	Return from interrupt enable	2	0000 0000 0001 000s	GIE/GIEH, PEIE/GIEL	
RETLW	k	Return with literal in WREG	2	0000 1100 kkkk kkkk	None	
RETURN	s	Return from Subroutine	2	0000 0000 0001 001s	None	
SLEEP	—	Go into Standby mode	1	0000 0000 0000 0011	$\overline{TO}$, $\overline{PD}$	

Slika 4.14b: Nabor krmilnih instrukcij pri mikrokrmilniku PIC18FXX20

Mnemonic, Operands		Description	Cycles	16-Bit Instruction Word	Status Affected	Notes
				MSb LSb		
LITERAL OPERATIONS						
ADDLW	k	Add literal and WREG	1	0000 1111 kkkk kkkk	C, DC, Z, OV, N	
ANDLW	k	AND literal with WREG	1	0000 1011 kkkk kkkk	Z, N	
IORLW	k	Inclusive OR literal with WREG	1	0000 1001 kkkk kkkk	Z, N	
LFSR	f, k	Move literal (12-bit) 2nd word	2	1110 1110 00ff kkkk	None	
		to FSRx 1st word		1111 0000 kkkk kkkk		
MOVLB	k	Move literal to BSR<3:0>	1	0000 0001 0000 kkkk	None	
MOVLW	k	Move literal to WREG	1	0000 1110 kkkk kkkk	None	
MULLW	k	Multiply literal with WREG	1	0000 1101 kkkk kkkk	None	
RETLW	k	Return with literal in WREG	2	0000 1100 kkkk kkkk	None	
SUBLW	k	Subtract WREG from literal	1	0000 1000 kkkk kkkk	C, DC, Z, OV, N	
XORLW	k	Exclusive OR literal with WREG	1	0000 1010 kkkk kkkk	Z, N	
DATA MEMORY ↔ PROGRAM MEMORY OPERATIONS						
TBLRD*		Table Read	2	0000 0000 0000 1000	None	
TBLRD*+		Table Read with post-increment		0000 0000 0000 1001	None	
TBLRD*-		Table Read with post-decrement		0000 0000 0000 1010	None	
TBLRD+*		Table Read with pre-increment		0000 0000 0000 1011	None	
TBLWT*		Table Write	2 (5)	0000 0000 0000 1100	None	
TBLWT*+		Table Write with post-increment		0000 0000 0000 1101	None	
TBLWT*-		Table Write with post-decrement		0000 0000 0000 1110	None	
TBLWT+*		Table Write with pre-increment		0000 0000 0000 1111	None	

Slika 4.14c: Nabor instrukcij s konstantami pri mikrokrmilniku PIC18FXX20

1.4.2 Opis instrukcij za mikrokrmilnik PIC18FXX20

ADDLW **ADD literal to W**

Syntax: [*label*] ADDLW k

Operands: $0 \leq k \leq 255$

Operation: (W) + k → W

Status Affected: N, OV, C, DC, Z

Encoding:

0000	1111	kkkk	kkkk

Description: The contents of W are added to the 8-bit literal 'k' and the result is placed in W.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'	Process Data	Write to W

Example: `ADDLW 0x15`

Before Instruction
W = 0x10
After Instruction
W = 0x25

ADDWF **ADD W to f**

Syntax: [*label*] ADDWF f [,d [,a]

Operands: $0 \leq f \leq 255$
$d \in [0,1]$
$a \in [0,1]$

Operation: (W) + (f) → dest

Status Affected: N, OV, C, DC, Z

Encoding:

0010	01da	ffff	ffff

Description: Add W to register 'f'. If 'd' is 0, the result is stored in W. If 'd' is 1, the result is stored back in register 'f' (default). If 'a' is 0, the Access Bank will be selected. If 'a' is 1, the BSR is used.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Example: `ADDWF REG, 0, 0`

Before Instruction
W = 0x17
REG = 0xC2
After Instruction
W = 0xD9
REG = 0xC2

ADDWFC — ADD W and Carry bit to f

Syntax:	[*label*] ADDWFC f [,d [,a]
Operands:	$0 \le f \le 255$ $d \in [0,1]$ $a \in [0,1]$
Operation:	(W) + (f) + (C) → dest
Status Affected:	N,OV, C, DC, Z
Encoding:	`0010` `00da` `ffff` `ffff`
Description:	Add W, the Carry Flag and data memory location 'f'. If 'd' is 0, the result is placed in W. If 'd' is 1, the result is placed in data memory location 'f'. If 'a' is 0, the Access Bank will be selected. If 'a' is 1, the BSR will not be overridden.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Example: `ADDWFC REG, 0, 1`

Before Instruction

Carry bit = 1
REG = 0x02
W = 0x4D

After Instruction

Carry bit = 0
REG = 0x02
W = 0x50

ANDLW — AND literal with W

Syntax:	[*label*] ANDLW k
Operands:	$0 \le k \le 255$
Operation:	(W) .AND. k → W
Status Affected:	N,Z
Encoding:	`0000` `1011` `kkkk` `kkkk`
Description:	The contents of W are ANDed with the 8-bit literal 'k'. The result is placed in W.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'	Process Data	Write to W

Example: `ANDLW 0x5F`

Before Instruction

W = 0xA3

After Instruction

W = 0x03

ANDWF — AND W with f

Syntax: [*label*] ANDWF f [,d [,a]

Operands: $0 \le f \le 255$
$d \in [0,1]$
$a \in [0,1]$

Operation: (W) .AND. (f) → dest

Status Affected: N,Z

Encoding:

0001	01da	ffff	ffff

Description: The contents of W are AND'ed with register 'f'. If 'd' is 0, the result is stored in W. If 'd' is 1, the result is stored back in register 'f' (default). If 'a' is 0, the Access Bank will be selected. If 'a' is 1, the BSR will not be overridden (default).

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Example: `ANDWF REG, 0, 0`

Before Instruction

W = 0x17
REG = 0xC2

After Instruction

W = 0x02
REG = 0xC2

BC — Branch if Carry

Syntax: [*label*] BC n

Operands: $-128 \le n \le 127$

Operation: if carry bit is '1'
(PC) + 2 + 2n → PC

Status Affected: None

Encoding:

1110	0010	nnnn	nnnn

Description: If the Carry bit is '1', then the program will branch.
The 2's complement number '2n' is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be PC+2+2n. This instruction is then a two-cycle instruction.

Words: 1

Cycles: 1(2)

Q Cycle Activity:
If Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	Write to PC
No operation	No operation	No operation	No operation

If No Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	No operation

Example: `HERE BC 5`

Before Instruction

PC = address (HERE)

After Instruction

If Carry = 1;
PC = address (HERE+12)
If Carry = 0;
PC = address (HERE+2)

BCF — Bit Clear f

Syntax: [*label*] BCF f,b[,a]

Operands: $0 \le f \le 255$
$0 \le b \le 7$
$a \in [0,1]$

Operation: $0 \rightarrow$ f<b>

Status Affected: None

Encoding:

1001	bbba	ffff	ffff

Description: Bit 'b' in register 'f' is cleared. If ‘a’ is 0, the Access Bank will be selected, overriding the BSR value. If ‘a’ = 1, then the bank will be selected as per the BSR value (default).

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write register 'f'

Example: `BCF FLAG_REG, 7, 0`

Before Instruction
FLAG_REG = 0xC7

After Instruction
FLAG_REG = 0x47

BN — Branch if Negative

Syntax: [*label*] BN n

Operands: $-128 \le n \le 127$

Operation: if negative bit is ’1’
$(PC) + 2 + 2n \rightarrow PC$

Status Affected: None

Encoding:

1110	0110	nnnn	nnnn

Description: If the Negative bit is ’1’, then the program will branch.
The 2’s complement number ’2n’ is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be PC+2+2n. This instruction is then a two-cycle instruction.

Words: 1

Cycles: 1(2)

Q Cycle Activity:

If Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	Write to PC
No operation	No operation	No operation	No operation

If No Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	No operation

Example: `HERE BN Jump`

Before Instruction
PC = address `(HERE)`

After Instruction
If Negative = 1;
PC = address `(Jump)`
If Negative = 0;
PC = address `(HERE+2)`

BNC Branch if Not Carry

Syntax: [*label*] BNC n

Operands: $-128 \leq n \leq 127$

Operation: if carry bit is '0'
(PC) + 2 + 2n → PC

Status Affected: None

Encoding:

1110	0011	nnnn	nnnn

Description: If the Carry bit is '0', then the program will branch.
The 2's complement number '2n' is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be PC+2+2n. This instruction is then a two-cycle instruction.

Words: 1

Cycles: 1(2)

Q Cycle Activity:
If Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	Write to PC
No operation	No operation	No operation	No operation

If No Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	No operation

Example: `HERE BNC Jump`

Before Instruction
PC = address (HERE)

After Instruction
If Carry = 0;
PC = address (Jump)
If Carry = 1;
PC = address (HERE+2)

BNN Branch if Not Negative

Syntax: [*label*] BNN n

Operands: $-128 \leq n \leq 127$

Operation: if negative bit is '0'
(PC) + 2 + 2n → PC

Status Affected: None

Encoding:

1110	0111	nnnn	nnnn

Description: If the Negative bit is '0', then the program will branch.
The 2's complement number '2n' is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be PC+2+2n. This instruction is then a two-cycle instruction.

Words: 1

Cycles: 1(2)

Q Cycle Activity:
If Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	Write to PC
No operation	No operation	No operation	No operation

If No Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	No operation

Example: `HERE BNN Jump`

Before Instruction
PC = address (HERE)

After Instruction
If Negative = 0;
PC = address (Jump)
If Negative = 1;
PC = address (HERE+2)

BNOV — Branch if Not Overflow

Syntax: [*label*] BNOV n

Operands: $-128 \le n \le 127$

Operation: if overflow bit is '0'
$(PC) + 2 + 2n \rightarrow PC$

Status Affected: None

Encoding:

1110	0101	nnnn	nnnn

Description: If the Overflow bit is '0', then the program will branch.
The 2's complement number '2n' is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be PC+2+2n. This instruction is then a two-cycle instruction.

Words: 1

Cycles: 1(2)

Q Cycle Activity:
If Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	Write to PC
No operation	No operation	No operation	No operation

If No Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	No operation

Example: `HERE BNOV Jump`

Before Instruction
PC = address (HERE)

After Instruction
If Overflow = 0;
PC = address (Jump)
If Overflow = 1;
PC = address (HERE+2)

BNZ — Branch if Not Zero

Syntax: [*label*] BNZ n

Operands: $-128 \le n \le 127$

Operation: if zero bit is '0'
$(PC) + 2 + 2n \rightarrow PC$

Status Affected: None

Encoding:

1110	0001	nnnn	nnnn

Description: If the Zero bit is '0', then the program will branch.
The 2's complement number '2n' is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be PC+2+2n. This instruction is then a two-cycle instruction.

Words: 1

Cycles: 1(2)

Q Cycle Activity:
If Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	Write to PC
No operation	No operation	No operation	No operation

If No Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	No operation

Example: `HERE BNZ Jump`

Before Instruction
PC = address (HERE)

After Instruction
If Zero = 0;
PC = address (Jump)
If Zero = 1;
PC = address (HERE+2)

BTFSC — Bit Test File, Skip if Clear

Syntax:	[*label*] BTFSC f,b[,a]
Operands:	0 ≤ f ≤ 255 0 ≤ b ≤ 7 a ∈ [0,1]
Operation:	skip if (f<b>) = 0
Status Affected:	None
Encoding:	`1011` `bbba` `ffff` `ffff`
Description:	If bit 'b' in register 'f' is 0, then the next instruction is skipped. If bit 'b' is 0, then the next instruction fetched during the current instruction execution is discarded, and a `NOP` is executed instead, making this a two-cycle instruction. If 'a' is 0, the Access Bank will be selected, over-riding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).
Words:	1
Cycles:	1(2) **Note:** 3 cycles if skip and followed by a 2-word instruction.

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	No operation

If skip:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation

If skip and followed by 2-word instruction:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation
No operation	No operation	No operation	No operation

Example:

```
HERE     BTFSC    FLAG, 1, 0
FALSE    :
TRUE     :
```

Before Instruction
PC = address (HERE)

After Instruction
If FLAG<1> = 0;
PC = address (TRUE)
If FLAG<1> = 1;
PC = address (FALSE)

BTFSS — Bit Test File, Skip if Set

Syntax:	[*label*] BTFSS f,b[,a]
Operands:	0 ≤ f ≤ 255 0 ≤ b ≤ 7 a ∈ [0,1]
Operation:	skip if (f<b>) = 1
Status Affected:	None
Encoding:	`1010` `bbba` `ffff` `ffff`
Description:	If bit 'b' in register 'f' is 1, then the next instruction is skipped. If bit 'b' is 1, then the next instruction fetched during the current instruction execution, is discarded and a `NOP` is executed instead, making this a two-cycle instruction. If 'a' is 0, the Access Bank will be selected, over-riding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).
Words:	1
Cycles:	1(2) **Note:** 3 cycles if skip and followed by a 2-word instruction.

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	No operation

If skip:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation

If skip and followed by 2-word instruction:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation
No operation	No operation	No operation	No operation

Example:

```
HERE     BTFSS    FLAG, 1, 0
FALSE    :
TRUE     :
```

Before Instruction
PC = address (HERE)

After Instruction
If FLAG<1> = 0;
PC = address (FALSE)
If FLAG<1> = 1;
PC = address (TRUE)

BTG **Bit Toggle f**

Syntax: [*label*] BTG f,b[,a]

Operands: $0 \le f \le 255$
$0 \le b \le 7$
$a \in [0,1]$

Operation: $(\overline{f<b>}) \rightarrow f<b>$

Status Affected: None

Encoding:

0111	bbba	ffff	ffff

Description: Bit 'b' in data memory location 'f' is inverted. If ‘a’ is 0, the Access Bank will be selected, overriding the BSR value. If ‘a’ = 1, then the bank will be selected as per the BSR value (default).

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write register 'f'

Example:
```
BTG     PORTC,  4, 0
```

Before Instruction:
PORTC = 0111 0101 [0x75]

After Instruction:
PORTC = 0110 0101 [0x65]

BOV **Branch if Overflow**

Syntax: [*label*] BOV n

Operands: $-128 \le n \le 127$

Operation: if overflow bit is ’1’
$(PC) + 2 + 2n \rightarrow PC$

Status Affected: None

Encoding:

1110	0100	nnnn	nnnn

Description: If the Overflow bit is ’1’, then the program will branch.
The 2’s complement number ’2n’ is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be PC+2+2n. This instruction is then a two-cycle instruction.

Words: 1

Cycles: 1(2)

Q Cycle Activity:

If Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	Write to PC
No operation	No operation	No operation	No operation

If No Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	No operation

Example:
```
HERE      BOV  Jump
```

Before Instruction
PC = address (HERE)

After Instruction
If Overflow = 1;
PC = address (Jump)
If Overflow = 0;
PC = address (HERE+2)

BZ — Branch if Zero

Syntax: [*label*] BZ n

Operands: $-128 \leq n \leq 127$

Operation: if Zero bit is '1'
(PC) + 2 + 2n → PC

Status Affected: None

Encoding:

1110	0000	nnnn	nnnn

Description: If the Zero bit is '1', then the program will branch.
The 2's complement number '2n' is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be PC+2+2n. This instruction is then a two-cycle instruction.

Words: 1

Cycles: 1(2)

Q Cycle Activity:
If Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	Write to PC
No operation	No operation	No operation	No operation

If No Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	No operation

Example: `HERE BZ Jump`

Before Instruction
PC = address (HERE)

After Instruction
If Zero = 1;
PC = address (Jump)
If Zero = 0;
PC = address (HERE+2)

CALL — Subroutine Call

Syntax: [*label*] CALL k [,s]

Operands: $0 \leq k \leq 1048575$
$s \in [0,1]$

Operation: (PC) + 4 → TOS,
k → PC<20:1>,
if s = 1
(W) → WS,
(STATUS) → STATUSS,
(BSR) → BSRS

Status Affected: None

Encoding:

1st word (k<7:0>)	1110	110s	$k_7$kkk	kkkk_0
2nd word(k<19:8>)	1111	k_{19}kkk	kkkk	kkkk_8

Description: Subroutine call of entire 2 Mbyte memory range. First, return address (PC+ 4) is pushed onto the return stack. If 's' = 1, the W, STATUS and BSR registers are also pushed into their respective shadow registers, WS, STATUSS and BSRS. If 's' = 0, no update occurs (default). Then, the 20-bit value 'k' is loaded into PC<20:1>. `CALL` is a two-cycle instruction.

Words: 2

Cycles: 2

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'<7:0>,	Push PC to stack	Read literal 'k'<19:8>, Write to PC
No operation	No operation	No operation	No operation

Example: `HERE CALL THERE,1`

Before Instruction
PC = address (HERE)

After Instruction
PC = address (THERE)
TOS = address (HERE + 4)
WS = W
BSRS = BSR
STATUSS= STATUS

CLRF **Clear f**

Syntax: [*label*] CLRF f [,a]

Operands: $0 \le f \le 255$
$a \in [0,1]$

Operation: 000h → f
1 → Z

Status Affected: Z

Encoding:

0110	101a	ffff	ffff

Description: Clears the contents of the specified register. If ‘a’ is 0, the Access Bank will be selected, overriding the BSR value. If ‘a’ = 1, then the bank will be selected as per the BSR value (default).

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write register 'f'

Example: `CLRF FLAG_REG,1`

Before Instruction
FLAG_REG = 0x5A

After Instruction
FLAG_REG = 0x00

CLRWDT **Clear Watchdog Timer**

Syntax: [*label*] CLRWDT

Operands: None

Operation: 000h → WDT,
000h → WDT postscaler,
1 → $\overline{TO}$,
1 → $\overline{PD}$

Status Affected: $\overline{TO}$, $\overline{PD}$

Encoding:

0000	0000	0000	0100

Description: `CLRWDT` instruction resets the Watchdog Timer. It also resets the postscaler of the WDT. Status bits $\overline{TO}$ and $\overline{PD}$ are set.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	No operation	Process Data	No operation

Example: `CLRWDT`

Before Instruction
WDT Counter = ?

After Instruction
WDT Counter = 0x00
WDT Postscaler = 0
$\overline{TO}$ = 1
$\overline{PD}$ = 1

COMF — Complement f

Syntax: [*label*] COMF f [,d [,a]

Operands: $0 \le f \le 255$
d ∈ [0,1]
a ∈ [0,1]

Operation: $(\overline{f}) \rightarrow$ dest

Status Affected: N, Z

Encoding:

0001	11da	ffff	ffff

Description: The contents of register 'f' are complemented. If 'd' is 0, the result is stored in W. If 'd' is 1, the result is stored back in register 'f' (default). If 'a' is 0, the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Example: `COMF REG, 0, 0`

Before Instruction
REG = 0x13
After Instruction
REG = 0x13
W = 0xEC

CPFSEQ — Compare f with W, skip if f = W

Syntax: [*label*] CPFSEQ f [,a]

Operands: $0 \le f \le 255$
a ∈ [0,1]

Operation: (f) – (W),
skip if (f) = (W)
(unsigned comparison)

Status Affected: None

Encoding:

0110	001a	ffff	ffff

Description: Compares the contents of data memory location 'f' to the contents of W by performing an unsigned subtraction.
If 'f' = W, then the fetched instruction is discarded and a `NOP` is executed instead, making this a two-cycle instruction. If 'a' is 0, the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).

Words: 1

Cycles: 1(2)
Note: 3 cycles if skip and followed by a 2-word instruction.

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	No operation

If skip:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation

If skip and followed by 2-word instruction:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation
No operation	No operation	No operation	No operation

Example:

```
HERE     CPFSEQ REG, 0
NEQUAL   :
EQUAL    :
```

Before Instruction
PC Address = `HERE`
W = ?
REG = ?
After Instruction
If REG = W;
PC = Address (`EQUAL`)
If REG ≠ W;
PC = Address (`NEQUAL`)

CPFSGT — Compare f with W, skip if f > W

Syntax: [*label*] CPFSGT f [,a]

Operands: $0 \le f \le 255$
$a \in [0,1]$

Operation: (f) – (W),
skip if (f) > (W)
(unsigned comparison)

Status Affected: None

Encoding:

0110	010a	ffff	ffff

Description: Compares the contents of data memory location 'f' to the contents of the W by performing an unsigned subtraction.
If the contents of 'f' are greater than the contents of WREG, then the fetched instruction is discarded and a `NOP` is executed instead, making this a two-cycle instruction. If 'a' is 0, the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).

Words: 1

Cycles: 1(2)
Note: 3 cycles if skip and followed by a 2-word instruction.

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	No operation

If skip:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation

If skip and followed by 2-word instruction:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation
No operation	No operation	No operation	No operation

Example:

```
HERE       CPFSGT REG, 0
NGREATER   :
GREATER    :
```

Before Instruction
PC = Address (HERE)
W = ?

After Instruction
If REG > W;
PC = Address (GREATER)
If REG ≤ W;
PC = Address (NGREATER)

CPFSLT — Compare f with W, skip if f < W

Syntax: [*label*] CPFSLT f [,a]

Operands: $0 \le f \le 255$
$a \in [0,1]$

Operation: (f) – (W),
skip if (f) < (W)
(unsigned comparison)

Status Affected: None

Encoding:

0110	000a	ffff	ffff

Description: Compares the contents of data memory location 'f' to the contents of W by performing an unsigned subtraction.
If the contents of 'f' are less than the contents of W, then the fetched instruction is discarded and a `NOP` is executed instead, making this a two-cycle instruction. If 'a' is 0, the Access Bank will be selected. If 'a' is 1, the BSR will not be overridden (default).

Words: 1

Cycles: 1(2)
Note: 3 cycles if skip and followed by a 2-word instruction.

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	No operation

If skip:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation

If skip and followed by 2-word instruction:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation
No operation	No operation	No operation	No operation

Example:

```
HERE    CPFSLT REG, 1
NLESS   :
LESS    :
```

Before Instruction
PC = Address (HERE)
W = ?

After Instruction
If REG < W;
PC = Address (LESS)
If REG ≥ W;
PC = Address (NLESS)

DAW — Decimal Adjust W Register

Syntax: [*label*] DAW

Operands: None

Operation: If [W<3:0> >9] or [DC = 1] then
(W<3:0>) + 6 → W<3:0>;
else
(W<3:0>) → W<3:0>;

If [W<7:4> >9] or [C = 1] then
(W<7:4>) + 6 → W<7:4>;
else
(W<7:4>) → W<7:4>;

Status Affected: C

Encoding:

0000	0000	0000	0111

Description: `DAW` adjusts the eight-bit value in W, resulting from the earlier addition of two variables (each in packed BCD format) and produces a correct packed BCD result.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register W	Process Data	Write W

Example1: `DAW`

Before Instruction

W = 0xA5
C = 0
DC = 0

After Instruction

W = 0x05
C = 1
DC = 0

Example 2:

Before Instruction

W = 0xCE
C = 0
DC = 0

After Instruction

W = 0x34
C = 1
DC = 0

DECF — Decrement f

Syntax: [*label*] DECF f [,d [,a]

Operands: $0 \le f \le 255$
$d \in [0,1]$
$a \in [0,1]$

Operation: (f) – 1 → dest

Status Affected: C, DC, N, OV, Z

Encoding:

0000	01da	ffff	ffff

Description: Decrement register 'f'. If 'd' is 0, the result is stored in W. If 'd' is 1, the result is stored back in register 'f' (default). If 'a' is 0, the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Example: `DECF CNT, 1, 0`

Before Instruction

CNT = 0x01
Z = 0

After Instruction

CNT = 0x00
Z = 1

DECFSZ — Decrement f, skip if 0

Syntax: [*label*] DECFSZ f [,d [,a]]

Operands: $0 \le f \le 255$
$d \in [0,1]$
$a \in [0,1]$

Operation: (f) – 1 → dest,
skip if result = 0

Status Affected: None

Encoding:

0010	11da	ffff	ffff

Description: The contents of register 'f' are decremented. If 'd' is 0, the result is placed in W. If 'd' is 1, the result is placed back in register 'f' (default). If the result is 0, the next instruction, which is already fetched, is discarded, and a `NOP` is executed instead, making it a two-cycle instruction. If 'a' is 0, the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).

Words: 1

Cycles: 1(2)
Note: 3 cycles if skip and followed by a 2-word instruction.

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

If skip:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation

If skip and followed by 2-word instruction:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation
No operation	No operation	No operation	No operation

Example:

```
HERE       DECFSZ   CNT, 1, 1
           GOTO     LOOP
CONTINUE
```

Before Instruction
PC = Address (HERE)

After Instruction
CNT = CNT - 1
If CNT = 0;
PC = Address (CONTINUE)
If CNT ≠ 0;
PC = Address (HERE+2)

DCFSNZ — Decrement f, skip if not 0

Syntax: [*label*] DCFSNZ f [,d [,a]

Operands: $0 \le f \le 255$
$d \in [0,1]$
$a \in [0,1]$

Operation: (f) – 1 → dest,
skip if result ≠ 0

Status Affected: None

Encoding:

0100	11da	ffff	ffff

Description: The contents of register 'f' are decremented. If 'd' is 0, the result is placed in W. If 'd' is 1, the result is placed back in register 'f' (default). If the result is not 0, the next instruction, which is already fetched, is discarded, and a `NOP` is executed instead, making it a two-cycle instruction. If 'a' is 0, the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).

Words: 1

Cycles: 1(2)
Note: 3 cycles if skip and followed by a 2-word instruction.

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

If skip:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation

If skip and followed by 2-word instruction:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation
No operation	No operation	No operation	No operation

Example:

```
HERE     DCFSNZ  TEMP, 1, 0
ZERO     :
NZERO    :
```

Before Instruction
TEMP = ?

After Instruction
TEMP = TEMP - 1,
If TEMP = 0;
PC = Address (ZERO)
If TEMP ≠ 0;
PC = Address (NZERO)

GOTO **Unconditional Branch**

Syntax:	[*label*] GOTO k
Operands:	0 ≤ k ≤ 1048575
Operation:	k → PC<20:1>
Status Affected:	None

Encoding:

1st word (k<7:0>)	1110	1111	$k_7$kkk	kkkk$_0$
2nd word(k<19:8>)	1111	k_{19}kkk	kkkk	kkkk$_8$

Description: GOTO allows an unconditional branch anywhere within entire 2 Mbyte memory range. The 20-bit value 'k' is loaded into PC<20:1>. GOTO is always a two-cycle instruction.

Words: 2

Cycles: 2

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'<7:0>,	No operation	Read literal 'k'<19:8>, Write to PC
No operation	No operation	No operation	No operation

Example: `GOTO THERE`

After Instruction
PC = Address (THERE)

INCF **Increment f**

Syntax:	[*label*] INCF f [,d [,a]
Operands:	0 ≤ f ≤ 255 d ∈ [0,1] a ∈ [0,1]
Operation:	(f) + 1 → dest
Status Affected:	C, DC, N, OV, Z

Encoding:

0010	10da	ffff	ffff

Description: The contents of register 'f' are incremented. If 'd' is 0, the result is placed in W. If 'd' is 1, the result is placed back in register 'f' (default). If 'a' is 0, the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Example: `INCF CNT, 1, 0`

Before Instruction
CNT = 0xFF
Z = 0
C = ?
DC = ?

After Instruction
CNT = 0x00
Z = 1
C = 1
DC = 1

INCFSZ — Increment f, skip if 0

Syntax: [*label*] INCFSZ f [,d [,a]

Operands: 0 ≤ f ≤ 255
d ∈ [0,1]
a ∈ [0,1]

Operation: (f) + 1 → dest,
skip if result = 0

Status Affected: None

Encoding:

0011	11da	ffff	ffff

Description: The contents of register 'f' are incremented. If 'd' is 0, the result is placed in W. If 'd' is 1, the result is placed back in register 'f'. (default) If the result is 0, the next instruction, which is already fetched, is discarded, and a `NOP` is executed instead, making it a two-cycle instruction. If 'a' is 0, the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).

Words: 1

Cycles: 1(2)
Note: 3 cycles if skip and followed by a 2-word instruction.

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

If skip:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation

If skip and followed by 2-word instruction:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation
No operation	No operation	No operation	No operation

Example:

```
HERE    INCFSZ  CNT, 1, 0
NZERO   :
ZERO    :
```

Before Instruction
PC = Address (HERE)

After Instruction
CNT = CNT + 1
If CNT = 0;
PC = Address (ZERO)
If CNT ≠ 0;
PC = Address (NZERO)

INFSNZ — Increment f, skip if not 0

Syntax: [*label*] INFSNZ f [,d [,a]

Operands: 0 ≤ f ≤ 255
d ∈ [0,1]
a ∈ [0,1]

Operation: (f) + 1 → dest,
skip if result ≠ 0

Status Affected: None

Encoding:

0100	10da	ffff	ffff

Description: The contents of register 'f' are incremented. If 'd' is 0, the result is placed in W. If 'd' is 1, the result is placed back in register 'f' (default). If the result is not 0, the next instruction, which is already fetched, is discarded, and a `NOP` is executed instead, making it a two-cycle instruction. If 'a' is 0, the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).

Words: 1

Cycles: 1(2)
Note: 3 cycles if skip and followed by a 2-word instruction.

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

If skip:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation

If skip and followed by 2-word instruction:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation
No operation	No operation	No operation	No operation

Example:

```
HERE    INFSNZ  REG, 1, 0
ZERO
NZERO
```

Before Instruction
PC = Address (HERE)

After Instruction
REG = REG + 1
If REG ≠ 0;
PC = Address (NZERO)
If REG = 0;
PC = Address (ZERO)

IORLW — Inclusive OR literal with W

Syntax: [*label*] IORLW k

Operands: $0 \le k \le 255$

Operation: (W) .OR. k → W

Status Affected: N, Z

Encoding:

0000	1001	kkkk	kkkk

Description: The contents of W are OR'ed with the eight-bit literal 'k'. The result is placed in W.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'	Process Data	Write to W

Example: `IORLW 0x35`

Before Instruction
W = 0x9A

After Instruction
W = 0xBF

IORWF — Inclusive OR W with f

Syntax: [*label*] IORWF f [,d [,a]

Operands: $0 \le f \le 255$
$d \in [0,1]$
$a \in [0,1]$

Operation: (W) .OR. (f) → dest

Status Affected: N, Z

Encoding:

0001	00da	ffff	ffff

Description: Inclusive OR W with register 'f'. If 'd' is 0, the result is placed in W. If 'd' is 1, the result is placed back in register 'f' (default). If 'a' is 0, the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Example: `IORWF RESULT, 0, 1`

Before Instruction
RESULT = 0x13
W = 0x91

After Instruction
RESULT = 0x13
W = 0x93

LFSR — Load FSR

Syntax: [*label*] LFSR f,k

Operands: $0 \le f \le 2$
$0 \le k \le 4095$

Operation: $k \rightarrow$ FSRf

Status Affected: None

Encoding:

1110	1110	00ff	k_{11}kkk
1111	0000	$k_7$kkk	kkkk

Description: The 12-bit literal 'k' is loaded into the file select register pointed to by 'f'.

Words: 2

Cycles: 2

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k' MSB	Process Data	Write literal 'k' MSB to FSRfH
Decode	Read literal 'k' LSB	Process Data	Write literal 'k' to FSRfL

Example: `LFSR 2, 0x3AB`

After Instruction
FSR2H = 0x03
FSR2L = 0xAB

MOVF — Move f

Syntax: [*label*] MOVF f [,d [,a]

Operands: $0 \le f \le 255$
$d \in [0,1]$
$a \in [0,1]$

Operation: $f \rightarrow$ dest

Status Affected: N, Z

Encoding:

0101	00da	ffff	ffff

Description: The contents of register 'f' are moved to a destination dependent upon the status of 'd'. If 'd' is 0, the result is placed in W. If 'd' is 1, the result is placed back in register 'f' (default). Location 'f' can be anywhere in the 256 byte bank. If 'a' is 0, the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write W

Example: `MOVF REG, 0, 0`

Before Instruction
REG = 0x22
W = 0xFF

After Instruction
REG = 0x22
W = 0x22

MOVFF **Move f to f**

Syntax: [*label*] MOVFF f_s,f_d

Operands: $0 \le f_s \le 4095$
$0 \le f_d \le 4095$

Operation: $(f_s) \rightarrow f_d$

Status Affected: None

Encoding:

1st word (source)	1100	ffff	ffff	ffff$_s$
2nd word (destin.)	1111	ffff	ffff	ffff$_d$

Description: The contents of source register 'f_s' are moved to destination register 'f_d'. Location of source 'f_s' can be anywhere in the 4096 byte data space (000h to FFFh), and location of destination 'f_d' can also be anywhere from 000h to FFFh.
Either source or destination can be W (a useful special situation).
`MOVFF` is particularly useful for transferring a data memory location to a peripheral register (such as the transmit buffer or an I/O port).

The `MOVFF` instruction cannot use the PCL, TOSU, TOSH or TOSL as the destination register.

Note: The `MOVFF` instruction should not be used to modify interrupt settings while any interrupt is enabled. See Section 8.0 for more information.

Words: 2

Cycles: 2 (3)

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f' (src)	Process Data	No operation
Decode	No operation No dummy read	No operation	Write register 'f' (dest)

Example: `MOVFF REG1, REG2`

Before Instruction
REG1 = 0x33
REG2 = 0x11

After Instruction
REG1 = 0x33,
REG2 = 0x33

MOVLB **Move literal to low nibble in BS**

Syntax: [*label*] MOVLB k

Operands: $0 \le k \le 255$

Operation: $k \rightarrow$ BSR

Status Affected: None

Encoding:

0000	0001	kkkk	kkk

Description: The 8-bit literal 'k' is loaded into the Bank Select Register (BSR)

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'	Process Data	Write literal 'k' t BSR

Example: `MOVLB 5`

Before Instruction
BSR register = 0x02

After Instruction
BSR register = 0x05

MOVLW — Move literal to W

Syntax: [*label*] MOVLW k

Operands: $0 \leq k \leq 255$

Operation: $k \rightarrow W$

Status Affected: None

Encoding:

0000	1110	kkkk	kkkk

Description: The eight-bit literal 'k' is loaded into W.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'	Process Data	Write to W

Example: `MOVLW 0x5A`

After Instruction

W = 0x5A

MOVWF — Move W to f

Syntax: [*label*] MOVWF f [,a]

Operands: $0 \leq f \leq 255$
$a \in [0,1]$

Operation: $(W) \rightarrow f$

Status Affected: None

Encoding:

0110	111a	ffff	ffff

Description: Move data from W to register 'f'. Location 'f' can be anywhere in the 256 byte bank. If ‘a’ is 0, the Access Bank will be selected, over-riding the BSR value. If ‘a’ = 1, then the bank will be selected as per the BSR value (default).

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write register 'f'

Example: `MOVWF REG, 0`

Before Instruction

W = 0x4F
REG = 0xFF

After Instruction

W = 0x4F
REG = 0x4F

MULLW — Multiply Literal with W

Syntax:	[*label*] MULLW k
Operands:	$0 \le k \le 255$
Operation:	(W) x k → PRODH:PRODL
Status Affected:	None
Encoding:	`0000` `1101` `kkkk` `kkkk`
Description:	An unsigned multiplication is carried out between the contents of W and the 8-bit literal 'k'. The 16-bit result is placed in PRODH:PRODL register pair. PRODH contains the high byte. W is unchanged. None of the status flags are affected. Note that neither overflow nor carry is possible in this operation. A zero result is possible but not detected.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'	Process Data	Write registers PRODH: PRODL

Example: `MULLW 0xC4`

Before Instruction

W = 0xE2
PRODH = ?
PRODL = ?

After Instruction

W = 0xE2
PRODH = 0xAD
PRODL = 0x08

MULWF — Multiply W with f

Syntax:	[*label*] MULWF f [,a]
Operands:	$0 \le f \le 255$ $a \in [0,1]$
Operation:	(W) x (f) → PRODH:PRODL
Status Affected:	None
Encoding:	`0000` `001a` `ffff` `ffff`
Description:	An unsigned multiplication is carried out between the contents of W and the register file location 'f'. The 16-bit result is stored in the PRODH:PRODL register pair. PRODH contains the high byte. Both W and 'f' are unchanged. None of the status flags are affected. Note that neither overflow nor carry is possible in this operation. A zero result is possible but not detected. If 'a' is 0, the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write registers PRODH: PRODL

Example: `MULWF REG, 1`

Before Instruction

W = 0xC4
REG = 0xB5
PRODH = ?
PRODL = ?

After Instruction

W = 0xC4
REG = 0xB5
PRODH = 0x8A
PRODL = 0x94

NEGF **Negate f**

Syntax:	[*label*] NEGF f [,a]
Operands:	$0 \le f \le 255$ $a \in [0,1]$
Operation:	$(\overline{f}) + 1 \rightarrow f$
Status Affected:	N, OV, C, DC, Z
Encoding:	`0110` `110a` `ffff` `ffff`
Description:	Location 'f' is negated using two's complement. The result is placed in the data memory location 'f'. If 'a' is 0, the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write register 'f'

Example: `NEGF REG, 1`

Before Instruction
REG = `0011 1010` [0x3A]

After Instruction
REG = `1100 0110` [0xC6]

NOP **No Operation**

Syntax:	[*label*] NOP
Operands:	None
Operation:	No operation
Status Affected:	None
Encoding:	`0000` `0000` `0000` `0000` `1111` `xxxx` `xxxx` `xxxx`
Description:	No operation.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	No operation	No operation	No operation

Example:

None.

POP — Pop Top of Return Stack

Syntax: [*label*] POP

Operands: None

Operation: (TOS) → bit bucket

Status Affected: None

Encoding:

0000	0000	0000	0110

Description: The TOS value is pulled off the return stack and is discarded. The TOS value then becomes the previous value that was pushed onto the return stack.
This instruction is provided to enable the user to properly manage the return stack to incorporate a software stack.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	No operation	POP TOS value	No operation

Example:

```
POP
GOTO    NEW
```

Before Instruction

TOS	=	0031A2h
Stack (1 level down)	=	014332h

After Instruction

TOS	=	014332h
PC	=	NEW

PUSH — Push Top of Return Stack

Syntax: [*label*] PUSH

Operands: None

Operation: (PC+2) → TOS

Status Affected: None

Encoding:

0000	0000	0000	01

Description: The PC+2 is pushed onto the to the return stack. The previous T value is pushed down on the st This instruction allows to implem a software stack by modifying T and then push it onto the return stack.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	PUSH PC+2 onto return stack	No operation	No operatic

Example:

```
PUSH
```

Before Instruction

TOS	=	00345Ah
PC	=	000124h

After Instruction

PC	=	000126h
TOS	=	000126h
Stack (1 level down)	=	00345Ah

RCALL Relative Call

Syntax:	[*label*] RCALL n
Operands:	$-1024 \le n \le 1023$
Operation:	$(PC) + 2 \rightarrow TOS$, $(PC) + 2 + 2n \rightarrow PC$
Status Affected:	None
Encoding:	`1101` `1nnn` `nnnn` `nnnn`
Description:	Subroutine call with a jump up to 1K from the current location. First, return address (PC+2) is pushed onto the stack. Then, add the 2's complement number '2n' to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be PC+2+2n. This instruction is a two-cycle instruction.
Words:	1
Cycles:	2

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'n' Push PC to stack	Process Data	Write to PC
No operation	No operation	No operation	No operation

Example: `HERE RCALL Jump`

Before Instruction
PC = Address `(HERE)`

After Instruction
PC = Address `(Jump)`
TOS = Address `(HERE+2)`

RESET Reset

Syntax:	[*label*] RESET
Operands:	None
Operation:	Reset all registers and flags that are affected by a $\overline{MCLR}$ Reset.
Status Affected:	All
Encoding:	`0000` `0000` `1111` `1111`
Description:	This instruction provides a way to execute a $\overline{MCLR}$ Reset in software.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Start reset	No operation	No operation

Example: `RESET`

After Instruction
Registers = Reset Value
Flags* = Reset Value

RETFIE **Return from Interrupt**

Syntax: [*label*] RETFIE [s]

Operands: s ∈ [0,1]

Operation: (TOS) → PC,
1 → GIE/GIEH or PEIE/GIEL,
if s = 1
(WS) → W,
(STATUSS) → STATUS,
(BSRS) → BSR,
PCLATU, PCLATH are unchanged.

Status Affected: GIE/GIEH, PEIE/GIEL.

Encoding:

0000	0000	0001	000s

Description: Return from Interrupt. Stack is popped and Top-of-Stack (TOS) is loaded into the PC. Interrupts are enabled by setting either the high or low priority global interrupt enable bit. If 's' = 1, the contents of the shadow registers WS, STATUSS and BSRS are loaded into their corresponding registers, W, STATUS and BSR. If 's' = 0, no update of these registers occurs (default).

Words: 1

Cycles: 2

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	No operation	No operation	pop PC from stack Set GIEH or GIEL
No operation	No operation	No operation	No operation

Example: `RETFIE 1`

After Interrupt

PC	=	TOS
W	=	WS
BSR	=	BSRS
STATUS	=	STATUSS
GIE/GIEH, PEIE/GIEL	=	1

RETLW **Return Literal to W**

Syntax: [*label*] RETLW k

Operands: 0 ≤ k ≤ 255

Operation: k → W,
(TOS) → PC,
PCLATU, PCLATH are unchanged

Status Affected: None

Encoding:

0000	1100	kkkk	kkkk

Description: W is loaded with the eight-bit literal 'k'. The program counter is loaded from the top of the stack (the return address). The high address latch (PCLATH) remains unchanged.

Words: 1

Cycles: 2

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'	Process Data	pop PC from stack, Write to W
No operation	No operation	No operation	No operation

Example:

```
    CALL TABLE ; W contains table
               ; offset value
               ; W now has
               ; table value
   :
TABLE
    ADDWF PCL  ; W = offset
    RETLW k0   ; Begin table
    RETLW k1   ;
   :
   :
    RETLW kn   ; End of table
```

Before Instruction
W = 0x07

After Instruction
W = value of kn

RETURN — Return from Subroutine

Syntax: [*label*] RETURN [s]

Operands: s ∈ [0,1]

Operation: (TOS) → PC,
if s = 1
(WS) → W,
(STATUSS) → STATUS,
(BSRS) → BSR,
PCLATU, PCLATH are unchanged

Status Affected: None

Encoding:

0000	0000	0001	001s

Description: Return from subroutine. The stack is popped and the top of the stack (TOS) is loaded into the program counter. If 's'= 1, the contents of the shadow registers WS, STATUSS and BSRS are loaded into their corresponding registers, W, STATUS and BSR. If 's' = 0, no update of these registers occurs (default).

Words: 1

Cycles: 2

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	No operation	Process Data	pop PC from stack
No operation	No operation	No operation	No operation

Example: `RETURN`

After Interrupt
PC = TOS

RLCF — Rotate Left f through Carry

Syntax: [*label*] RLCF f [,d [,a]

Operands: $0 \le f \le 255$
d ∈ [0,1]
a ∈ [0,1]

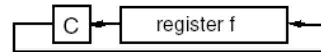
Operation: (f<n>) → dest<n+1>,
(f<7>) → C,
(C) → dest<0>

Status Affected: C, N, Z

Encoding:

0011	01da	ffff	ffff

Description: The contents of register 'f' are rotated one bit to the left through the Carry Flag. If 'd' is 0, the result is placed in W. If 'd' is 1, the result is stored back in register 'f' (default). If 'a' is 0, the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).



Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Example: `RLCF REG, 0, 0`

Before Instruction
REG = 1110 0110
C = 0

After Instruction
REG = 1110 0110
W = 1100 1100
C = 1

RLNCF **Rotate Left f (no carry)**

Syntax: [*label*] RLNCF f [,d [,a]

Operands: $0 \le f \le 255$
$d \in [0,1]$
$a \in [0,1]$

Operation: (f<n>) → dest<n+1>,
(f<7>) → dest<0>

Status Affected: N, Z

Encoding:

0100	01da	ffff	ffff

Description: The contents of register 'f' are rotated one bit to the left. If 'd' is 0, the result is placed in W. If 'd' is 1, the result is stored back in register 'f' (default). If 'a' is 0, the Access Bank will be selected, overriding the BSR value. If 'a' is 1, then the bank will be selected as per the BSR value (default).

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Example: `RLNCF REG, 1, 0`

Before Instruction
REG = `1010 1011`

After Instruction
REG = `0101 0111`

RRCF **Rotate Right f through Carry**

Syntax: [*label*] RRCF f [,d [,a]

Operands: $0 \le f \le 255$
$d \in [0,1]$
$a \in [0,1]$

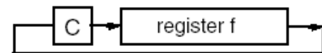
Operation: (f<n>) → dest<n-1>,
(f<0>) → C,
(C) → dest<7>

Status Affected: C, N, Z

Encoding:

0011	00da	ffff	ffff

Description: The contents of register 'f' are rotated one bit to the right through the Carry Flag. If 'd' is 0, the result is placed in W. If 'd' is 1, the result is placed back in register 'f' (default). If 'a' is 0, the Access Bank will be selected, overriding the BSR value. If 'a' is 1, then the bank will be selected as per the BSR value (default).



Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Example: `RRCF REG, 0, 0`

Before Instruction
REG = `1110 0110`
C = 0

After Instruction
REG = `1110 0110`
W = `0111 0011`
C = 0

RRNCF — Rotate Right f (no carry)

Syntax: [*label*] RRNCF f [,d [,a]

Operands: $0 \le f \le 255$
$d \in [0,1]$
$a \in [0,1]$

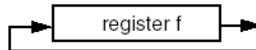
Operation: (f<n>) → dest<n-1>,
(f<0>) → dest<7>

Status Affected: N, Z

Encoding:

0100	00da	ffff	ffff

Description: The contents of register 'f' are rotated one bit to the right. If 'd' is 0, the result is placed in W. If 'd' is 1, the result is placed back in register 'f' (default). If 'a' is 0, the Access Bank will be selected, overriding the BSR value. If 'a' is 1, then the bank will be selected as per the BSR value (default).



Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Example 1: `RRNCF REG, 1, 0`

Before Instruction
REG = 1101 0111
After Instruction
REG = 1110 1011

Example 2: `RRNCF REG, 0, 0`

Before Instruction
W = ?
REG = 1101 0111
After Instruction
W = 1110 1011
REG = 1101 0111

SETF — Set f

Syntax: [*label*] SETF f [,a]

Operands: $0 \le f \le 255$
$a \in [0,1]$

Operation: FFh → f

Status Affected: None

Encoding:

0110	100a	ffff	ffff

Description: The contents of the specified register are set to FFh. If 'a' is 0, the Access Bank will be selected, overriding the BSR value. If 'a' is 1, then the bank will be selected as per the BSR value (default).

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write register 'f'

Example: `SETF REG,1`

Before Instruction
REG = 0x5A
After Instruction
REG = 0xFF

SLEEP **Enter SLEEP mode**

Syntax:	[*label*] SLEEP
Operands:	None
Operation:	00h → WDT, 0 → WDT postscaler, 1 → $\overline{TO}$, 0 → $\overline{PD}$
Status Affected:	$\overline{TO}$, $\overline{PD}$
Encoding:	`0000` `0000` `0000` `0011`
Description:	The power-down status bit ($\overline{PD}$) is cleared. The time-out status bit ($\overline{TO}$) is set. Watchdog Timer and its postscaler are cleared. The processor is put into SLEEP mode with the oscillator stopped.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	No operation	Process Data	Go to sleep

Example: `SLEEP`

Before Instruction
  $\overline{TO}$ = ?
  $\overline{PD}$ = ?

After Instruction
  $\overline{TO}$ = 1 †
  $\overline{PD}$ = 0

† If WDT causes wake-up, this bit is cleared.

SUBFWB **Subtract f from W with borrow**

Syntax:	[*label*] SUBFWB f [,d [,a]
Operands:	$0 \le f \le 255$ $d \in [0,1]$ $a \in [0,1]$
Operation:	(W) – (f) – ($\overline{C}$) → dest
Status Affected:	N, OV, C, DC, Z
Encoding:	`0101` `01da` `ffff` `ffff`
Description:	Subtract register 'f' and carry flag (borrow) from W (2's complement method). If 'd' is 0, the result is stored in W. If 'd' is 1, the result is stored in register 'f' (default). If 'a' is 0, the Access Bank will be selected, overriding the BSR value. If 'a' is 1, then the bank will be selected as per the BSR value (default).
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Example 1: `SUBFWB REG, 1, 0`

Before Instruction
  REG = 3
  W = 2
  C = 1

After Instruction
  REG = FF
  W = 2
  C = 0
  Z = 0
  N = 1 ; result is negative

Example 2: `SUBFWB REG, 0, 0`

Before Instruction
  REG = 2
  W = 5
  C = 1

After Instruction
  REG = 2
  W = 3
  C = 1
  Z = 0
  N = 0 ; result is positive

Example 3: `SUBFWB REG, 1, 0`

Before Instruction
  REG = 1
  W = 2
  C = 0

After Instruction
  REG = 0
  W = 2
  C = 1
  Z = 1 ; result is zero
  N = 0

SUBLW Subtract W from literal

Syntax: [*label*] SUBLW k

Operands: $0 \le k \le 255$

Operation: $k - (W) \rightarrow W$

Status Affected: N, OV, C, DC, Z

Encoding:

0000	1000	kkkk	kkkk

Description: W is subtracted from the eight-bit literal 'k'. The result is placed in W.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'	Process Data	Write to W

Example 1: `SUBLW 0x02`

Before Instruction

```
W = 1
C = ?
```

After Instruction

```
W = 1
C = 1     ; result is positive
Z = 0
N = 0
```

Example 2: `SUBLW 0x02`

Before Instruction

```
W = 2
C = ?
```

After Instruction

```
W = 0
C = 1     ; result is zero
Z = 1
N = 0
```

Example 3: `SUBLW 0x02`

Before Instruction

```
W = 3
C = ?
```

After Instruction

```
W = FF    ; (2's complement)
C = 0     ; result is negative
Z = 0
N = 1
```

SUBWF Subtract W from f

Syntax: [*label*] SUBWF f [,d [,a]

Operands: $0 \le f \le 255$
$d \in [0,1]$
$a \in [0,1]$

Operation: $(f) - (W) \rightarrow dest$

Status Affected: N, OV, C, DC, Z

Encoding:

0101	11da	ffff	ffff

Description: Subtract W from register 'f' (2's complement method). If 'd' is 0, the result is stored in W. If 'd' is 1, the result is stored back in register 'f' (default). If 'a' is 0, the Access Bank will be selected, overriding the BSR value. If 'a' is 1, then the bank will be selected as per the BSR value (default).

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Example 1: `SUBWF REG, 1, 0`

Before Instruction

```
REG = 3
W   = 2
C   = ?
```

After Instruction

```
REG = 1
W   = 2
C   = 1   ; result is positive
Z   = 0
N   = 0
```

Example 2: `SUBWF REG, 0, 0`

Before Instruction

```
REG = 2
W   = 2
C   = ?
```

After Instruction

```
REG = 2
W   = 0
C   = 1   ; result is zero
Z   = 1
N   = 0
```

Example 3: `SUBWF REG, 1, 0`

Before Instruction

```
REG = 1
W   = 2
C   = ?
```

After Instruction

```
REG = FFh ;(2's complement)
W   = 2
C   = 0   ; result is negative
Z   = 0
N   = 1
```

SUBWFB — Subtract W from f with Borrow

Syntax:	[*label*] SUBWFB f [,d [,a]
Operands:	$0 \le f \le 255$ $d \in [0,1]$ $a \in [0,1]$
Operation:	$(f) - (W) - (\overline{C}) \rightarrow dest$
Status Affected:	N, OV, C, DC, Z
Encoding:	`0101` `10da` `ffff` `ffff`
Description:	Subtract W and the carry flag (borrow) from register 'f' (2's complement method). If 'd' is 0, the result is stored in W. If 'd' is 1, the result is stored back in register 'f' (default). If 'a' is 0, the Access Bank will be selected, overriding the BSR value. If 'a' is 1, then the bank will be selected as per the BSR value (default).
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Example 1: `SUBWFB REG, 1, 0`

```
Before Instruction
    REG  =  0x19   (0001 1001)
    W    =  0x0D   (0000 1101)
    C    =  1
After Instruction
    REG  =  0x0C   (0000 1011)
    W    =  0x0D   (0000 1101)
    C    =  1
    Z    =  0
    N    =  0      ; result is positive
```

Example 2: `SUBWFB REG, 0, 0`

```
Before Instruction
    REG  =  0x1B   (0001 1011)
    W    =  0x1A   (0001 1010)
    C    =  0
After Instruction
    REG  =  0x1B   (0001 1011)
    W    =  0x00
    C    =  1
    Z    =  1      ; result is zero
    N    =  0
```

Example 3: `SUBWFB REG, 1, 0`

```
Before Instruction
    REG  =  0x03   (0000 0011)
    W    =  0x0E   (0000 1101)
    C    =  1
After Instruction
    REG  =  0xF5   (1111 0100)
                   ; [2's comp]
    W    =  0x0E   (0000 1101)
    C    =  0
    Z    =  0
    N    =  1      ; result is negative
```

SWAPF — Swap f

Syntax:	[*label*] SWAPF f [,d [,a]
Operands:	$0 \le f \le 255$ $d \in [0,1]$ $a \in [0,1]$
Operation:	$(f<3:0>) \rightarrow dest<7:4>$, $(f<7:4>) \rightarrow dest<3:0>$
Status Affected:	None
Encoding:	`0011` `10da` `ffff` `ffff`
Description:	The upper and lower nibbles of register 'f' are exchanged. If 'd' is 0, the result is placed in W. If 'd' is 1, the result is placed in register 'f' (default). If 'a' is 0, the Access Bank will be selected, overriding the BSR value. If 'a' is 1, then the bank will be selected as per the BSR value (default).
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Example: `SWAPF REG, 1, 0`

```
Before Instruction
    REG  =  0x53
After Instruction
    REG  =  0x35
```

TBLRD Table Read

Syntax: [*label*] TBLRD (*; *+; *-; +*)

Operands: None

Operation:
if TBLRD *,
(Prog Mem (TBLPTR)) → TABLAT;
TBLPTR - No Change;
if TBLRD *+,
(Prog Mem (TBLPTR)) → TABLAT;
(TBLPTR) +1 → TBLPTR;
if TBLRD *-,
(Prog Mem (TBLPTR)) → TABLAT;
(TBLPTR) -1 → TBLPTR;
if TBLRD +*,
(TBLPTR) +1 → TBLPTR;
(Prog Mem (TBLPTR)) → TABLAT;

Status Affected: None

Encoding:

0000	0000	0000	10nn nn=0 * =1 *+ =2 *- =3 +*

Description: This instruction is used to read the contents of Program Memory (P.M.). To address the program memory, a pointer called Table Pointer (TBLPTR) is used. The TBLPTR (a 21-bit pointer) points to each byte in the program memory. TBLPTR has a 2 Mbyte address range.

TBLPTR[0] = 0: Least Significant Byte of Program Memory Word

TBLPTR[0] = 1: Most Significant Byte of Program Memory Word

The `TBLRD` instruction can modify the value of TBLPTR as follows:

- no change
- post-increment
- post-decrement
- pre-increment

Words: 1

Cycles: 2

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	No operation	No operation	No operation
No operation	No operation (Read Program Memory)	No operation	No operation (Write TABLAT)

TBLRD Table Read (cont'd)

Example1: `TBLRD *+ ;`

Before Instruction

TABLAT	=	0x55
TBLPTR	=	0x00A356
MEMORY(0x00A356)	=	0x34

After Instruction

TABLAT	=	0x34
TBLPTR	=	0x00A357

Example2: `TBLRD +* ;`

Before Instruction

TABLAT	=	0xAA
TBLPTR	=	0x01A357
MEMORY(0x01A357)	=	0x12
MEMORY(0x01A358)	=	0x34

After Instruction

TABLAT	=	0x34
TBLPTR	=	0x01A358

TBLWT Table Write

Syntax:	[*label*] TBLWT (*; *+; *-; +*)
Operands:	None
Operation:	if TBLWT*, (TABLAT) → Holding Register; TBLPTR - No Change; if TBLWT*+, (TABLAT) → Holding Register; (TBLPTR) +1 → TBLPTR; if TBLWT*-, (TABLAT) → Holding Register; (TBLPTR) -1 → TBLPTR; if TBLWT+*, (TBLPTR) +1 → TBLPTR; (TABLAT) → Holding Register;
Status Affected:	None

Encoding:

0000	0000	0000	11nn nn=0 * =1 *+ =2 *- =3 +*

Description: This instruction uses the 3 LSbs of the TBLPTR to determine which of the 8 holding registers the TABLAT data is written to. The 8 holding registers are used to program the contents of Program Memory (P.M.). See Section 5.0 for information on writing to FLASH memory.

The TBLPTR (a 21-bit pointer) points to each byte in the program memory. TBLPTR has a 2 MBtye address range. The LSb of the TBLPTR selects which byte of the program memory location to access.

TBLPTR[0] = 0: Least Significant Byte of Program Memory Word

TBLPTR[0] = 1: Most Significant Byte of Program Memory Word

The `TBLWT` instruction can modify the value of TBLPTR as follows:

- no change
- post-increment
- post-decrement
- pre-increment

Words: 1

Cycles: 2

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	No operation	No operation	No operation
No operation	No operation (Read TABLAT)	No operation	No operation (Write to Holding Register or Memory)

TBLWT Table Write (Continued)

Example1: `TBLWT *+;`

Before Instruction

TABLAT	=	0x55
TBLPTR	=	0x00A356
HOLDING REGISTER (0x00A356)	=	0xFF

After Instructions (table write completion)

TABLAT	=	0x55
TBLPTR	=	0x00A357
HOLDING REGISTER (0x00A356)	=	0x55

Example 2: `TBLWT +*;`

Before Instruction

TABLAT	=	0x34
TBLPTR	=	0x01389A
HOLDING REGISTER (0x01389A)	=	0xFF
HOLDING REGISTER (0x01389B)	=	0xFF

After Instruction (table write completion)

TABLAT	=	0x34
TBLPTR	=	0x01389B
HOLDING REGISTER (0x01389A)	=	0xFF
HOLDING REGISTER (0x01389B)	=	0x34

TSTFSZ — Test f, skip if 0

Syntax: [*label*] TSTFSZ f [,a]

Operands: $0 \le f \le 255$
$a \in [0,1]$

Operation: skip if f = 0

Status Affected: None

Encoding:

0110	011a	ffff	ffff

Description: If 'f' = 0, the next instruction, fetched during the current instruction execution, is discarded and a NOP is executed, making this a two-cycle instruction. If 'a' is 0, the Access Bank will be selected, overriding the BSR value. If 'a' is 1, then the bank will be selected as per the BSR value (default).

Words: 1

Cycles: 1(2)
Note: 3 cycles if skip and followed by a 2-word instruction.

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	No operation

If skip:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation

If skip and followed by 2-word instruction:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation
No operation	No operation	No operation	No operation

Example:

```
HERE    TSTFSZ  CNT, 1
NZERO     :
ZERO    :
```

Before Instruction
PC = Address (HERE)

After Instruction
If CNT = 0x00,
PC = Address (ZERO)
If CNT ≠ 0x00,
PC = Address (NZERO)

XORLW — Exclusive OR literal with W

Syntax: [*label*] XORLW k

Operands: $0 \le k \le 255$

Operation: (W) .XOR. k $\rightarrow$ W

Status Affected: N, Z

Encoding:

0000	1010	kkkk	kkkk

Description: The contents of W are XORed with the 8-bit literal 'k'. The result is placed in W.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'	Process Data	Write to W

Example: `XORLW 0xAF`

Before Instruction
W = 0xB5

After Instruction
W = 0x1A

XORWF **Exclusive OR W with f**

Syntax:	[*label*] XORWF f [,d [,a]
Operands:	$0 \le f \le 255$ $d \in [0,1]$ $a \in [0,1]$
Operation:	(W) .XOR. (f) $\rightarrow$ dest
Status Affected:	N, Z
Encoding:	`0001` `10da` `ffff` `ffff`
Description:	Exclusive OR the contents of W with register 'f'. If 'd' is 0, the result is stored in W. If 'd' is 1, the result is stored back in the register 'f' (default). If 'a' is 0, the Access Bank will be selected, overriding the BSR value. If 'a' is 1, then the bank will be selected as per the BSR value (default).
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Example: `XORWF REG, 1, 0`

Before Instruction
REG = 0xAF
W = 0xB5

After Instruction
REG = 0x1A
W = 0xB5