Procesorski sistemi v TK I

Uvod v mikrokrmilnik 8051

Priročnik za laboratorijske vaje

Matej Zajc

Ljubljana, 2005

Uvod

Priročnik Uvod v mikrokrmilnik 8051 je namenjen študentom 3. letnika univerzitetnega študija na smeri telekomunikacije, kot učni pripomoček pri laboratorijskih vajah predmeta Procesorski sistemi v TK I.

Priročnik predstavi zgradbo in programski model mikrokrmilnika 8051. Povzete so bistvene teme, potrebne za hitro spoznavanje zgradbe in delovanja mikrokrmilnika.

Priročnik je tematsko razdeljen na dva dela. Prvi obsega poglavji 1 in 2 ter obravnava zgradbo mikrokrmilnika 8051 ter programiranje v zbirnem jeziku. Drugi del pokriva programiranje mikrokrmilnika v jeziku C ter uporabo vhodno/izhodnih vrat mikrokrmilnika, časovnikov, števcev ter serijskih vrat. V dodatkih se nahajajo opisi ukazov zbirnega jezika, operatorji jezika C ter kratek uvod v programski paket Keil µVision. Podrobnosti zgradbe, delovanja mikrokrmilnika ter programiranje mikrokrmilnika v zbirnem jeziku in jeziku C se nahajajo v priročnikih, ki so zbrani v Literaturi.

Laboratorijske vaje potekajo na programskem paketu Keil µVision. Programski paket Keil µVision je moderno orodje, ki omogoča programiranje mikrokrmilnika v zbirnem jeziku kot tudi v jeziku C. Na voljo je v omejeni brezplačni različici (http://www.keil.com/demo/eval/c51.htm).

Praktično delo z mikrokrmilnikom 8051 poteka na vezjih z mikrokrmilnikom T89C51RD2, ki jih je izdelalo podjetje IPS d.o.o. Navodila za delo z njimi so podana na laboratorijskih vajah.

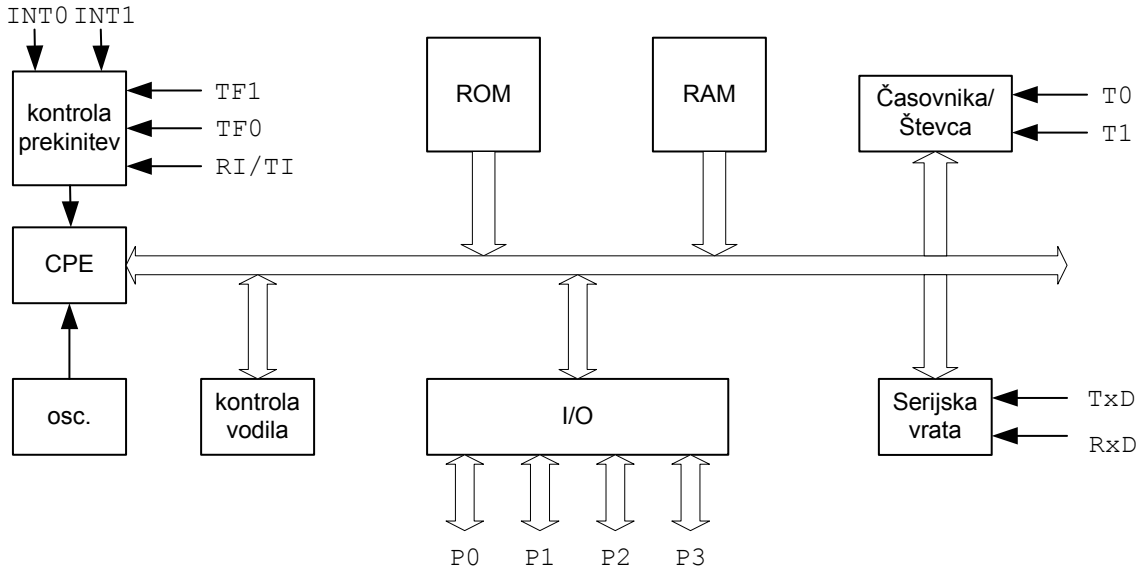
Matej Zajc

Kazalo

1 Arhitektura mikrokrmilnika 8051

Mikrokrmilnik 8051 je 8 bitni mikroprocesor. Razvili so ga pri podjetju Intel konec sedemdesetih let. Mikrokrmilnik je postal zelo popularen, njegove različice pa danes izdeluje več kot deset proizvajalcev v več kot sto različicah [Keil, 8052].

V okviru vaj bomo spoznali osnovno arhitekturo 8051. Poudarek bo predvsem na konceptih, ki so podobni tudi pri ostalih mikrokrmilnikih.



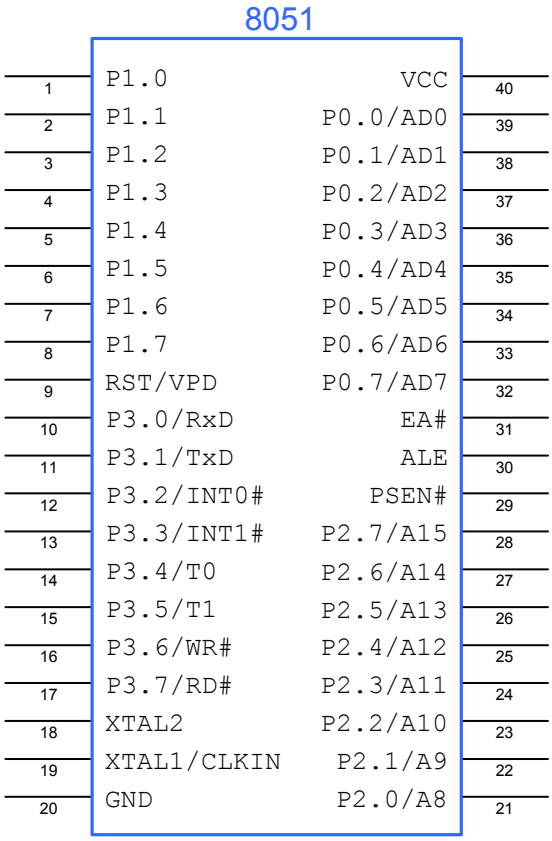
Slika 1: Poenostavljena arhitektura mikrokrmilnika 8051 [C51_AO]

Osnovna izvedenka mikrokrmilnika 8051 ima 4k besed programskega pomnilnika (ROM) ter 128 besed podatkovnega pomnilnika (RAM) ter različne periferne enote: dva 16 bitna časovnika/števca, serijska vrata (enoto UART) ter štiri vzporedna 8 bitna vhodno/izhodna vrata. Na trgu je mnogo različic, katerih arhitektura je nadgrajena tudi z drugimi perifernimi enotami, analogno-digitalnimi pretvorniki, vmesniki za I^2C vodila, itd. [Keil, 8052].

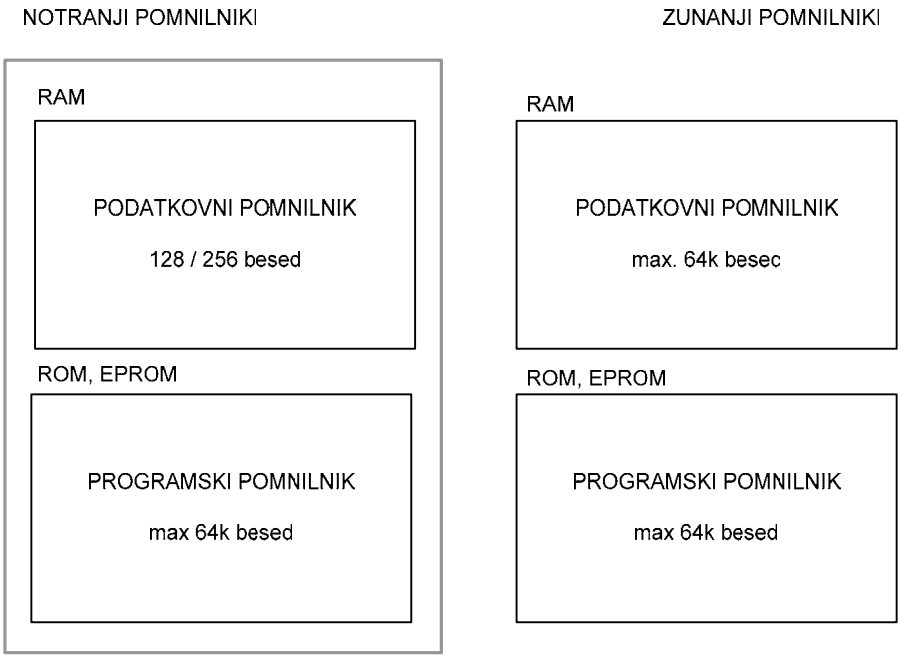
8051 ima 8 bitno podatkovno vodilo. S 16 bitnim programskim števcem lahko naslovimo 64k besed zunanjega programskega pomnilnika, s 16 bitnim podatkovnim števcem pa do 64k zunanjega podatkovnega pomnilnika.

1.1 Organizacija pomnilnika

Arhitektura 8051 podpira več pomnilnikov za hranjenje programske kode in podatkov. Slika 3 prikazuje organizacijo pomnilnikov mikrokrmilnika 8051. Pomnilniški prostor v grobem delimo na programski pomnilnik (en ali več) in podatkovni pomnilnik (en ali več) ter na notranji pomnilnik (nahaja se na čipu, torej je del arhitekture mikrokrmilnika) in zunanji pomnilnik (nahaja se poleg mikrokrmilnika).



Slika 2: Dostopni signali



Slika 3: Organizacija pomnilniškega prostora

1.1.1 Programski pomnilnik

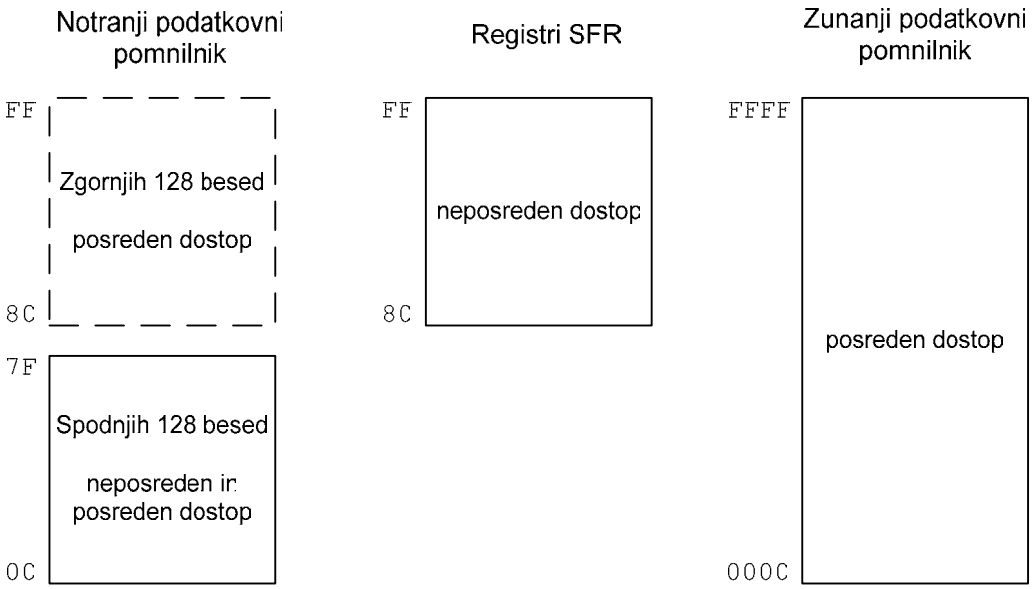
Programski pomnilnik lahko med izvajanjem programa le beremo, pisati vanj ne moremo. Poleg programske kode lahko v programski pomnilnik shranimo tudi konstante. Programski pomnilnik se nahaja znotraj 8051 CPE (centralne procesne enote), kot del notranjega pomnilnika. Lahko je dodan tudi zunanji programski pomnilnik. Velikost in razporeditev programskega pomnilnika je odvisna od različice mikrokrmilnika 8051. Velikost programskega pomnilnika je omejena na 64k besed (2^{16}= 65536). Velikost je omejena s 16 bitnim programskim števcem, ki naslavlja programski del pomnilnika.

Do programskega pomnilnika dostopimo z rezervirano besedo `code`.

1.1.2 Podatkovni pomnilnik

Podatkovni pomnilnik je bralno-pisalni. Slika 4 prikazuje delitev podatkovnega pomnilnika. Podatkovni pomnilnik sestavljajo:

- notranji podatkovni pomnilnik
- registri SFR (*Special Function Registers*)
- zunanji podatkovni pomnilnik



Slika 4: Podatkovni pomnilnik

Notranji podatkovni pomnilnik se nahaja znotraj 8051 CPE in je omejen na 256 besed, ki jih razdelimo na:

- spodnjih 128 besed
- zgornjih 128 besed[1]

Slika 5 prikazuje zgradbo spodnjih 128 besed notranjega podatkovnega pomnilnika:

- splošno namenski registri,
- bitni pomnilnik in

[1] Osnovna arhitektura 8051 ima samo spodnjih 128 besed notranjega podatkovnega pomnilnika.

- splošno namenski pomnilnik.

Registrov `Rn` je 32 organiziranih v 4 skupine registrov po osem registrov (`R0 – R7`)[2].

Bitno naslovljiv pomnilnik (*bit addressable memory*) zaseda 128 bitov (16 besed). Preostali del, 80 besed, pa predstavlja splošno namenski pomnilnik, kjer so shranjeni podatki.

Slika 6 prikazuje naslove registrov SFR. Tabela 2 podaja pomen kratic registrov SFR. Podrobnejši opis in uporabo bomo spoznali kasneje.

Zgornjih 128 besed notranjega podatkovnega pomnilnika se naslovno prekriva z 128 besedami pomnilnega prostora, kjer se nahajajo registri SFR (*Special Function Registers*). Ker se zgornjih 128 besed prekriva z registri SFR (imajo enake naslove: `80h – FFh`) lahko ta del notranjega podatkovnega pomnilnika naslovimo samo posredno. Registre SFR pa naslovimo samo neposredno. Tako je odvisno od načina naslavljanja ali gre za zgornjih 128 besed notranjega podatkovnega pomnilnika ali za registre SFR.

Slika 5: Spodnjih 128 besed notranjega podatkovnega pomnilnika

Notranji podatkovni je tako pomnilnik razdeljen na tri prekrivajoče enote, ki jih določajo naslednje rezervirane besede:

- `data` določa spodnjih 128 besed pomnilnika, naslovimo ga z neposrednim naslavljanjem.
- `idata` določa vseh 256 besed notranjega podatkovnega pomnilnika, do podatkov lahko dostopimo samo s posrednim naslavljanjem.
- `bdata` določa 16 besed pomnilnika, ki ga lahko naslovimo po posameznih bitih, nahaja se na naslovih (`20h – 2Fh`), podatke shranjene v tem delu pomnilnika lahko naslovimo tudi po posameznih bitih.

Zunanji podatkovni pomnilnik je prav tako bralno-pisalni. V primerjavi z notranjim podatkovnim pomnilnikom je dostop mnogo počasnejši. Dostop do zunanjega podatkovnega pomnilnika je posreden s 16 bitnim podatkovnim kazalcem `DPTR`, s katerim lahko naslovimo 64 k besed.

Dve rezervirani besedi določata zunanji podatkovni pomnilnik:

- `xdata` določa celoten zunanji podatkovni pomnilnik (do 64 k besed).

[2] Med skupinami registrov izbiramo z vrednostjo bitov `rs0` in `rs1` v registru `psw` (glej Tabela 3 in Tabela 4).

- `pdata` določa prvih 256 besed zunanjega podatkovnega pomnilnika. Ta del zunanjega pomnilnika lahko naslovimo tudi z 8 bitnimi registri `R0` in `R1` izbranega nabora registrov (glej načine naslavljanj).

`code`	programski pomnilnik (*program memory*) `MOVC A, @A+DPTR`
`data`	notranji podatkovni pomnilnik - neposreden dostop (*directly addressable internal data memory*) (spodnjih 128 besed), `MOV A, 30h`
`idata`	celoten notranji pomnilnik - posredni dostop (*indirectly addressable internal data memory*) (256 besed), `MOV A, @Rn`
`bdata`	bitni pomnilnik (*bit-addressable internal data memory*), (16 besed) `SETB 07h`
`xdata`	zunanji podatkovni pomnilnik (*external data memory*) (64 k besed) `MOVX A, @DPTR`
`pdata`	prvih 256 besed zunanjega podatkovnega pomnilnika (*paged external data memory*) (256 besed) `MOVX A, @Rn`

Tabela 1: Rezervirane besede za deklaracijo pomnilnika [C51]

1.2 Registri SFR (Special Function Registers)

Mikrokrmilnik 8051 pozna več načinov delovanja. Programer lahko preveri ali spremeni režim delovanja mikrokrmilnika s spreminjanjem vrednosti registrov SFR. Registri SFR omogočajo dostop do vhodno/izhodnih vrat mikrokrmilnika, nastavitve časovnikov in števcev, nastavitve serijskih vrat, nastavitve prekinitev, itd.

Registri SFR so del notranjega pomnilnika (Slika 4). Nahajajo se na naslovih med 80h in FFh, pokrivajo 128 pomnilniških lokacij (besed), vendar je uporabljenih le 21. Vsi ostali naslovi so neveljavni[3]. Vsak register ima svoje ime ter svoj naslov. Slika 6 prikazuje razporeditev registrov SFR v pomnilniškem prostoru ter njihove oznake. Registrom v prvem stolpcu tabele lahko spreminjamo tudi posamezne bite. 16 bitne registre sestavljata po dva SFR registra, na primer podatkovni števec sestavljata DPH in DPL, kjer DPH hrani zgornjo besedo, DPL pa spodnjo besedo podatkovnega števca.

Registri ACC, B in PSW, so opisani v nadaljevanju. Ostale registre SFR bomo spoznali v ostalih poglavjih. Opis in delovanje registrov SFR se nahaja v [C51_PG, C51_HD].

[3] Posamezne izvedenke osnovne arhitekture poznajo dodatne registre SFR.

P0	vrata 0 (*Port 0*)
SP	skladovni kazalec (*Stack Pointer*)
DPL/DPH	podatkovni števec DPTR (*Data Pointer*)
PCON	nadzor delovanja (*Power Control*)
TCON	nadzor časovnika (*Timer Control*)
TMOD	nastavitve časovnika (*Timer Mode*)
TL0/TH0	časovnik/števec 0 (*Timer 0 / Counter 0*)
TL1/TH1	časovnik/števec 1 (*Timer 1 / Counter 1*)
P1	vrata 1 (*Port 1*)
SCON	nadzor serijske komunikacije (*Serial Control*)
SBUF	serijski medpomnilnik (*Serial Buffer*)
P2	vrata 2 (*Port 2*)
IE	omogoči prekinitev (*Interrupt Enable*)
P3	vrata 3 (*Port 3*)
IP	prioriteta prekinitev (*Interrupt Priority*)
PSW	register stanja (*Program Status Register*)
ACC	akumulator A (*Accumulator*)
B	register B

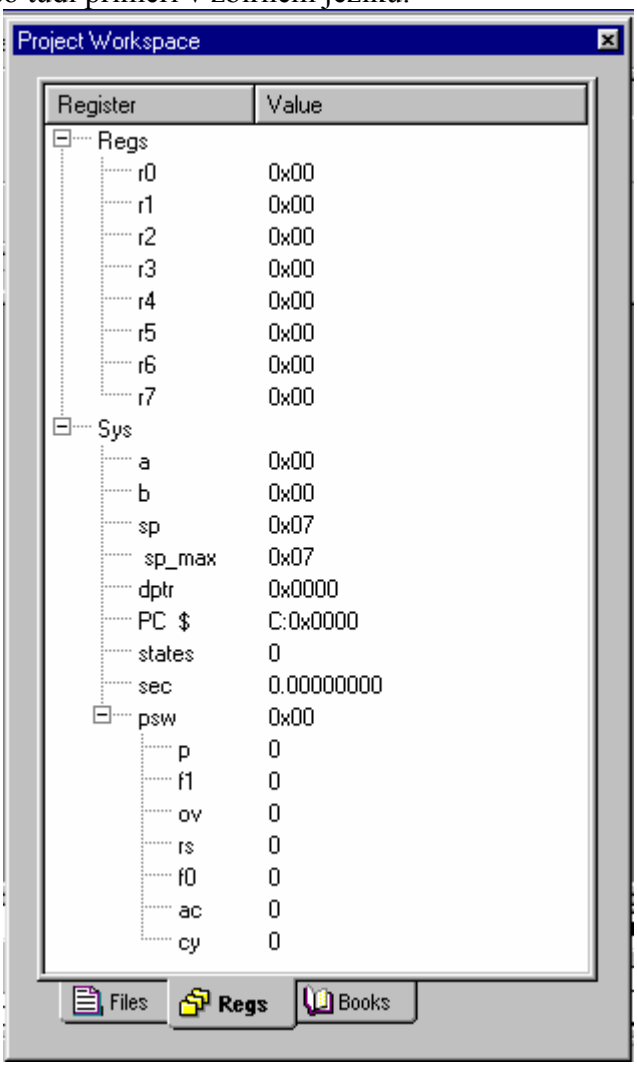
Tabela 2: Opis registrov SFR [C51_PG]

80	P0	SP	DPL	DPH				PCON	87
88	TCON	TMOD	TL0	TL1	TH0	TH1			8F
90	P1								97
98	SCON	SBUF							9F
A0	P2								A7
A8	IE								AF
B0	P3								B7
B8	IP								BF
C0									C7
C8									CF
D0	PSW								D7
D8									DF
E0	ACC								E7
E8									EF
F0	B								F7
F8									FF

Slika 6: Registri SFR [C51_PG]

1.3 Registri

Slika 7 prikazuje registre v programskem okolju Keil µVision. Registri so opisani v nadaljevanju. Podani so tudi primeri v zbirnem jeziku.



Slika 7: Registri v okolju Keil µVision

1.3.1 Akumulator A

Akumulator `A` je 8 bitni splošno namenski register. Več kot polovica od 255 ukazov uporablja akumulator. Nahaja se med registri SFR na naslovu `E0h` (Slika 6).

```
MOV A, #20h
MOV E0h, #20h
```

V obeh primerih je rezultat enak, v akumulator, t.j. na naslov `E0h`, se shrani vrednost `20h`. Prvi način je primernejši saj je ukaz dolg dve besedi, medtem ko je v drugem primeru dolg tri besede.

1.3.2 Registri R

Registri `Rn` imajo oznake `R0`, `R1`, `R2...R7`. Registri `Rn` so pomožni registri pri izvrševanju mnogih operacij. Največkrat nam služijo za hranjenje vmesnih rezultatov.

```
MOV A, R4
```

Po izvršenem ukazu ima akumulator `A` vrednost registra `R4`.

Pri mikrokrmilniku 8051 imamo na razpolago 4 skupine registrov `Rn` (Slika 5) Skupino registrov `Rn` izbiramo z bitoma `rs0` in `rs1`, ki sta del registra stanja `PSW` (Tabela 4).

1.3.3 Register B

Register `B` je 8 bitni register podobno kot akumulator `A`. Uporabljamo ga pri dveh ukazih: množenju in deljenju. Nahaja se med SFR registri na naslovu `F0h` (Slika 6)

```
MUL AB
DIV AB
```

Sicer pa lahko register `B` uporabljamo podobno kot registre `Rn` za hranjenje vmesnih rezultatov. Register `B` se nahaja na naslovu `F0h`, med SFR registri (Slika 6).

1.3.4 Podatkovni števec DPTR

Podatkovni števec je edini uporabniku dostopen 16 bitni register, saj so akumulator, registri `Rn` in register `B` so vsi 8 bitni registri. `DPTR` sestavljata dva osembitna SFR registra `DPL` in `DPH` (Slika 6).

Podatkovni števec uporabljajo ukazi za naslavljanje zunanjega podatkovnega pomnilnika. Vsebuje lahko vrednosti med `0000h` in `FFFFh`.

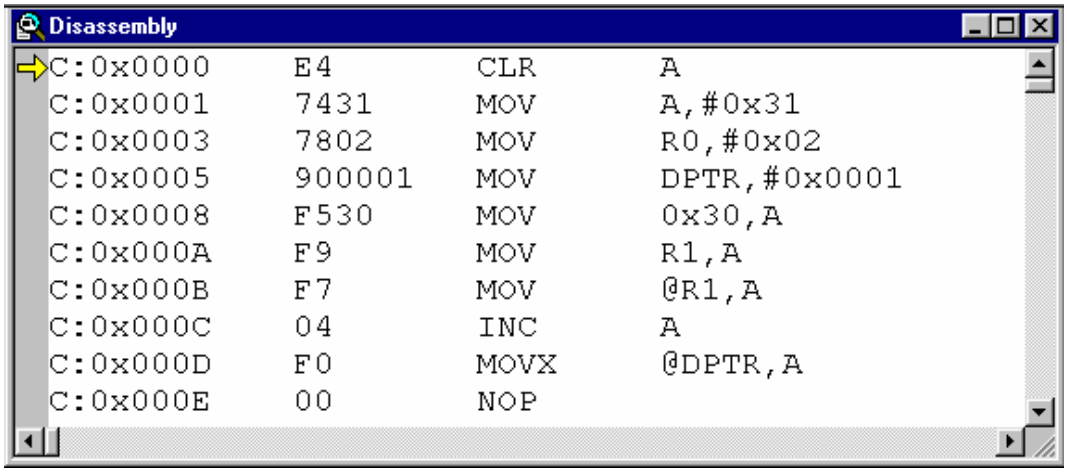
1.3.5 Programski števec PC

Programski števec vsebuje 16 bitni naslov naslednjega ukaza v programskem pomnilniku. Ob inicializaciji 8051 se vsebina programskega števca nastavi na `0000h`. Vrednost programskega števca se po vsaki operaciji poveča, tako da kaže na naslednji ukaz (Slika 8).

Vrednosti programskega števca ne moremo neposredno nastavljati. Vrednost programskega števca lahko spremenimo z vejitvami, skočnimi ukazi ter s klicem podprograma (podpoglavje 2.3).

Slika 8 prikazuje sled (*disassembly*) preprostega programa. V prvem stolpcu se nahaja vrednost programskega števca. Vidimo, da programski števec ne zavzame vseh vrednosti. Programski števec kaže vedno naslov naslednjega ukaza. Ker lahko posamezen ukaz zaseda od enega do treh besed se temu primerno spreminja tudi vrednost programskega števca. Na primer, prvi ukaz zaseda eno besedo (`E4`), zato je vrednost programskega števca enaka `0x0001`[4]. Drugi ukaz zaseda dve besedi (`74 31`), zato je po izvedenem ukazu vrednost programskega števca enaka `0x0003`. Četrti ukaz pa zaseda tri besede (`90 00 01`). Vsebina programskega števca se zato poveča za tri. Vrednost programskega števca je enaka `0x0008`.

[4] `0x0001` ustreza šestnajstiškemu zapisu števila 1 ali `0001h`.



Slika 8: Sled preprostega programa

1.3.6 Skladovni kazalec SP

Skladovni kazalec je 8 bitni register. Skladovni kazalec kaže trenutno lokacijo na skladu. Sklad uporabljamo ponavadi ob klicu podprograma (`LCALL`).

Ko želimo shraniti vrednost na sklad z ukazom `PUSH`, procesor poveča kazalec za ena in shrani vrednost. Ko želimo prebrati vrednost s sklada z ukazom `POP`, procesor prebere vrednost z naslova kamor kaže skladovni kazalec, ter ga zmanjša za ena.

```
PUSH PSW
```

Ukaz shrani vrednosti registra `PSW` na sklad. Naslov kamor shrani vrednost registra `PSW` določa vrednost skladovnega kazalca `SP`.

Na primer, ob inicializaciji 8051 kaže skladovni kazalec (`SP`) na naslov `07h` (Slika 7). Če bi želeli shraniti vrednost na sklad, bi se vrednost shranila v notranji pomnilnik na naslov `08h`, saj se vrednost skladovnega kazalca najprej poveča za ena. Sklad deluje po principu LIFO (*last in first out*). Zadnji podatek, ki smo ga zapisali na sklad preberemo s sklada kot prvega.

Skladovni kazalec neposredno spreminjajo naslednji ukazi: `PUSH`, `POP`, `ACALL`, `LCALL`, `RET` in `RETI`. Uporaba skladovnega kazalca je opisana v podpoglavju 2.4.

Naslov `08h` pa je naslov tudi prvega registra `R1` v drugi skupini registrov (Slika 5). Druga, tretja in četrta skupina registrov (Slika 5) se uporabljajo tudi za sklad.

1.3.7 Register stanja PSW

Register stanja (*Program Status Word*) sestavljajo kontrolni signali, ki opisujejo stanje mikrokrmilnika. Nahaja se na naslovu `D0h` med registri SFR (Slika 6).

Prenos `cy` nastavljajo ALU operacije. Prenos `cy` se postavi na ena, če je prišlo do prenosa pri aritmetičnih operacijah. Poleg tega pa se bit `cy` uporablja tudi za hranjenje rezultata (kot akumulator) pri logičnih operacijah. Aritmetične operacije nastavljajo tudi bita `ac` in `ov`. Polovični prenos `ac` uporabljamo pri BCD aritmetiki, preliv `ov` pa uporabljamo pri aritmetiki s predznačenimi števili.

zastavica	opis	bit
cy	prenos (*carry*)	7
ac	polovični prenos (*auxiliary carry*)	6
f0	splošno namenska statusna zastavica	5
rs1	nabor registrov bit 1	4
rs0	nabor registrov bit 0	3
ov	preliv (*overflow*)	2
f1	določi uporabnik	1
p	pariteta (*parity*)	0

Tabela 3: Register stanja (PSW)

Pariteta `p` se nastavlja glede na stanje akumulatorja: 1 če je v akumulatorju liho število enic in 0 če je v akumulatorju parno število enic.

rs1	rs0	nabor registrov
0	0	0
0	1	1
1	0	2
1	1	3

Tabela 4: Izbor skupine registrov z zastavicama rs0 in rs1

Z bitoma `rs1` in bit `rs0` izbiramo med štirimi skupinami registrov `Rn`. Z ukazoma

```
SETB PSW.3
SETB PSW.4
```

postavimo bita `rs1` in bit `rs0` na ena ter tako izberemo tretjo skupino registrov (Slika 5).

1.4 Naloge

1. S pomočjo priročnikov ugotovite velikost notranjega podatkovnega in programskega pomnilnika za mikrokrmilnik Atmel T89C51RD2.

2. Določite naslov registra `R4` glede na stanje zastavic `rs0` in `rs1`.

3. S pomočjo tabele ukazov zbirnega jezika ugotovite kateri ukazi vplivajo na stanje zastavic `cy` in `ov`.

2 Programiranje v zbirnem jeziku

2.1 Nabor ukazov

Ukaz v zbirnem jeziku sestavljata mnemonična koda in operand, v strojni kodi pa ukazna koda in operand (Tabela 5).

Vrnimo se na primer preprostega programa v prejšnjem podpoglavju (Slika 8). V drugem stolpcu se nahaja program zapisan v strojnem jeziku. Ukaz sestavljata ukazna koda in pripadajoči operand.

V tretjem in četrtem stolpcu je program zapisan v zbirnem jeziku. Sestavljata ga mnemonična koda (mnemonik) in operand. Operand je lahko konstanta, naslov podatka ali naslov programskega pomnilnika.

V strojni kodi je način naslavljanja zapisan v ukazni kodi, medtem ko je v zbirniku način naslavljanja razviden iz operanda (Tabela 5). V prvem primeru se v akumulator zapiše vrednost `32h` (neposredno naslavljanje), v drugem primeru pa se v akumulator zapiše vrednost, ki je shranjena na naslovu `32h` notranjega podatkovnega pomnilnika (posredno naslavljanje). Načini naslavljanj so podani v (Tabela 6). Ukazi zbirnega jezika so zbrani v dodatku A.

zbirni jezik	strojni jezik
`MOV A, #32h`	`74 32`
`MOV A, 32h`	`E5 32`

Tabela 5: Primerjava zapisa v zbirnem jeziku in v strojnem jeziku

Na tem mestu velja omeniti še zapis števil. Če številu dodamo znak `H` ali `h` je le to zapisano v šestnajstiškem zapisu (npr. `12h`). Kadar želimo število zapisati v binarnem zapisu dodamo znak `B` ali `b` (npr. `00001011B`). Če pa je število desetiško ne dodamo nič (npr. 23).

Pri tem moramo opozoriti še na eno posebnost okolja Keil µVision. V primeru zapisa šestnajstiškega števila, ki se začne s črko, moramo na prvo mesto dodati ničlo. Tako na primer `D3h` zapišemo kot `0D3h`. Kot smo že zapisali, lahko šestnajstiško število zapišemo tudi kot `0xD3`.

2.2 Načini naslavljanj

Način naslavljanja pomeni način dostopa do podane pomnilniške lokacije oziroma podatka. Načinov naslavljanj je mnogo. Vsak mikroprocesor ali mikrokrmilnik ima svoj nabor načinov naslavljanj. Osnovni načini naslavljanj so enaki ali podobni pri vseh mikroprocesorjih, čeprav

jih različni proizvajalci imenujejo različno. V nadaljevanju so predstavljeni načini naslavljanj pri mikrokrmilniku 8051.

vsebovano (*register specific, inherent*)	`CLR A`
takojšnje naslavljanje (*immediate*)	`MOV A, #20h`
neposredno naslavljanje (*direct*)	`MOV A, 30h`
posredno naslavljanje (*indirect*) - notranji pomnilnik	`MOV A, @R0`
posredno naslavljanje (*indirect*) - zunanji pomnilnik	`MOVX A, @DPTR`
	`MOVX A, @R0`
registrsko naslavljanje (*register instructions*)	`MOV A, R0`
indeksno naslavljanje (*indexed*)	`MOVC A, @A+DPTR`

Tabela 6: Načini naslavljanj pri 8051

2.2.1 Vsebovano (*Register specific - Inherent addressing*)

Vsebovano naslavljanje najdemo pri ukazih, ki so specifični za določen register ali akumulator. Pri vsebovanem naslavljanju je operand vsebovan v ukazni kodi. Ti ukazi so enobesedni. Ukazi tako ne potrebujejo dodatnega operanda, kot na primer `CLR A` in `INC A`.

```
CLR A
```

Po izvršenem ukazu bo v akumulatorju vrednost `00h`.

2.2.2 Takojšnje naslavljanje (*Immediate addressing*)

Pri takojšnjem naslavljanju se nahaja podatek v programskem pomnilniku takoj za ukazno kodo. V tem primeru je operand vrednost in ne naslov, ker je pred številom znak `#`.

```
MOV A, #20h
```

Po izvršenem ukazu bo v akumulatorju vrednost `20h`. Ukaz je dvobeseden, sestavljata ga ukazna koda in operand, ki je shranjen v programskem pomnilniku. V primeru, da je operand dvobesedno število bi ukaz zasedal tri besede.

Takojšnje naslavljanje je zelo hitro, saj je vrednost del ukaza. Ker je vrednost določena ob prevajanju programa je med izvajanjem ne moremo spreminjati.

2.2.3 Neposredno naslavljanje (*Direct addressing*)

Pri neposrednem naslavljanju predstavlja operand naslov notranjega podatkovnega pomnilnika, kjer se nahaja podatek. Operand je osembitno število. Z neposrednim naslavljanjem lahko dostopimo v notranji podatkovni pomnilnik od naslova `00h` do `7Fh` ter registre SFR.

```
MOV A, 30h
```

Ukaz prebere vrednost na naslovu 30h notranjega pomnilnika in ga shrani v akumulator.

Čeprav ni podana vrednost temveč naslov je neposredno naslavljanje hitro, saj je vrednost shranjena v notranjem podatkovnem pomnilniku.

2.2.4 Registrsko naslavljanje (*Register instructions*)

Registrsko naslavljanje uporablja registre `Rn` (`R0-R7`). Registrsko naslavljanje zaseda samo eno besedo. Zadnji trije biti ukazne kode vsebujejo informacijo za katerega izmed sedmih registrov gre.

```
MOV A, R6
```

Po izvršenem ukazu se v akumulatorju shrani vsebina (kopija) registra `R6`.

2.2.5 Posredno naslavljanje (*Indirect addressing*)

Pri posrednem naslavljanju operand ni naslov. Naslov predstavlja vsebina registra, ki je podan kot operand. Znak `@` nam pove, da gre za posredno naslavljanje. Naslov je lahko 8 biten v primeru, da uporabljamo registra `R0` in `R1` ali pa 16 biten v primeru `DPTR` registra.

Pri notranjem podatkovnem pomnilniku lahko s posrednim naslavljanjem naslovimo naslove `00h-FFh`.

```
MOV A, @R0
```

Vrednost registra `R0` je naslov, kjer se nahaja vrednost, ki se shrani v akumulator. Po izvršenem ukazu bo v akumulatorju vrednost lokacije notranjega pomnilnika, na katero kaže vsebina registra `R0`.

Za dostop do zunanjega pomnilnika uporabljamo poseben ukaz `MOVX`. Ukazi za delo z zunanjim pomnilnikom uporabljajo podatkovni števec `DPTR`. V podatkovni števec moramo predhodno shraniti naslov lokacije zunanjega pomnilnika katero želimo brati ali v njo pisati. Vsebina, ki jo hrani podatkovni števec je 16 bitni naslov zunanjega pomnilnika. Z zunanjim posrednim naslavljanjem lahko naslovimo naslove `0000h` do `FFFFh`, kar predstavlja 64k besed zunanjega pomnilnika (Slika 3).

Pri naslavljanju zunanjega pomnilnika s posrednim naslavljanjem imamo na voljo

```
MOVX A, @DPTR
MOVX @DPTR, A
```

S prvim ukazom beremo vsebino zunanjega pomnilnika v akumulator, z drugim pa v zunanji pomnilnik pišemo vsebino akumulatorja.

Zunanji pomnilnik lahko dostopimo s posrednim naslavljanjem tudi z registroma `R0` in `R1`. V tem primeru smo omejeni na 256 besed zunanjega pomnilnika (naslovi `00h` do `FFh`).

```
MOVX @R0, A
```

Zgornji ukaz bo zapisal vsebino akumulatorja na lokacijo zunanjega pomnilnika, katere naslov je shranjen v registru `R0`.

2.2.6 Indeksno naslavljanje (*Indexed addressing*)

Indeksno naslavljanje lahko uporabimo samo za branje programskega pomnilnika. Za dostop do zunanjega pomnilnika uporabljamo poseben ukaz `MOVC`. Ta način naslavljanja je namenjen branju tabel s konstantami, nizov, itd., ki so shranjeni v programskem pomnilniku.

Pri indeksnem naslavljanju uporabimo 16 bitni register `DPTR` ali pa `PC`, ter vanj vpišemo 16 bitni naslov prvega elementa tabele. Akumulator `A` uporabimo kot indeks s katerim dostopamo do posameznih elementov tabele[5].

```
MOVC A, @A+DPTR
MOVC A, @A+PC
```

Z ukazom beremo vsebino programskega pomnilnika v akumulator, kjer je naslov lokacije določen z vrednostjo `A+DPTR`, oziroma `A+PC`.

2.2.7 Naloge

4. Aritmetična operacija seštevanja podpira 4 načine naslavljanj. Ugotovite tip naslavljanja za naslednje primere:

```
ADD A, 7Fh  _____________________________
ADD A, @R0  _____________________________
ADD A, R7   _____________________________
ADD A, #127 _____________________________
```

5. S pomočjo tabele ukazov zbirnega jezika ugotovite katere načine naslavljanj podpirajo ukazi za delo z registri `RN`. Za vsakega napišite primer.

6. Seštevanje nepredznačenih števil

   a. S pomočjo nabora ukazov ugotovite pomen posameznih ukazov v programu ter skušajte ugotoviti delovanje programa.

```
; vaja1_1.a
x       data 30h
y       data 31h
z       data 32h
        mov A, x
        mov R0, y
        add A, R0
        mov z, A
        end
```

Tabela 7: `vaja1_1.a`

[5] Podobno kot v jeziku C. Ime polja je naslov prvega elementa (npr. `polje`), medtem ko z indeksom dostopimo do posameznih elementov polja (npr. `polje[i]`).

b. Identificirajte načine naslavljanj za posamezne ukaze ter ugotovite koliko besed obsega posamezen ukaz v programskem pomnilniku.

c. S pomočjo okna *disassembly* ugotovite strojno kodo za posamezen zbirniški ukaz.

d. Opazujte vsebino registrov in pomnilnika med izvajanjem programa:

   i. Določite koliko besed programskega pomnilnika zaseda program.

   ii. Določite stanje podatkovnega pomnilnika.

e. Rezultat seštevanja dveh n-bitnih števil je v splošnem lahko n+1-bitno število. Dopolnite program tako, da boste upoštevali tudi prenosni bit. Rezultat naj bo sestavljen iz dveh besed.

7. Seštevanje 16 bitnih števil

   a. Predelajte program `vaja1_1.a` tako, da bo primeren za seštevanje dveh 16 bitnih števil. Pri tem upoštevajte, da lahko rezultat preseže 16 bitov!

   b. Grafično predstavite algoritem za seštevanje dveh 16 bitnih nepredznačenih števil!

   c. Program testirajte z različnimi vhodnimi podatki. Sestavite nabor vhodnih podatkov, ki bo učinkovito preveril delovanje vašega programa.

8. Seštevanje predznačenih števil

   a. Nadgradite program `vaja1_1.a` tako, da bo primeren tudi za seštevanje predznačenih števil.

   b. Kako vlogo imata zastavici prenosa in preliva?

   c. Grafično predstavite algoritem za seštevanje dveh 16 bitnih predznačenih števil!

   d. Program testirajte z različnimi vhodnimi podatki. Sestavite nabor vhodnih podatkov, ki bo učinkovito preveril delovanje vašega programa.

9. Mikrokrmilnik 8051 pozna ukaz za množenje dveh števil (`MUL AB`). Rezultat množenja je 16 bitno število. Zgornja beseda rezultata se nahaja v akumulatorju `A`, spodnja beseda pa v registru `B`. V primeru množenja dveh n-bitnih števil lahko v splošnem rezultat obsega $2n$ bitov. V primeru množenja 8-bitnih števil to pomeni, da je rezultat lahko dolg največ 16 bitov.

   a. Napišite program, ki bo množil dve osem bitni nepredznačeni števili z uporabo ukaza `MUL`.

   b. Koliko urinih ciklov je potrebnih za izvedbo rutine? (glej [C51_PG])

10. Množenje z mnogokratnikom števila 2 (2^n) lahko realiziramo tudi s pomikom v levo. Množenju z 2 ustreza pomik besede za en bit v levo. Na najmanj pomemben bit pa dodamo ničlo.

```
1:   0000 0001
     0000 001   ← 0
2:   0000 0010
```

a. Napišite rutino za množenje s konstanto 2 brez uporabe ukaza `MUL`. Uporabite ukaz `RLC` pri tem pa pazite, da je pred uporabo ukaza zastavica `cy` postavljena na vrednost 0.

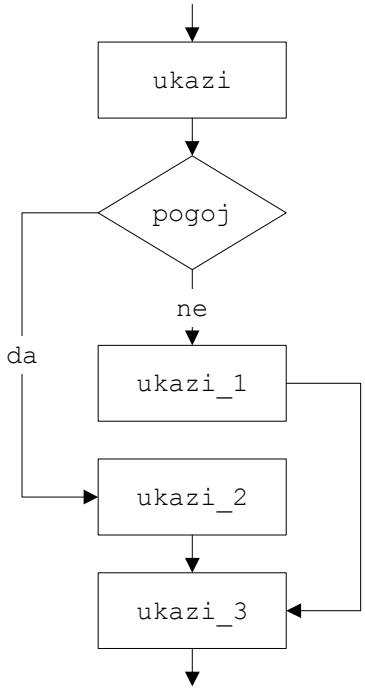
b. Razmislite kako bi na podoben način realizirali množenje s konstanto 6? Narišite algoritem!

c. Primerjajte oba načina množenja ter ugotovite v katerih primerih je pomikanje učinkovitejše od uporabe ukaza za množenje? [C51_PG]

2.3 Skočni in vejitveni ukazi

Skočni in vejitveni ukazi omogočajo programerju odločanje in ponavljanje. Ob resetu se programski števec postavi na `0000h`. Mikrokrmilnik izvršuje ukaze enega za drugim, sekvenčno, kot so shranjeni v programskem pomnilniku. Vrstni red izvajanja ukazov lahko spremenijo skočni in vejitveni ukazi, s katerimi spreminjamo vrednost programskega števca `PC`.

Pomembna razlika med vejitvenimi ukazi in skočnimi ukazi je ta, da se v primeru skočnih ukazov programski tok vedno spremeni. V primeru vejitvenih ukazov pa se programski tok spremeni samo, če je določen pogoj izpolnjen.



Slika 9: Primer vejitve in skoka

Vejitve glede na stanje bitov registra stanja (`PSW`) ali drugih bitov v pomnilniku vplivajo na vrstni red izvajanja ukazov. Tako se lahko izvrši kateri drug ukaz in ne tisti, ki je v pomnilniku naslednji na vrsti. Na tak način lahko zaporedje ukazov, ki se večkrat ponovi, zaobjamemo z zanko. Vejitveni ukazi predstavljajo osnovo, saj z njimi izvršimo vse odločitve. Vloga vejitev je podobna `if....then` zanki, pri višjih programskih jezikih.

Poleg vejitvenih poznamo tudi skočne ukaze. Skočne ukaze potrebujemo za primere, ko moramo nujno skočiti na nek naslov brez predhodnega odločanja. Skočni ukazi ustrezajo `goto` v višjih programskih jezikih. V primerih, ko uporabimo skočne ukaze želimo, da se program brezpogojno nadaljuje od določene pomnilniške lokacije naprej.

Zgornji algoritem (Slika 9) zapišimo v pseudo-zbirnem jeziku

```
                ukazi
                JB 45h, naslov_1
                ukazi_1
                LJMP naslov_2
naslov_1:       ukazi_2
naslov_2:       ukazi_3
```

Tabela 8: Algoritem (Slika 9)

Vejitveni ukaz `JB` (*Jump if Bit set*) se izvrši, če je bit na naslovu `45h` bitnega pomnilnika postavljen (enak 1). Če je bit postavljen se bo izvrševanje programa nadaljevalo od labele `naslov_1` z izvajanjem ukaznega bloka `ukazi_2`, ukazni blok `ukazi_1` pa bo izpuščen. Če bit ni postavljen se izvrševanje programa nadaljuje in ukaz ukazni blok `ukazi_1` se izvrši.

`LJMP`	*long jump*	brezpogojni skok - dolg
`SJMP`	*short jump*	brezpogojni skok - kratek (+/- 128 besed)
`JC`	*jump if carry*	vejitev, če je prenos postavljen
`JNC`	*jump if not carry*	vejitev, če prenos ni postavljen
`CJNE`	*compare and jump if not equal*	primerjaj, če ni enako izvedi vejitev
`DJNZ`	*decrement and jump if not zero*	zmanjšaj, če ni enako nič izvedi vejitev
`JZ`	*jump if zero*	vejitev, če je enako nič
`JNZ`	*jump if not zero*	vejitev, če ni enako nič
`JB`	*jump if bit*	vejitev, če je bit postavljen
`JNB`	*jump if not bit*	vejitev, če bit ni postavljen
`JBC`	*jump if bit and then clear*	vejitev, če je bit postavljen ter pobriši bit

Tabela 9: Skočni in vejitveni ukazi

Pri vejitvah moramo paziti na dejstvo, da se labela lahko nahaja samo ± 128 besed od vejitvenega ukaza. V zgornjem primeru to pomeni, da se mora labela `naslov_1` nahajati ±128 besed od vejitvenega ukaza `JB`.

Brezpogojni skok izvedemo z ukazom `LJMP` (*Long JuMP*). Ko mikrokrmilnik izvede ta ukaz se v programski števec vpiše naslov labele `naslov_2` in programa se nadaljuje z izvajanjem ukaznega bloka `ukazi_3`.

Podobno vlogo ima tudi ukaz `SJMP` (*Short JuMP*). V tem primeru mora biti naslov kamor skočimo največ ± 128 besed od skočnega ukaza. Pomembna razlika je, da `SJMP` zasede samo dve besedi, medtem ko `LJMP` zasede tri besede pomnilnika.

2.3.1 Nalogi

11. Napišite program, ki zapiše vrednost akumulatorja `A` na izhod `P1`, če je `R7` pozitiven in vrednost registra `B`, če je `R7` negativen. Predznak (bit 7) preverite tako, da vsebino registra `R7` shranite v bitni pomnilnik ter testirate ustrezen bit.

12. Napišite program, ki z vhoda `P1` bere zaporedje števil. Izračunajte vsoto najmanj pomembnih bitov (bit 0). Program naj se konča, ko z vhoda prebere število 0.

2.4 Podprogrami

Podprogram je zaporedje ukazov, ki ima svojo labelo in se zaključi z ukazom `RET`. Podprogram pokličemo z ukazom `LCALL`, ki mu sledi labela. Ko se podprogram izvrši, se vrnemo v glavni program na mesto klica podprograma z ukazom `RET`. Podobno kot skočni in vejitveni ukazi, tudi klici in vrnitve iz podprograma, spreminjajo vrednost programskega števca `PC`.

```
                ukazi
                LCALL podprogram
                ukazi_1
podprogram:     ukazi_2
                RET
```

Tabela 10: Klic in vrnitev iz podprograma

Oglejmo si dogajanje ob klicu podprograma in vrnitvi iz podprograma (Tabela 10). Ob klicu podprograma `LCALL` se izvrši naslednje:

- `PC` se poveča za 3 in samodejno shrani na sklad. Na sklad se najprej shrani spodnja beseda ter nato še zgornja beseda naslova.
- v `PC` se naloži naslov začetka podprograma (naslov labele `podprogram`)
- podprogram se izvede (`ukazi_2`)
- ob vrnitvi iz podprograma (`RET`) se v `PC` zapiše vrednost, ki je shranjena na skladu. `PC` kaže na `ukazi_1`.

Tako se program nadaljuje z ukaznim blokom `ukazi_1`, ki sledi ukazu `LCALL`.

Če se med izvajanjem podprograma spremeni vrednost akumulatorja, registra `PSW` ali katerega drugega registra, katerega vrednost potrebujemo po vrnitvi iz podprograma, moramo njihove vrednosti ob začetku podprograma shraniti na sklad.

Shranjevanje vsebin registrov na sklad izvedemo z ukazom `PUSH`. Ob koncu podprograma vsebino sklada zapišemo v ustrezne registre v obratnem vrstnem redu z ukazom `POP` (Tabela 11).

```
podprogram: PUSH ACC
            PUSH PSW
            ukazi_2
            POP PSW
            POP ACC
            RET
```

Tabela 11: Pisanje na sklad in branje s sklada

2.4.1 Nalogi

13. V glavnem programu pokličite podprogram, ki bo zmnožil vrednosti registrov `R0` in `R1`. Spodnja beseda rezultata naj bo shranjena v `R0`, zgornja pa v `R1`. Na sklad shranite tiste registre, katerim vsebina se v podprogramu spremeni. To so akumulator `A`, register `B` ter register `PSW`.

14. Zgornji podprogram izvajajte po korakih in opazujte vrednost programskega števca `PC`.

3 Programiranje v jeziku C

V tem delu najdemo nekatere dodatke k ANSI C standardu, ki jih uporablja prevajalnik Cx51. Ti dodatki ali razširitve se nanašajo predvsem na arhitekturne značilnosti mikrokrmilnika, v našem primeru arhitekture 8051.

Cx51 prinaša naslednje pomembnejše dodatke:

- razlikuje različne tipe pomnilnikov in pomnilniške lokacije,
- določa rezervirane besede za določitev tipa pomnilnika,
- določa rezervirane besede za določitev podatkovnega tipa spremenljivk,
- deklaracija bitnih spremenljivk in naslavljanje bitnih spremenljivk,
- delo z registri SFR,
- pomnilniško določene kazalce,
- posebnosti pri deklaraciji funkcij.

Poglavje na kratko prikazuje pomembnejše posebnosti Cx51. Cx51 je natančno opisan v [C51].

3.1 Tipi pomnilnikov

Organizacija pomnilnika je obdelana v podpoglavju 1.1. Zaradi preglednosti so bistveni podatki zbrani v spodnji tabeli (Tabela 12).

`code`	programski pomnilnik (*program memory*)
`data`	notranji podatkovni pomnilnik - neposreden dostop (*directly addressable internal data memory*) (spodnjih 128 besed),
`idata`	celoten notranji pomnilnik - posredni dostop (in*directly addressable internal data memory*) (256 besed),
`bdata`	del notranjega pomnilnika (*bit-addressable internal data memory*), ki ga naslovimo po posameznih bitih ali besedah (16 besed)
`xdata`	zunanji podatkovni pomnilnik (*external data memory*) (64k besed)
`pdata`	(*paged external data memory*) (256 besed)

Tabela 12: Rezervirane besede v Cx51 za deklaracijo pomnilnika [C51]

3.2 Podatkovni tipi

Cx51 podpira standardne podatkovne tipe, pozna pa tudi podatkovne tipe, ki so specifični za mikrokrmilnik 8051.

V primeru, ko imamo opravka z 16 bitnim procesorjem je smiselno uporabljati podatkovni tip `int`, saj je arhitektura narejena za delo s 16 bitnimi podatki. V primeru 8051 predstavlja osnovni podatkovni tip `char`, saj je arhitektura procesorja 8 bitna. Vsesplošna uporaba 16 bitnih spremenljivk je tako v tem primeru potratna s prostorom, izvajanje pa je počasno.

Previdni moramo biti pri uporabi predznačenih števil, saj 8051 ne pozna ukazov za predznačeno aritmetiko. Izvajanje je v tem primeru odvisno od počasnih knjižničnih funkcij.

tip	biti	besede	interval
`bit`*	1		0 , 1
`signed char`	8	1	-128 , +127
`unsigned char`	8	1	0 , 256
`enum`	8 / 16	1 / 2	-128 , +127 / -32.768 , +32.767
`signed short`	16	2	-32.768 , +32.767
`unsigned short`	16	2	0 , 65.535
`signed int`	16	2	-32.768 , +32.767
`unsigned int`	16	2	0 , 65.535
`signed long`	32	4	-2.147.483.648 , +2.147.483.647
`unsigned long`	32	4	0 , 4.294.967.295
`float`	32	4	±1.175494E-38 , ±3.402823E+38
`sbit`*	1		0 , 1
`sfr`*	8	1	0 , 255
`sfr16`*	16	2	0 , 65.535

Tabela 13: Seznam podatkovnih tipov Cx51. Podatkovni tipi specifični za Cx51 so označeni z *.

3.2.1 Tip bit

Spremenljivke tipa `bit` so shranjene v bitnem delu notranjega podatkovnega pomnilnika. Ta segment je dolg le 16 besed, tako da je na razpolago le 128 bitnih lokacij.

Podatkovni tip `bit` uporabimo za deklaracijo enobitnih spremenljivk:

```
bit zastavica;          /*zastavica je bitna spremenljivka*/
zastavica = 1;          /*ki ji priredimo vrednost 1*/
```

Podatkovni tip `bit` lahko uporabimo za deklaracijo spremenljivk, pri funkcijah pa kot argument in vrnjeno vrednost.

Nekaj primerov napačne uporabe tipa `bit`:

```
bit ptr;              /*kazalec ne more biti tipa bit*/
bit polje[5];         /*polje ne more biti tipa bit*/
```

Pri eksplicitni deklaraciji spremenljivk lahko pri tipu `bit` uporabimo samo rezervirani besedi `data` in `idata`.

3.2.2 Tip sfr

Registri SFR predstavljajo rezervirani del notranjega podatkovnega pomnilnika (`0x80` - `0xFF`). Registri SFR določajo delovanje časovnikov (*timers*), števcev (*counters*), serijskih vhodov in izhodov, vzporednih vhodov in izhodov, ter ostalih perifernih enot.

Tip `sfr` uporabljamo kot druge podatkovne tipe.

```
sfr P0 = 0x80;        /* vrata P0 so na naslovu 0x80 */
```

Nekatere izpeljanke 8051 poznajo tudi tip `sfr16`, ki ga uporabljamo za deklaracijo 16 bitnih spremenljivk v registrih SFR.

3.2.3 Tip sbit

V bitnem delu pomnilnika deklariramo spremenljivke, ki jih lahko naslovimo po besedah ali po posameznih bitih. To so spremenljivke, ki so na primer tipa `int` ali `char`, kjer lahko s posebnimi ukazi spreminjamo vrednosti posameznim bitom.

```
char bdata beseda = 0x20; /* deklaracija v bitnem pomnilniku */
sbit bit0 = beseda^0;      /* poimenovanje 1. bita
                              spremenljivke beseda*/
```

V zgornjem primeru je v bitnem pomnilniku (`bdata`) (3.3.1) deklarirana spremenljivka `beseda`, ki je tipa `char`, kar pomeni, da zaseda 8 bitov. V drugi vrstici najmanj pomembnemu bitu spremenljivke `beseda` priredimo novo ime `bit0`. Znak `^` določa posamezen bit spremenljivke.

Tudi večino registrov SFR lahko naslovimo tudi po posameznih bitih. Posameznim bitom registra SFR so prirejene bitne spremenljivke.

```
sbit CY = PSW^7;      /* 8. bit PSW priredimo zastavici CY */
```

Deklaracije SFR registrov so zbrane v knjižnici, ki jo v program vključimo podobno kot druge knjižnice (5.1.1)

```
#include <89c51rd2.h>
```

3.3 Deklaracija spremenljivk in konstant

3.3.1 Eksplicitno deklarirane spremenljivke

Pri Cx51 lahko pri deklaraciji spremenljivk tudi eksplicitno določimo del pomnilnika, kjer bo spremenljivka shranjena. Pri tem uporabimo poleg podatkovnega tipa (Tabela 13) eno izmed rezerviranih besed (Tabela 12), s katero izberemo tudi pomnilnik.

```
char data var1;
char code tekst[] = "Vnesi parameter:";
unsigned long xdata polje[100];
float idata x, y, z;
unsigned int pdata dolzina;
unsigned char xdata matrika[3][4][5];
char bdata zastavica;
```

3.3.2 Deklaracija konstant

Konstante so zapisane v programskem pomnilniku (`code`) ali pa so shranjene v zunanjem pomnilniku (`xdata`) in inicializirane ob zagonu. V primeru mikrokrmilnikov je prvi način v splošnem primernejši je pa vsekakor hitrejši. Tako je na primer izvedba ukaza `MOVC A, @DPTR` mnogo hitrejša kot `MOVX A, @DPTR`.

Primer deklaracije konstant v programskem pomnilniku (`code`):

```
code unsigned char x = 0x02;
code unsigned int temp = 5;
code unsigned char tabela[]={'1','2','3','4','5'};
```

Pri tem moramo opozoriti, da `code` ni enako `const`! Če deklariramo spremenljivke kot konstante s `const` bodo le te shranjene v podatkovem delu pomnilnika (RAM).

Na splošno si zapomnimo, da vse konstante, še posebej daljše tabele (*lookup tables*) shranimo v programski del pomnilnika. Pri deklaraciji spremenljivk moramo namreč pozorni, da ne zapolnimo notranjega podatkovnega pomnilnika (128 besed).

Konstante zapisane v podatkovnem delu pomnilnika deklariramo:

```
unsigned char x = 128;
unsigned int konstanta = 0xFD34;
```

Njihove vrednosti lahko med izvajanjem programa spreminjamo.

3.3.3 Implicitno deklarirane spremenljivke

Pri deklaraciji spremenljivk lahko tip pomnilnika tudi izpustimo. V tem primeru se avtomatsko izbere implicitni pomnilniški tip (*default*).

Implicitni pomnilniški tipi so vnaprej določeni z izbranim pomnilniškim modelom. Prevajalnik pozna tri pomnilniške modele: `SMALL`, `COMPACT` in `LARGE` [C51].

V splošnem se vse spremenljivke nahajajo v podatkovnih pomnilnikih (RAM). Izbrani pomnilniški model določa ali je to notranji podatkovni pomnilnik ali zunanji podatkovni pomnilnik.

Tako bodo v primeru `#pragma SMALL` vse spremenljivke shranjene v notranjem podatkovnem pomnilniku. V kolikor bi neko spremenljivko želeli zapisati v zunanji podatkovni pomnilnik (RAM) bi uporabili rezervirano besedo `xdata`.

```
#pragma SMALL
unsigned char x;
xdata unsigned char y;
xdata char veliko_polje[200];
```

Spremenljivka `x` je deklarirana v notranjem podatkovnem pomnilniku. Spremenljivki `y` in `veliko_polje` pa sta deklarirani v zunanjem podatkovnem pomnilniku.

3.4 Kazalci

Kazalce deklariramo tako, kot v standardnem jeziku C.

```
char *s;
int *kaz;
```

V prvem primeru bo kazalec `s` kazal na podatek tipa `char`, v drugem pa kazalec `kaz` na podatek tipa `int`.

Deklaracija kazalca zaseda tri besede v pomnilniku. Prva beseda hrani informacijo o pomnilniku kamor kaže, preostala dva pa odmik (relativni naslov). Kazalci lahko kažejo na katerokoli spremenljivko v kateremkoli izmed pomnilnikov 8051. Tako deklarirane kazalce imenujemo tudi generični kazalci (*generic pointers*).

Cx51 pozna še drugo vrsto kazalcev. Imenujemo jih tudi pomnilniško določeni kazalci (*memory-specific pointers*). Ob deklaraciji kazalca je določen tudi pomnilnik kamor kazalec kaže.

```
char data *niz;
int xdata *num;
```

V prvem primeru bo kazalec `niz` kazal na podatek tipa `char` v notranjem podatkovnem pomnilniku, ki ga določa rezervirana beseda `data`. V drugem primeru pa bo kazalec `num` kazal na podatek tipa `int` v zunanjem podatkovnem pomnilniku določenem z `xdata`.

Tako deklarirani kazalec zaseda samo dve besedi. Ker je pomnilnik, kamor bo kazalec kazal, določen že ob prevajanju ne potrebujemo prve besede, kot v primeru generičnih kazalcev.

3.5 Funkcije

Funkcije predstavljajo večje programske gradnike, ki izvajajo določene dejavnosti programa [Bratkovič98]. Funkcije sestavljajo stavki. Funkcije lahko razdelimo v tri skupine:

- funkcija `main`
- knjižnične funkcije
- lastne funkcije

V nadaljevanju si bomo ogledali dve izmed posebnosti pri delu z lastnimi funkcijami v Cx51:

- funkcija, kot prekinitvena rutina
- uporaba skupine registrov

3.5.1 Nabor registrov

Ena izmed posebnosti je, da posamezni funkciji lahko določimo skupino registrov s katerim razpolaga (1.3).

```
void funkcija(void) using 3
 {
       stavki
 }
```

Tabela 14: Izbira nabora registrov

Z rezervirano besedo `using` določimo katero izmed štirih skupin registrov bo funkcija uporabljala[6]. V zgornjem primeru funkcija uporabi četrto skupino splošno namenskih registrov `R0 – R7`, ki so na naslovih `18h – 1Fh` (Slika 5) [C51]. Ko se izvajanje funkcije zaključi postane aktiven prejšnji nabor registrov.

3.5.2 Prekinitve

Pri mikrokrmilniku 8051 imamo na voljo več prekinitev:

- dve zunanji prekinitvi (`INT0` in `INT1`)
- prekinitvi, ki ju pokliče stanje časovnikov (*Timer 0* in *Timer 1*)
- prekinitev serijskih vrat ob sprejemu in oddaji znaka

Vsaka prekinitev ima določeno lokacijo prekinitvene rutine. Ob prekinitvi CPE začne izvajati prekinitveno rutino.

EA	–	–	ES	ET1	EX1	ET0	EX0

Tabela 15: Register IE

[6] Pri deklaraciji funkcije rezervirano besedo `using` izpustimo.

Posamezna prekinitev je omogočena če postavimo ustrezen bit v registru `IE` (*interrupt enable*). Posebno vlogo ima bit `EA` (*global interrupt enable/disable*). Če je bit `EA = 0` so prekinitve onemogočene. Če je bit `EA = 1` ter je postavljen še kateri izmed bitov registra `IE`, je izbrana prekinitev omogočena.

prekinitev	opis
0	zunanja prekinitev `INT0`
1	časovnik *Timer 0*
2	zunanja prekinitev `INT1`
3	časovnik *Timer 1*
4	serijska vrata (*Serial port*)

Tabela 16: Prekinitve 8051 [C51_PG]

Ko programiramo v jeziku C nam prevajalnik omogoča, da zapišemo prekinitveno rutino (ISR - *interrupt service routine*) v obliki funkcije. Ob nastopu izbrane prekinitve se ta prekinitvena rutina samodejno izvede tako, da prevajalnik samodejno generira prekinitveni vektor. Prekinitvena rutina ima podobno sintakso kot funkcija, le da programer določi katera prekinitev jo bo klicala. Torej te funkcije nimajo eksplicitnega klica funkcije, ravno tako ne sprejemajo in vračajo argumentov.

```
unsigned char stevec_prekinitev = 0;
void  zunanja_prekinitev() interrupt 0
{
  stevec_prekinitev++;
}
void main ()
{
inicializacija_casovnika0 ()    /* poglavje 5 */
IT0 = 1;         /* TCON: zadnja fronta na INT0*/
EX0 = 1;             /* omogoči prekinitev EX0 */
EA = 1;                  /* omogoči prekinitve */
while (1)
  {
  }
}
```

Tabela 17: Primer funkcije, ki jo kliče zunanja prekinitev

Naslednji primer prikazuje uporabo zunanjega signala (Tabela 16) za klic prekinitve (Tabela 17). Bit `IT0 = 1` (`TCON.0`) določa, da funkcijo `zunanja_prekinitev()` kliče zadnja

fronta signala na vhodu INT0 (P3.2) (Tabela 21). Postavljen bit EX0 omogoči zunanje prekinitve, postavljen bit EA pa globalno omogoči prekinitve (Tabela 15).

V spodnjem primeru želimo, da se rutina izvede ob prekinitvi, ki jo povzroči časovnik *Timer 0* (Tabela 16).

```
void prekinitev_casovnik0() interrupt 1
{
 koda prekinitvene rutine
}
```

Tabela 18: Sintaksa prekinitvene rutine

Tako rutino uporabimo, kadar želimo, da se del kode izvede samo, ko se resetira časovnik *Timer 0*.

3.6 Postavljanje in brisanje bitov

Logične operatorje uporabljamo tudi za postavljanje in brisanje bitov v besedi. Videli smo, da lahko v zbirnem jeziku 8051 v bitnem delu pomnilnika in nekaterih SFR registrih zelo enostavno postavimo ali zbrišemo posamezen bit (3.2). V nadaljevanju si bomo ogledali kako v jeziku C spreminjamo posamezne bite v besedi in pri tem ne spremenimo vrednosti ostalih bitov v besedi. Za tovrstne operacije uporabimo logične operatorje.

Na primer, da želimo postaviti četrti bit v registru P0 (naslov 0x80) in pri tem želimo, da ostanejo ostali biti nespremenjeni. V zbirnem jeziku to lahko naredimo z masko, ki ima enice na mestih, kjer želimo enice

```
ORL 80h, #10h
```

Lahko pa postavimo samo posamezen bit z ukazi za delo z biti

```
SETB P0.4
```

Tudi pri programiranju v jeziku C lahko nastavljamo vrednost posameznih bitov z uporabo mask. V zgornjem primeru bi uporabili operator ALI in masko, ki ima enice na mestih, kjer želimo postaviti enice

```
P0 = P0 | 0x10;
```

Kot smo že videli, Cx51 pa omogoča, da spreminjamo posamezne bite z bitnimi ukazi tudi v jeziku C. Določeni registri SFR in bitni del pomnilnika so bitno naslovljivi, zato se lahko izognemo uporabi mask.

Pri 8051 lahko spremenljivko shranimo v bitni pomnilnik (začne se na naslovu 20h). V bitnem pomnilniku lahko rezerviramo tudi večje spremenljivke, ki jih lahko naslovimo po besedah ali po posameznih bitih. Prevajalnik shrani spremenljivke, ki imajo pri deklaraciji dodan bdata v bitni pomnilnik.

```
int bdata iX;
char bdata polje[4];
```

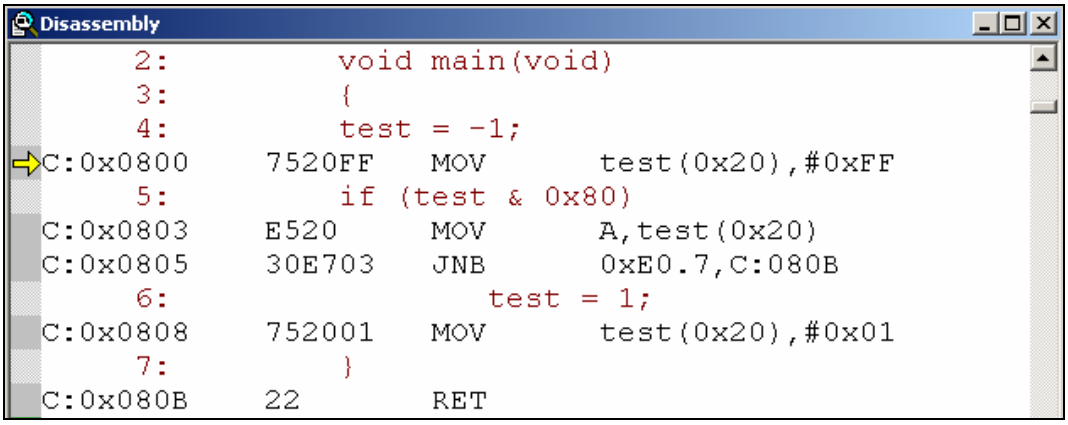
Spremenljivki `iX` in `polje[4]` lahko naslovimo tudi po posameznih bitih. Tako lahko spreminjamo posamezne bite teh dveh spremenljivk. Za dostop do posameznih bitov uporabimo `sbit`.

```
sbit bit0 = iX^0;           /* bit 0  */
sbit bit15 = iX^15;         /* bit 15 */
sbit znak03 = polje[0]^3;   /* bit 3 znaka polje[0] */
sbit znak37 = polje[3]^7;   /* bit 7 znaka polje[3] */
```

Tabela 19 podaja primer uporabe maske. V bitnem delu pomnilnika je deklarirana znakovna spremenljivka `test`. V kolikor je najpomembnejši bit spremenljivke `test` postavljen spremenljivki test priredimo vrednost 1.

```
bdata char test;
void main(void)
{
test = -1;
if (test & 0x80)
      test = 1;
}
```

Tabela 19 Primer uporabe maske



Slika 10 Sled programa (Tabela 19)

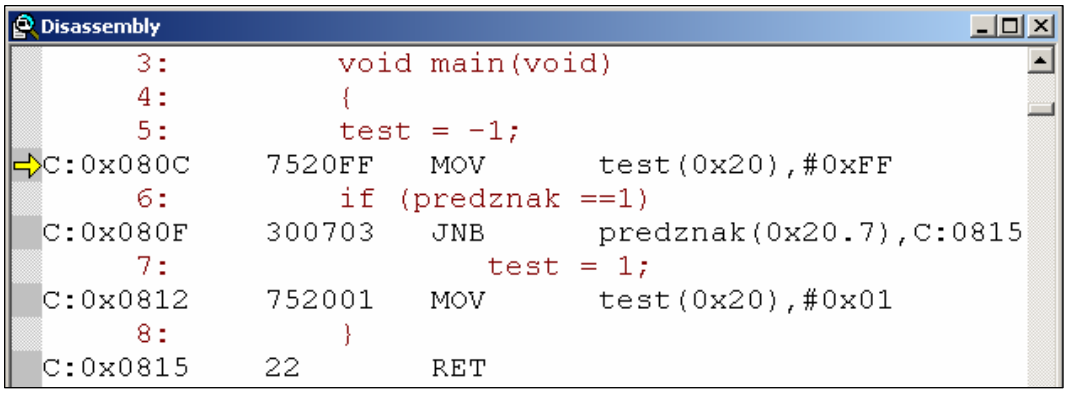
Naslov `0xE0` je naslov akumulatorja (`ACC`), `0xE0.7` pa je njegov najpomembnejši bit.

Tabela 20 podaja primer uporabe bitnega pomnilnika. V bitnem delu pomnilnika je deklarirana znakovna spremenljivka `test`, bitna spremenljivka `predznak` pa ustreza najpomembnejšemu bitu spremenljivke `test`. V kolikor je najpomembnejši bit spremenljivke test postavljen spremenljivki priredimo vrednost 1.

V primeru ko uporabimo `sbit` se celota primerjava izvrši kot en sam ukaz `JNB`, kar je hitreje, kot v primeru ko uporabimo masko.

```
bdata char test;
sbit predznak = test^7;
void main(void)
{
      test = -1;
      if (predznak ==1)
      test = 1;
}
```

Tabela 20 Primer uporabe bitnega pomnilnika



Slika 11: Sled programa (Tabela 20)

3.7 Naloge

15. V registru P0 želimo pobrisati bit 0 in bit 5 in pri tem ohraniti vrednost ostalih bitov. Napišite ukaze v zbirnem jeziku z uporabo maske in uporabo ukazov za delo z bitnim pomnilnikom ter v jeziku C z uporabo maske.

16. Napišite ukaze s katerimi spremenljivki `char test`:

    a. postavite spodnje štiri bite na 0, vrednost zgornjih štirih bitov pa se ohrani,

    b. postavite spodnje štiri bite na 1, vrednost zgornjih štirih bitov pa se ohrani.

17. Napišite rutino, ki sešteje vsa:

    a. liha števila v polju. Števila so 8 bitna nepredznačena števila. Polje se prične na naslovu `0x20` ter konča na naslovu `0x2F` podatkovnega pomnilnika.

    b. negativna števila v polju. Števila so 8 bitna predznačena števila. Polje se prične na naslovu `0x20` ter konča na naslovu `0x2F` podatkovnega pomnilnika.

18. Napišite rutino, ki na izhod P1 izpisuje:

a. binarna števila po vrsti, kot binarni števec. Izpisu `1111 1111` naj zopet sledi `0000 0000`. Med posameznimi izpisi naj bo kratka zakasnitev.

b. naslednji vzorec:

```
1000 0000
1100 0000
1110 0000
...
1111 1111
1111 1110
1111 1100
...
```

4 Časovniki in števci

Mikrokrmilnik 8051 ima dva neodvisna časovnika/števca, ki imata tri osnovne funkcije:

- merjenje časa in merjenje trajanja dogodkov (časovnik),
- štetje dogodkov (števec),
- nastavitev hitrosti prenosa serijskih vrat (poglavje 5).

Časovnika/števca imata oznaki *Timer 0* in *Timer 1* oziroma *Counter 0* in *Counter 1*. Časovnik ali števec je isti del strojne opreme. Različna termina uporabljamo glede na uporabo. Če merimo čas uporabimo termin časovnik, če pa štejemo uporabimo termin števec.

Časovnika/števca lahko delujeta v različnih režimih glede na nastavitve dveh SFR registrov `TCON` in `TMOD`. Vsebini registrov moramo nastaviti preden uporabimo časovnika/števca.

`P3.0`	serijski vhod `RDX` (*serial input port*)
`P3.1`	serijski izhod `TDX` (*serial output port*)
`P3.2`	zunanja prekinitev `INT0` (*external interrupt 0*)
`P3.3`	zunanja prekinitev `INT1` (*external interrupt 1*)
`P3.4`	zunanji vhod števca *Counter 0*
`P3.5`	zunanji vhod števca *Counter 1*
`P3.6`	`WR` (*external data memory write strobe*)
`P3.7`	`RD` (*external data memory read strobe*)

Tabela 21 Vrata `P3` [C51_HD]

Register `TMOD` (*Timer mode*) določa način dela obeh časovnikov. Zgornji štirje biti določajo delovanje časovnika *Timer 1*, spodnji pa časovnika *Timer 0*. Z nastavitvijo bita `C/Tx` izbiramo med časovnikom in števcem. Postavljen bit `GATEx` določa, da zunanji signal proži časovnika. Bita `TxM0` in `TxM1` določata način delovanja časovnika oziroma števca. Oznaka `x` je lahko 0 ali 1, določa pa enega od obeh časovnikov.

GATE1	C/T1	T1M1	T1M0	GATE0	C/T0	T0M1	T0M0

Tabela 22: Register `TMOD`

Register `TCON` (*Timer control*) določa delovanje (`TR0` in `TR1`) ter vsebuje informacijo o stanju časovnikov (`TF0` in `TF1`). Če sta bita `TR0` in `TR1` postavljena časovnika štejeta. Ko se časovnika resetirata se postavita bita `TF0` in `TF1`.

TF1	TR1	TF0	TR0				

Tabela 23: Register TCON

Vhodno/izhodna vrata P3 (Slika 2) so namenjena tudi za posebne naloge. Tabela 21 opisuje vlogo posameznih bitov vrat P3.

4.1 Merjenje časa z uporabo prekinitev

Osnovna naloga časovnikov je, da merijo čas. Časovnik se poveča za ena vsak strojni cikel, za razliko od drugih ukazov, ki potrebujejo od enega do štiri strojne cikle za izvršitev. Časovnik je torej števec strojnih ciklov. Pri mikrokrmilniku 8051 traja strojni cikel 12 period oscilatorja.

Pri frekvenci oscilatorja 16,384 MHz se števec v eni sekundi poveča za 1.365.333 krat, saj strojni cikel sestavlja 12 period oscilatorja.

16.384.000 / 12 = 1.365.333,333

Torej je ločljivost časovnika, oziroma najkrajši čas, ki ga lahko merimo pri dani frekvenci oscilatorja, 0,7324 μs.

Oglejmo si kolikokrat se števec poveča v izbranem intervalu, npr. v 1/100 sekunde? V 0,01 sekunde se bo torej števec povečal 13.653 krat. Zaradi celoštevilskega zapisa naredimo majhno napako. Čas, ki je pretekel je enak 0,0099997 s. Napaka je enaka 0,000000244 sekunde kar je zadovoljivo za večino aplikacij.

Poglejmo si kako nastavimo časovnika *Timer 0* in *Timer 1*, da bosta merila čas. Vsak od njiju ima par SFR registrov TH0/TL0 ter TH1/TL1 (Slika 6), kjer je zapisano stanje časovnika. Poleg tega si časovnika delita dva SFR registra TMOD in TCON, kjer so zapisane nastavitve delovanja obeh časovnikov (Slika 6).

Vrednost časovnika ali stanje je tako zapisana z dvema besedama ali 16 biti. Register z oznako THx hrani zgornjih osem bitov, register TLx pa spodnjih osem bitov. Največja vrednost, ki jo lahko hrani časovnik je 65.535. Če časovnik povečamo za ena se bo resetiral in v njem bo zapisana vrednost nič.

V večini primerov ni toliko pomembna trenutna vrednost časovnika, kot to, da vemo, kdaj se je časovnik resetiral. Ta princip lahko uporabimo za izvedbo želene zakasnitve.

Ko se vrednost časovnika spremeni iz največje vrednosti v nič mikrokrmilnik samodejno nastavi bit TFx. Prva možnost je, da programsko testiramo bit TFx. Druga možnost, pa je uporaba prekinitev (3.5).

Oglejmo kako inicializiramo časovnik *Timer 0*, kot 16 bitni časovnik (način 1). Ker je postavljen bit TR0 časovnik šteje. Ko se časovnik resetira (po vrednosti FFFFh) šteje zopet od začetka (0000h).

X	X	X	X	0	0	0	1

Tabela 24: Nastavitve registra `TMOD`

```
TMOD = (TMOD & 0xF0) | 0x01; /* Timer0: 1.način */
TR0 = 1;                     /* Timer 0 šteje */
```

Tabela 25: Časovnik *Timer 0* v načinu 1

Kot drug primer si oglejmo nastavitve za *Timer 1*, ki naj deluje kot 8 bitni časovnik v *auto-reload* načinu (način 2). Ker je postavljen bit `TR1` časovnik šteje do vrednosti 255. V naslednji periodi se časovnik resetira vendar se register `TL1` ne postavi na vrednost 0 temveč se vanj vpiše vrednost, ki je v naprej določena v registru `TH1`. Časovnik ponovno šteje od te vrednosti vnaprej. Ta način delovanja je natančneje podan v poglavju 5.1.

0	0	1	0	X	X	X	X

Tabela 26: Nastavitve registra `TMOD`

```
TMOD = (TMOD & 0x0F)|0x20;  /* Timer1: način 2 */
TH1 = 256 - 100;            /* TL1 šteje 100 period */
TL1 = TH1;
TR1 = 1;                    /* Timer 1 šteje */
```

Tabela 27: Časovnik *Timer1* v načinu 2

4.1.1 Primer

Kot primer si oglejmo program, ki uporabi časovnik *Timer 0*. *Timer 0* deluje kot 16 bitni časovnik. Vsakokrat, ko se časovnik resetira (prehod iz `0xFFFF` v `0x0000`), prekinitev (*interrupt*) pokliče funkcijo `prekinitev_casovnik0()` ter poveča vrednost spremenljivki `stevec_prekinitev`.

Ob inicializaciji časovnika moramo omogočiti ustrezne prekinitve. Postavitev bita `ET0` omogoči prekinitve, ki jih generira časovnik *Timer 0*. Bit `EA` pa je bit, ki globalno omogoči izvajanje prekinitev.

```
static unsigned long stevec_prekinitev = 0;
void prekinitev_casovnik0() interrupt 1
  {
      stevec_prekinitev++;
  }
```

```
void main()
  {
       TMOD = (TMOD & 0xF0) | 0x01;
       TR0 = 1;
       ET0 = 1;
       EA = 1;

       while(1)
       {
       }
}
```

Tabela 28: Primer uporabe 16 bitnega časovnika

4.2 Merjenje trajanja dogodkov

Mnogokrat moramo določiti trajanje dogodka. V tem primeru bomo uporabili časovnik za merjenje trajanja dogodka na vhodu mikrokrmilnika. Ko bo vhodni signal prisoten bo časovnik meril čas, sicer pa ne.

V ta namen uporabimo bit `GATEx` (Tabela 22). Na primer, če je bit `GATE0` postavljen, delovanje časovnika določa stanje na zunanjem vhodu `INT0`. `INT0` je tudi drugo ime za drugi pin tretjih vzporednih vrat (`P3.2`) (Tabela 21). V kolikor je postavljen bit `GATE0` bo časovnik *Timer 0* štel, kadar bo vhod `INT0` enak ena. V primeru časovnika *Timer 1* sta ustrezna bita `GATE1` in vhod `INT1` (`P3.3`).

X	X	X	X	1	0	0	1

Tabela 29: Nastavitve registra `TMOD`

```
TMOD = (TMOD & 0xF0) | 0x09;
ET0 = 1;
EA = 1;
```

Tabela 30: Časovnik *Timer 0* meri trajanje zunanjega dogodka

4.3 Štetje dogodkov

Števec lahko uporabimo tudi kot števec dogodkov, kjer se vrednost števca poveča samo ob spremembah na zunanjem vhodu. Dogodek na vhodu imenujemo prehod iz visokega stanja v nizko stanje, to je iz 1 v 0. Zunanji vhod števca *Counter 0* je pin `P3.4` za števec *Counter 1* pa pin `P3.5` (Tabela 21).

4.3.1 Primer

Števec *Counter 1* se poveča za ena ob vsakem dogodku na vhodu `P3.5` (Tabela 21). Tabela 31 prikazuje nastavitve registra `TMOD`.

0	1	0	1	X	X	X	X

Tabela 31: Nastavitve registra TMOD

```
TMOD = (TMOD & 0x0F) | 0x50;
TR1 = 1;
```

Tabela 32: Števec *Counter 1* šteje zunanje dogodke

4.4 Naloge

19. Določite razpon in resolucijo 16 bitnega časovnika, če je frekvenca oscilatorja *fosc*=16,384 MHz. Realizirajte 1ms dolgo zakasnitev.

20. Koliko časa preteče med dvema zaporednima resetoma časovnika *Timer 0* v primeru 4.1.1, če je *fosc*=16,384 MHz in je strojni cikel dolg 12 period oscilatorja?

21. Sestavite program, ki inicializira *Timer 1* kot 8 bitni časovnik.

    a. Nastavite vrednosti naslednjim SFR registrom: `TMOD`, `TH1`, `TL1`, `ET1`, `TR1`, `EA`.

    b. Vsakokrat, ko se časovnik resetira (prehod iz `0xFF` v `0x00`), naj prekinitev pokliče funkcijo `prekinitev_casovnik1()` ter poveča vrednost spremenljivki `stevec_prekinitev`.

    c. Koliko časa preteče med dvema prekinitvama, če je `TH1`=(256-100) in *fosc*=16,384 MHz?

22. Napišite rutino, ki meri trajanje dogodkov na vhodu `INT0`.

    a. Uporabite časovnik *Timer 0*. Deluje naj kot 16 bitnem načinu. Vsakokrat ko se časovnik resetira naj poveča vrednost števca `stevec_prekinitev`.

    b. Določite način izračuna trajanja zunanjega dogodka na podlagi vrednosti časovnika ter vrednosti števca `stevec_prekinitev`.

23. Napišite rutino, ki bo štela dogodke na vhodu `P3.4`. Vsakič, ko gre vhod iz visokega v nizko stanje se vrednost števca poveča za 1. Za števec uporabite *Counter 0*. Deluje naj kot 16 bitni števec.

5 Serijska komunikacija

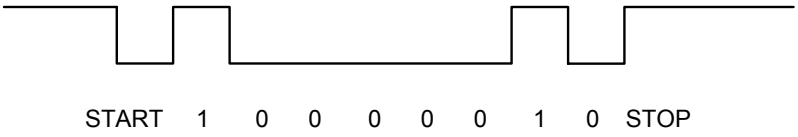
RS-232 je standard za serijsko komunikacijo naprav. Podatki se prenašajo serijsko med dvema uporabnikoma. Parametri ki določajo RS-232 komunikacijo so:

- hitrost prenosa (*Baud rate*): določa koliko informacije se prenese v danem časovnem intervalu. Izražena je v bitih na sekundo. Tipične vrednosti se nahajajo v razponu med 110 – 76800 bit/s, kjer na primer 9600 pomeni hitrost prenosa 9600 bit/s.
- dolžina besede (*Data Width*): ustreza dolžini znaka, ki ga prenašamo. Beseda je lahko dolga 7 ali 8 bitov, odvisno od izbranega formata. V primeru, da prenašamo ASCII znake je dolžina besede enaka 8 bitov.
- pariteta (*Parity*): se uporablja za preverjanje pravilnosti prenesenega znaka.

Podatki se prenašajo serijsko. Besedo (okvir) sestavljajo:

- 1 start bit
- ali 8 podatkovnih bitov
- paritetni bit
- 1 stop bit

Mnogokrat tako zasledimo 10 bitni okvir, ki ga sestavlja 1 start bit, 8 podatkovnih bitov, ter 1 stop bit. Zapis znaka 'A' v ASCII formatu `01000001b` je na sliki (Slika 12). Prenos podatkovnih bitov se začne z najmanj pomembnim bitom. Pri 8051 za serijski prenos skrbi enota UART (Slika 1).



Slika 12: Prenos znaka ‘A’.

Za oddajni (`TxD`) in sprejemni signal (`RxD`) sta rezervirana vhoda `P3.1` in `P3.0` (Tabela 21). V nadaljevanju si oglejmo kako nastavimo delovanje serijskih vrat ter vzpostavimo komunikacijo z drugo napravo.

5.1 Inicializacija serijskih vrat

Da lahko serijska vrata uporabimo za prenos podatkov, jih moramo inicializirati. Z drugimi besedami, nastaviti moramo določene SFR registre, ki določajo delovanje serijskih vrat.

Inicializacijo si oglejmo na primeru (Tabela 33). Funkcija `serial_init()` inicializira serijska vrata na 9600 bit/s pri frekvenci oscilatorja 16,384 MHz.

```
void serial_init()
{
        SCON = 0x50;
        TMOD = 0x20;
        PCON = 0x80;
        TH1 = 0xF7;
        TR1 = 1;
        TI = 1;
}
```

Tabela 33: Funkcija za inicializacijo serijskih vrat

Da bi razumeli nastavitve si oglejmo opise SFR registrov [C51_HD].

SM0	SM1	SM2	REN	TN8	RB8	TI	RI

Tabela 34: Register SCON

Nastavitev `SCON = 0x50` pomeni:

- `SM0 = 0`, `SM1 = 1`: (*Mode 1*) 8 bitni UART z nastavljivo hitrostjo prenosa
- `REN = 1`: omogoči serijski sprejem

Nastavitev `TMOD = 0x20` pomeni:

- `M1=1`, `M0=0`: (8 bit *auto-reload timer-counter*) V `TH1` je shranjena vrednost, ki se prepiše v `TL1` vsakokrat, ko se števec resetira (4.3).

Vrednost registra `TH1` določimo glede na podano frekvenco oscilatorja ter želeno hitrost prenosa [C51_PG]

$$TH1 = 256 - (fosc/384)/BAUD$$

V kolikor je `PCON.7 = 1`

$$TH1 = 256 - (fosc/192)/BAUD$$

Ker v `TH1` zapišemo samo celoštevilski del, izberemo tistega od obeh rezultatov, pri katerem je napaka zaradi rezanja manjša.

Bit `TR1` je del SFR registra `TCON`:

- `TR1=1`: določa delovanje števca 1. Če je postavljen na 1 je števec *Counter 1* vključen.

Bit `TI` je del SFR registra `SCON`:

- `TI=1`: uporablja se pri oddajanju. Postavi ga strojna oprema, ko pri oddajanju naleti na stop bit. Pobrisati ga moramo programsko.

5.1.1 Primer 1

Kot primer si oglejmo rutino, ki inicializira serijska vrata ter na serijski izhod izpiše tekst. Inicializacijo serijskih vrat opravi funkcija `serial_init()`, ki nastavi potrebne SFR registre. Za izpis je uporabljena knjižnična funkcija `printf( )`.

Za pravilno delovanje serijskih vrat moramo nastaviti naslednje SFR registre: `SCON`, `TMOD`, `TH1`, `TR1` in `TI`.

```
#include <stdio.h>
#include <89c51rd2.h>
void serial_init()
{
        SCON = 0x50;
        TMOD = 0x20;
        PCON = 0x80;
        TH1 = 0xF7;
        TR1 = 1;
        TI = 1;
}
void main()
{
        serial_init();
        for (;;)
        {
              printf("test\n");
        }
}
```

Tabela 35: Primer 1

Z vrstico `#include <89c51rd2.h>` je projektu dodana še knjižnica mikrokrmilnika `T89C51RD2`. V knjižnici so določeni naslovi in imena registrov SFR.

5.1.2 Primer 2

V drugem primeru si oglejmo kako lahko nadomestimo knjižnično funkcijo `printf( )` z lastno funkcijo, ki bo ob klicu izpisala znak na serijska vrata.

Funkcija `poslji_znak()` izpiše znak na serijski izhod. Znake moramo zapisati v SFR register `SBUF` ter počakati toliko časa, da se bit `TI` postavi nazaj na 1, kar pomeni, da je bil znak poslan.

Zapis v novo vrstico realiziramo tako, da izpišemo zaporedje znakov `CR` (*carriage return*) (`0x0D`) in `LR` (*line feed*) (`0x0A`). Izpis znakov `LF` in `CR` nadomesti znak "`\n`", s katerim smo v prvem primeru postavili kurzor na začetek nove vrstice.

Algoritem, ki ga funkcija opravlja lahko zapišemo kot:

1. Preberi znak ter ga zapiši v SFR register `SBUF`,
2. Postavi zastavico `TI` na 0,
3. Počakaj, da se zastavica `TI` postavi na 1.

znak	ASCII
LF	0Ah
CR	0Dh
A	41h
B	42h
a	61h
b	62h
0	30h
1	31h

Tabela 36: Nekateri ASCII znaki

```
void poslji_znak(unsigned char znak)
{
        SBUF = znak;
        TI=0;
        while(!TI)
        {
        }
}
```

```
void main()
{
        serial_init();
        while (1)
        {
              poslji_znak('a');
              poslji_znak(0x0D);
              poslji_znak(0x0A);
        }
}
```

Tabela 37: Primer 2

5.2 Naloge

24. Preverite nastavitev registra `TH1` v inicializaciji serijskih vrat pri *baud rate* = 9600, *fosc* = 16,384 MHz (Tabela 33).

25. Dopolnite nalogo 23 tako, da vrednosti števca *Timer 0* izpisujete na serijska vrata. Nastavite ustrezne SFR registre za vzpostavitev serijske komunikacije. Timer1 uporabite za nastavitev hitrosti prenosa (*baud rate* = 9600, *fosc* = 16,384 MHz).

    a. Določite vrednosti SFR registrom:

    Timer 1: `SCON, TMOD, TH1, TR1, TI, PCON`

    Timer 0: `TMOD, STR0`

    b. Program preverite v okolju Keil µVision tako, da odprete okno za vrata `P3` (`Peripherals>I/O Ports>Port 3`) ter serijsko okno (`View>Serial Window #1`). Med izvajanjem programa spreminjajte vrednost bita `P3.4` ter opazujte izpis na serijskem oknu.

26. Napišite rutino, ki bo s serijskih vrat prebrala ASCII znak, ga povečala za ena ter izpisala na serijska vrata. Uporabite knjižnično funkcijo `getchar()`.

27. Funkcijo `poslji_znak()` nadgradite za pošiljanje nizov. Funkcija `poslji_niz()` izpisuje znake niza na serijski izhod. Posamezne znake niza moramo zapisati v SFR register `SBUF` ter počakati toliko časa, da se bit `TI` postavi nazaj na 1, kar pomeni, da je bil znak poslan. Postopek ponavljamo dokler niso poslani vsi znaki v nizu.

28. Zgornjo rutino popravite tako, da ne bo potrebovala knjižnične funkcije za branje znakov `getchar( )`. Uporabite lastno funkcijo, ki bo brala znake s serijskih vrat ob prekinitvi. Pri inicializaciji serijskih vrat nastavite bita `EA` in `ES`, ki omogočita prekinitve. Znak je na

voljo v registru `SBUF`, ko je postavljen bit `RI`. Ko znak preberemo moramo programsko postaviti bit `RI` na nič.

29. Sestavite preprost kalkulator. Napišite program, ki bo izvajal osnovne aritmetične operacije na 8 bitnih nepredznačenih številih. Za komunikacijo s terminalom uporabite RS232 vrata ter lastne funkcije za branje in pisanje.

6 Literatura

6.1 Knjige

[Ibrahim00] Dogan Ibrahim, Microcontroller Projects in C for the 8051, Newnes, 2000, ISBN 0 7506 46403.

[Bratkovič98] F. Bratkovič, Uvod v C, Založba FE in FRI, 1998.

6.2 Splet

[Keil] List of Supported Chips, http://www.keil.com/dd/, januar 2005.

[8052] 8052 Derivative Microcontrollers, http://www.8052.com/chips.phtml, januar 2005.

[Steiner04] Craig Steiner, The 8052 Tutorial & Reference, http://www.8052.com/, januar 2005.

6.3 Priročniki

6.3.1 Splet

[C51_AO] Architectural Overview of the C51 Family, TEMIC, 1997, http://www.keil.com/dd/chip/3176.htm, januar 2005.

[C51_HD] Hardware Description of the C51 Family Products, TEMIC, 1997, http://www.keil.com/dd/chip/3176.htm, januar 2005.

[C51_PG] C51 Programmer's Guide and Istruction Set, TEMIC, 1997, http://www.keil.com/dd/chip/3176.htm, januar 2005.

[4188D] Data Sheet, TEMIC, 1997, http://www.keil.com/dd/chip/3176.htm, januar 2005.

6.3.2 Keil µVision

[C51] *Cx51 compiler*, User's guide 02.2001, Keil µVision Software, zavihek Books.

[GS51] *Getting started with µVision2*, User's guide 02.2001, Keil µVision Software, zavihek *Books*.

7 Dodatek A: Ukazi zbirnega jezika

ukaz	C	OV	AC	ukaz	C	OV	AC
ADD	x	x	x	CLRC	O		
ADDC	x	x	x	CPL C	x		
SUBB	x	x	x	ANL C , bit	x		
MUL	0	x		ANL C,/bit	x		
DIV	0	x		ORL C,bit		x	
DA	x			ORL C,/bit		x	
RRC	x			MOV C,bit		x	
RLC	x			CJNE	x		
SETB C	x						

Tabela 38: Ukazi ki vplivajo na stanje zastavic [C51_PG]

Legenda:

Rn	registri R0-R7 izbranega nabora registrov.
direct	8 bitni naslov notranjega podatkovnega pomnilnika ali registrov SFR.
@Ri	8 bitni naslov notranjega podatkovnega pomnilnika, ki ga dostopimo z registroma R0 in R1.
# data	8 bitna konstanta, ki je del ukaza.
# data 16	16 bitna konstanta, ki je del ukaza.
addr 16	16 bitni naslov (LCALL in LJMP).
addr 11	11 bitni naslov (ACALL in AJMP).
rel	predznačen 8 bitni odmik (SJMP in pogojni skoki)
bit	direktno naslovljen bit v notranjem podatkovnem pomnilniku ali registrih SFR.

ukaz	opis	besede	periode
ADD A, Rn	Add register to Accumulator	1	12
ADD A, direct	Add direct byte to Accumulator	2	12
ADD A, @Ri	Add indirect RAM to Accumulator	1	12
ADD A, #data	Add immediate data to Accumulator	2	12
ADDC A, Rn	Add register to Accumulator with Carry	1	12
ADDC A, direct	Add direct byte to Accumulator with Carry	2	12
ADDC A, @Ri	Add indirect RAM to Accumulator with Carry	1	12
ADDC A, #data	Add immediate data to Accumulator with Carry	2	12
SUBB A, Rn	Subtract Register from Accumulator with borrow	1	12
SUBB A, direct	Subtract indirect RAM from Accumulator with borrow	2	12
SUBB A, @Ri	Subtract indirect RAM from Accumulator with borrow	1	12
SUBB A, #data	Subtract immediate data from Accumulator with borrow	2	12
INC A	Increment Accumulator	1	12
INC Rn	Increment register	1	12
INC direct	Increment direct byte	2	12
INC @Ri	Increment direct RAM	1	12
DEC A	Decrement Accumulator	1	12
DEC Rn	Decrement Register	1	12
DEC direct	Decrement direct byte	2	12
DEC @Ri	Decrement indirect RAM	1	12
INC DPTR	Increment Data Pointer	1	24
MUL AB	Multiply A & B	1	48
DIV AB	Divide A by B	1	48
DA A	Decimal Adjust Accumulator	1	12

Tabela 39: Aritmetične operacije [C51_PG]

ukaz	opis	besede	periode
ANL A,Rn	AND register to ACC	1	12
ANL A,direct	AND direct byte to ACC	2	12
ANL A,@Ri	AND indirect RAM to ACC	1	12
ANL A,#data	AND immediate data to ACC	2	12
ANL direct,A	AND ACC to direct byte	2	12
ANL direct,#data	AND immediate data to direct byte	3	24
ORL A,Rn	OR register to ACC	1	12
ORL A,direct	OR direct byte to ACC	2	12
ORL A,@Ri	OR indirect RAM to ACC	1	12
ORL A,#data	OR immediate data to ACC	2	12
ORL direct,A	OR ACC to direct byte	2	12
ORL direct,#data	OR immediate data to direct byte	3	24
XRL A,Rn	XOR register to ACC	1	12
XRL A,direct	XOR direct byte to ACC	2	12
XRL A,@Ri	XOR indirect RAM to ACC	1	12
XRL A,#data	XOR immediate data to ACC	2	12
XRL direct,A	XOR ACC to direct byte	2	12
XRL direct,#data	XOR immediate data to direct byte	3	24
CLR A	Clear the ACC	1	12
CPL A	Complement the ACC	1	12
RL A	Rotate the ACC left	1	12
RLC A	Rotate the ACC left through Carry	1	12
RR A	Rotate the ACC right	1	12
RRC A	Rotate the ACC right through Carry	1	12
SWAP A	Swap nibbles in the ACC	1	12

Tabela 40: Logični operatorji [C51_PG]

ukaz	opis	besede	periode
MOV A,Rn	Move Register to Accumulator	1	12
MOV A,direct	Move Direct byte to Accumulator	2	12
MOV A,@Ri	Move Indirect byte to Accumulator	1	12
MOV A,#data	Move Immediate data to Accumulator	2	12
MOV Rn,A	Mov Accumulator to Register	1	12
MOV Rn,direct	Move Direct byte to Register	2	24
MOV Rn,#data	Move Immediate data to Register	2	12
MOV direct,A	Move ACC to Direct byte	2	12
MOV direct,Rn	Move Register to Direct byte	2	24
MOV direct,direct	Move Direct byte to Direct byte	3	24
MOV direct,@Ri	Mov Indirect RAM to Direct byte	3	24
MOV direct,#data	Move Immediate data to Direct byte	3	24
MOV @Ri,A	Move ACC to Indirect RAM	1	12
MOV @Ri,direct	Move direct byte to indirect RAM.	2	24
MOV @Ri,#data	Move Immediate data to Indirect RAM	2	12
MOV DPTR,#data16	Load datapointer with 16 bit constant	3	24
MOVC A,@A+DPTR	Move code byte at Acc+DPTR to Accumulator	1	24
MOVC A,@A+PC	Move code byte at Acc+PC to Accumulator	1	24
MOVX A,@Ri	Move external RAM to Accumulator	1	24
MOVX @Ri,A	Move Accumulator to external RAM	1	24
MOVX A,@DPTR	Move external RAM to Accumulator	1	24
MOVX @DPTR,A	Move Accumulator to external RAM	1	24
PUSH direct	Push direct byte to stack	2	24
POP direct	Pop direct byte from stack	2	24
XCH A,Rn	Exchange register with Accumulator	1	12
XCH A,direct	Exchange direct byte with Accumulator	2	12
XCH A,@Ri	Exchange indirect RAM with Accumulator	1	12
XCHD A,@Ri	Exchange low order digit indirect RAM with Acc	1	12

Tabela 41: Branje in pisanje [C51_PG]

ukaz	opis	besede	periode
CLR C	Clear carry	1	12
CLR bit	Clear direct bit	2	12
SETB C	Set carry	1	12
SETB bit	Set direct bit	2	12
CPL C	Complement carry	1	12
CPL bit	Complement direct bit	2	12
ANL C,bit	AND direct bit to carry	2	24
ANL C,/bit	AND complement of direct bit to carry	2	24
ORL C,bit	OR direct bit to carry	2	24
ORL C,/bit	OR complement of direct bit to carry	2	24
MOV C,bit	Move direct bit to carry	2	12
MOV bit,C	Move carry to direct bit	2	24
JC rel	Jump if carry is set	2	24
JNC rel	Jump if carry is NOT set	2	24
JB bit,rel	Jump if direct bit is set	3	24
JNB bit,rel	Jump if direct bit is NOT set	3	24
JBC bit,rel	Jump if direct bit is set and clear that bit	3	24

Tabela 42: Boolovi operatorji [C51_PG]

ukaz	opis	besede	periode
ACALL addr11	Absolute Subroutine Call	2	24
LCALL addr16	Long Subroutine Call	3	24
RET	Return from Subroutine	1	24
RETI	Return from interrupt	1	24
AJMP addr11	Absolute Jump	2	24
LJMP addr16	Long Jump	2	24
SJMP rel	Short Jump (relative address)	2	24
JMP @A+DPTR	Jump direct relative to the DPTR	1	24
JZ rel	Jump if Accumulator is Zero	2	24
JNZ rel	Jump if Accumulator is not Zero	2	24
CJNE A,direct,rel	Compare direct byte to Acc and Jump if Not	3	24
CJNE A,#data,rel	Compare immediate to Acc and Jump if Not Equal	3	24
CJNE Rn,#data,rel	Compare immediate to register and Jump if Not Equal	3	24
CJNE@Ri,#data,rel	Compare immediate to indirect and Jump if Not Equal	3	24
DJNZ Rn,rel	Decrement register and Jump if Not Zero	2	24
DJNZ direct,rel	Decrement direct byte and Jump if Not Zero	3	24
NOP	No Operation	1	12

Tabela 43: Vejitve in skoki [C51_PG]

8 Dodatek B: Operatorji

Operator	Opis	Primer
+	seštevanje	`x = a + b;`
-	odštevanje	`x = a - b;`
*	množenje	`x = a * b;`
/	deljenje	`x = a / b;`
%	modulo	`x = a % b;`

Tabela 44: Aritmetični operatorji

Operator	Opis	Primer
==	je enak	`if (i= =0)`
!=	je različen	`if (i !=0)`
<	je manjši	`for (int i = 0; i < 5; i++)`
<=	je manjši ali enak	`for (int i = 0; i <= 5; i++)`
>	je večji	`for (int i = 10; i > 5; i--)`
>=	je večji ali enak	`for (int i = 10; i >= 5; i--)`

Tabela 45: Primerjalni operatorji

Operator	Opis	Primer
&&	in	`if ((x == 5) && (y == 6))`
||	ali	`if ((x == 5) || (y == 6))`
!	ne	`if (!x)`

Tabela 46: Logični operatorji

Operator	Opis	Primer
&	in	`y = x & 0xAA;`
|	ali	`y = x | 0xAA;`
^	ex-ali	`y = x ^ 0xAA;`
<< n	pomik v levo za n bitov	`y = x << 1;`
>> n	pomik v desno za n bitov	`y = x >> 2;`
~	eniški komplement	`y = ~x;`
-	dvojiški komplement	`y = -x;`

Tabela 47: Bitni operatorji

9 Dodatek C: Uvod v Keil µVision

9.1 Projekt v zbirnem jeziku

Najprej odprimo nov projekt `Project>New project`. Izberemo mapo, kjer bodo shranjene datoteke projekta. Izberemo ime projekta. Projekt ima končnico `*.uv2`. Odpre se okno `Select Device for Target 'Target 1'`. Izberemo ciljni mikrokrmilnik. V seznamu proizvajalcev izberemo `Atmel`, ter nadalje `T89C51RD2`. V desnem delu okna vidimo karakteristike izbranega mikrokrmilnika. Nadalje potrdimo `Copy Standard 8051 Startup Code to Project Folder and Add File to Project`.

V levem oknu (zavihek `Files`) se pojavi mapa z imenom `Target 1`. S klikom na `+` pred mapo se nam prikaže vsebina projekta. Znotraj mape `Target 1` je mapa `Source Group 1`, v kateri je trenutno samo datoteka `STARTUP.A51`. V tej mapi se bodo nahajali naši programi napisani v zbirnem jeziku ali v jeziku C.

Z desnim klikom na `Target 1` odpremo meni, kjer lahko izbiramo ciljni mikrokrmilnik (`Select Device for Target 'Target 1'`) in nastavitve (`Options for Target 'Target1'`). K nastavitvam se bomo vrnili kasneje.

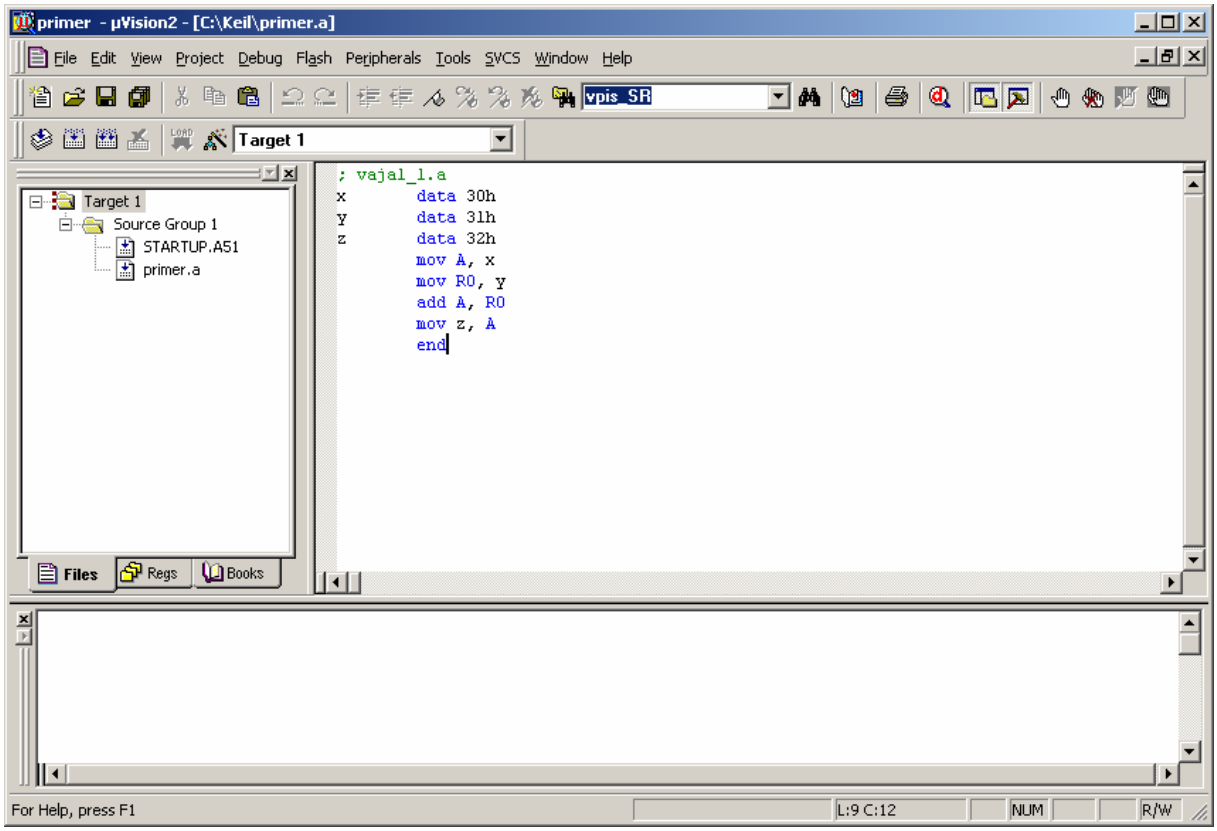
Oglejmo si kako v projekt dodajamo izvorne programe. V kolikor program še nismo napisali izberemo `File>New` s katerimi odpremo novo tekstovno okno. Tekstovno okno shranimo z ukazom `File>Save as` v mapo, kjer se nahaja naš projekt. Izberemo primerno ime ter končnico. V primeru, da bomo program napisan v jeziku C bomo izbrali končnico `*.c`. V kolikor pa bo program napisan v zbirnem jeziku pa bomo uporabili končnico `*.a`.

Z desnim klikom na mapo `Source Group 1` odpremo meni: `Add Files to Group 'Source Group 1'`. Izberemo datoteko, ki smo jo pravkar shranili in izberemo `Add` in `Close`. V mapi `Source Group 1` sedaj vidimo izvorno datoteko. Na tak način dodamo tudi ostale datoteke, ki sestavljajo projekt.

Slika 13 prikazuje primer projekta, ki ga sestavlja izvorna datoteka `primer1.a` v okolju Keil µVision [Keil]. V desnem oknu je odprt preprost program napisan v zbirnem jeziku mikrokontrolerja 8051. Levo okno ima tri zavihke:

- *Files* prikazuje organizacijo datotek v projektu
- *Regs* prikazuje stanje registrov v načinu *debug*
- *Books* pa predstavlja knjižnico knjig za pomoč

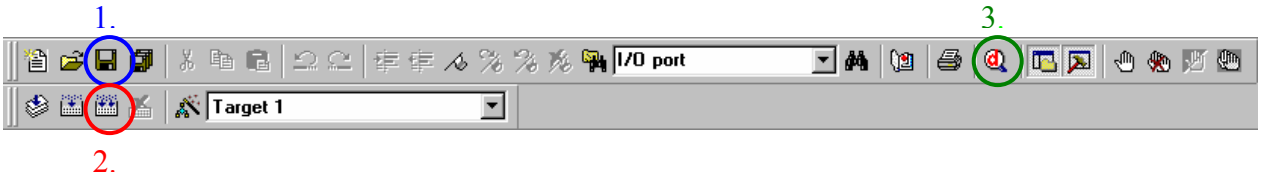
V spodnjem oknu nam program izpisuje sporočila. Tako na primer pri prevajanju programa javi napake, itd.



Slika 13: Programsko okolje Keil µVision

V nadaljevanju si oglejmo korake s katerimi dani projekt shranimo, prevedemo in poženemo. Slika 14 prikazuje naslednje korake:

1. Shrani (`File>Save`)
2. Prevedi (`Project>Rebuild all target files`)
3. Izvedi (`Debud>Start/Stop debug session`)

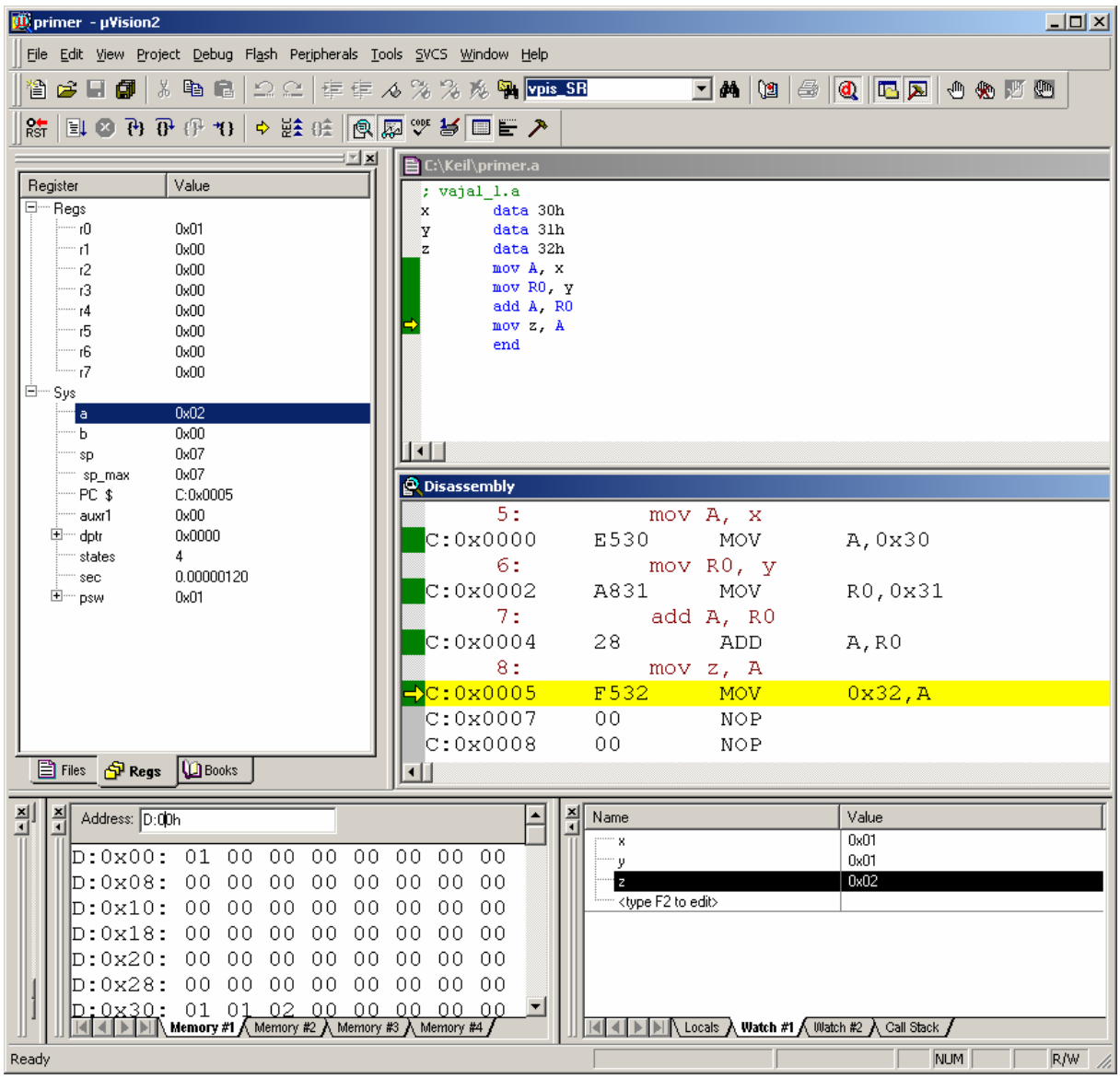


Slika 14: Orodna vrstica v osnovnem načinu

Po tretjem koraku se nahajamo v načinu *debug*, kjer preverimo delovanje programa. V nadaljevanju je na kratko opisan postopek dela v načinu *debug*.

9.1.1 Način *debug*

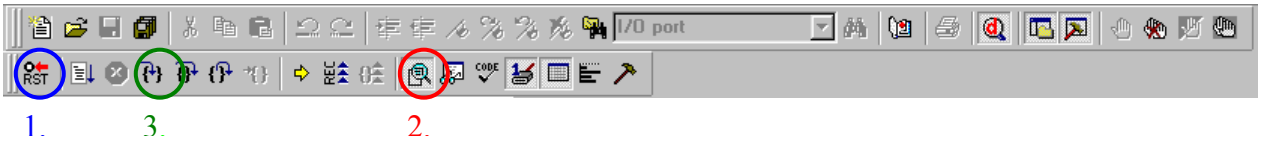
Slika 15 prikazuje programsko okolje v načinu *debug*. Osrednji del sestavljajo tri okna. V levem oknu lahko spremljamo vrednosti posameznih registrov. V srednjem oknu je program v zbirnem jeziku. Desno okno pa je okno *disassembly*, kjer je program preveden v strojno kodo.



Slika 15: Programsko okolje v načinu *debug*

Slika 16 prikazuje orodno vrstico v načinu *debug*. S številkami so označeni naslednji osnovni koraki:

1. Reset (`RST`),
2. Okno `disassembly`,
3. `Step into`, s katerim izvedemo posamezen ukaz.



Slika 16: Orodna vrstica v načinu *debug*

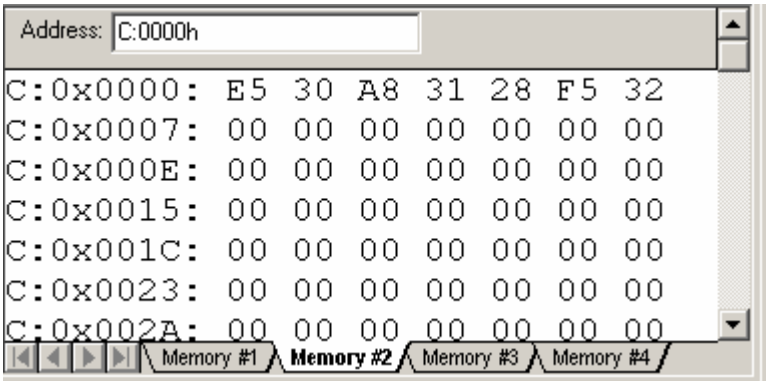
9.1.2 Izvajanje programa

S pritiskom na reset (RST) resetiramo program in postavimo programski števec na začetek. Okno *disassembly* (Slika 15) prikazuje naš program preveden v strojno kodo. Z desnim klikom na okno se odpre meni, kjer vidimo, da je izbran način *Mixed mode*, ki združuje posamezne vrstice našega programa, stanje programskega števca (PC), strojno kodo ukaza in odgovarjajoč ukaz zapisan v zbirnem jeziku. Za razliko od vrstic našega programa je prevajalnik namesto spremenljivk, ki smo jih deklarirali s psevdoukazi, vstavil naslove posameznih pomnilniških lokacij. Tako je na primer `y` zamenjan z naslovom `0x31`.

Program lahko izvedemo v celoti (`Debug>Go`) ali pa po korakih (`Debug>Step into`).

9.1.3 Pomnilniški prostor

V spodnjem desnem oknu so prikazani posamezni segmenti pomnilnika. Ker se v okencu (*Address*) nahaja naslov `C: 00h` vidimo programski del pomnilnika od naslova `0000h` dalje. Na voljo imamo 4 zavihke, tako da lahko spremljamo do 4 različne segmente pomnilnika, npr. programski pomnilnik, notranji podatkovni pomnilnik, zunanji podatkovni pomnilnik, SFR registre, itd.



Slika 17 Vsebina programskega pomnilnika

Če bi želeli opazovat notranji podatkovni pomnilnik, kjer se nahajajo podatki bi prvo izbrali zavihek *Memory #1*, kot naslov pa vpisali `D: 00h` (Slika 15). V prvi vrstici vidimo vrednosti posameznih registrov `R0 – R7`. Podatki pa se nahajajo od naslova `30h` dalje.

primer	opis
`C: 0000h`	programski pomnilnik (`C` – *code*); naslov `0000h`
`D: 30h`	notranji podatkovni pomnilnik (`D` – *data*); naslov `30h`
`X: 0000h`	zunanji podatkovni pomnilnik (`X` – *external*); naslov `0000h`

Tabela 48: Oznake pomnilniških prostorov

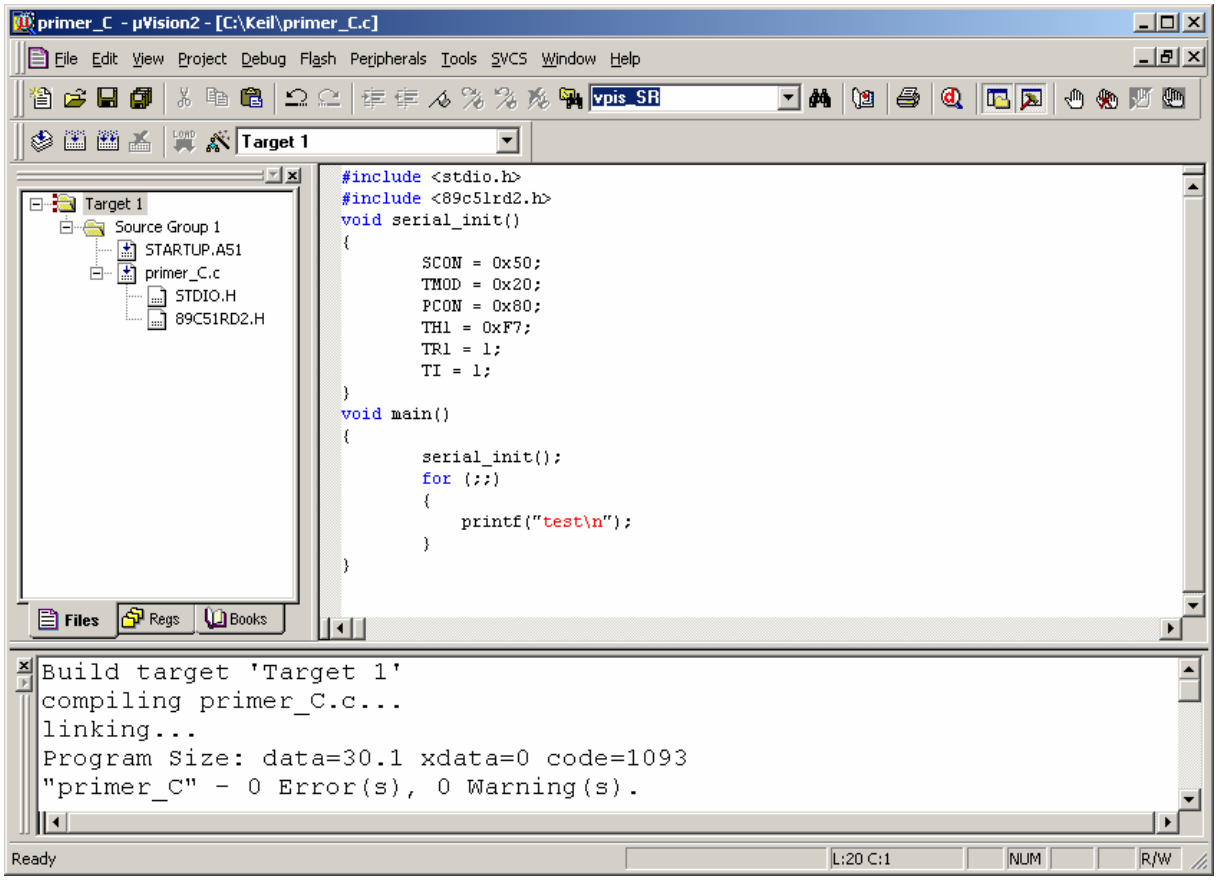
9.1.4 Okno *Watch*

Vrednosti spremenljivk in izrazov lahko opazujemo med izvajanjem programa z oknom *Watch*. Okno *Watch* odpremo z `View>Watch & Call Stack Window`. V zavihek *Watch #1* vpišemo imena spremenljivk, katere želimo opazovati.

9.2 Projekt v jeziku C

Oglejmo si še kako kreiramo projekt v jeziku C. Za primer vzemimo program (Tabela 35), ki je opisan v 5.1.1. Začetni koraki so enaki, kot pri projektu v zbirnem jeziku (9.1), le da ima datoteka končnico `*.c`.

V oknu `Files` vidimo organizacijo projekta. Datoteki `primer_C.c` sta dodani še obe knjižnici `STDIO.H` in `89C51RD2.H`. Prva je potrebna zaradi uporabe knjižnične funkcije `printf()`, druga pa je potrebna za izvajanje programa na mikrokrmilniški plošči z mikrokrmilnikom `T89C51RD2`.

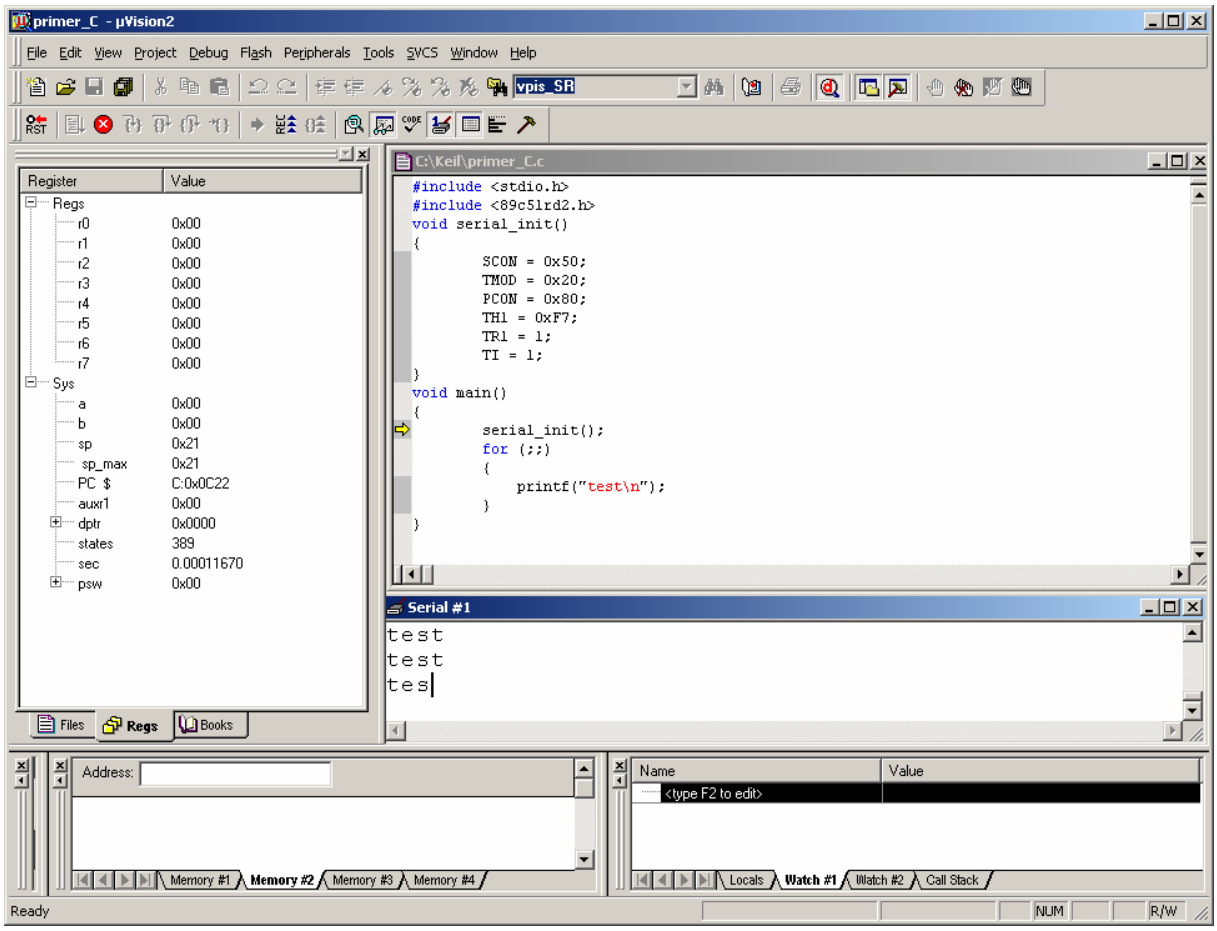


Slika 18 Projekt v jeziku C

9.2.1 Izvajanje programa

Delovanje programa lahko testiramo v okolju Keil µVision podobno, kot v primeru v zbirnem jeziku (9.1.2). Progam moramo shraniti ter prevesti, izvajamo pa ga v načinu *debug*.

Da bomo lahko opazovali stanje serijskih vrat izberemo `View>Serial Window #1`. Program zaženemo z ukazom `Debug>Go`.



Slika 19 Testiranje programa v načinu *debug*

9.2.2 Izhodna *.hex datoteka

Ko imamo delujoč program izdelamo izhodno `*.hex` datoteko, ki jo bomo naložili na mikrokrmilniško ploščo. V osnovnem načinu izberemo `Project>Options for Target 'Target 1'`.

V zavihku *Target* določimo frekvenco kristala oscilatorja na 16,384 MHz, v zavihku *Output* pa `Create HEX File`.

Projekt moramo na novo prevesti. Izhodna datoteka `*.hex` se nahaja v projektni mapi.