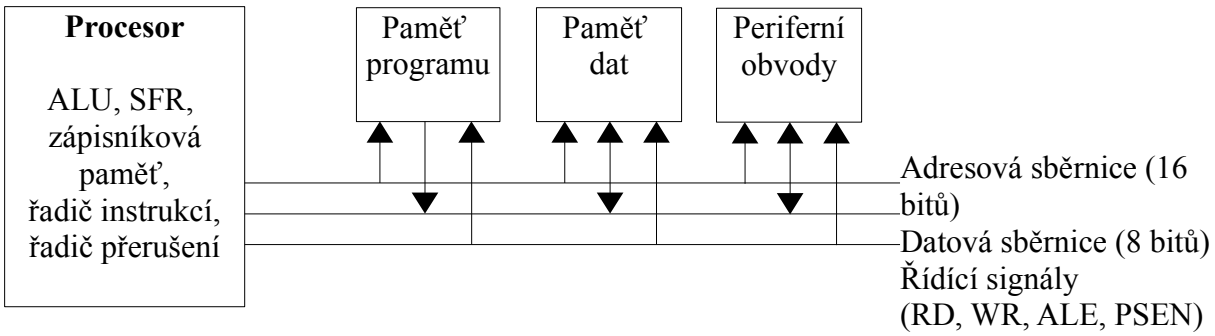
Programátorský model procesoru x51

Základní schéma procesoru

V rámci cvičení tohoto předmětu budeme programovat jeden konkrétní procesor řady x51. Abychom ho mohli začít programovat, musíme si nejprve představit jeho zjednodušený model. Postupně bude tento model doplňován a upřesňován tak, aby poskytl ucelenou představu o funkcích a možnostech jednočipových procesorů.

Procesory řady x51 používají harvardskou architekturu, tzn. mají oddělenou paměť pro program a paměť pro data. Programové pamětí i paměti pro data se bude věnovat dále tento text podrobněji. Pro tuto chvíli budeme pouze předpokládat, že v programové paměti jsou uloženy instrukce programu, který se má vykonat. Datová paměť (někdy se také označuje jako zápisníková paměť – scratch-pad memory) pak slouží pro ukládání obecných dat a obsahuje i blok speciálních funkčních registrů (SFR).



Celý procesor je osmibitový a proto datová sběrnice, všechny paměťové buňky i speciální funkční registry mají osm bitů (až na několik výjimek, viz dále). Adresová sběrnice procesorů řady x51 je šestnáctibitová. To znamená, že můžeme adresovat maximálně 2^{16} = 65536 buněk v datové i programové paměti. Řídící signály slouží k řízení a časování na sběrnici – signály RD a WR jako hodiny pro čtení a zápis do datové paměti, signál PSEN jako hodinový signál pro čtení z programové paměti a signál ALE slouží pro řízení multiplexované adresové sběrnice (bližší podrobnosti v kapitole o paměti).

Aby procesor mohl komunikovat s okolním světem, potřebuje i nějaké vstupy a výstupy. Ty jsou také osmibitové a jsou mapovány v paměti SFR. Pracuje se s nimi tedy stejným způsobem, jako s datovou pamětí.

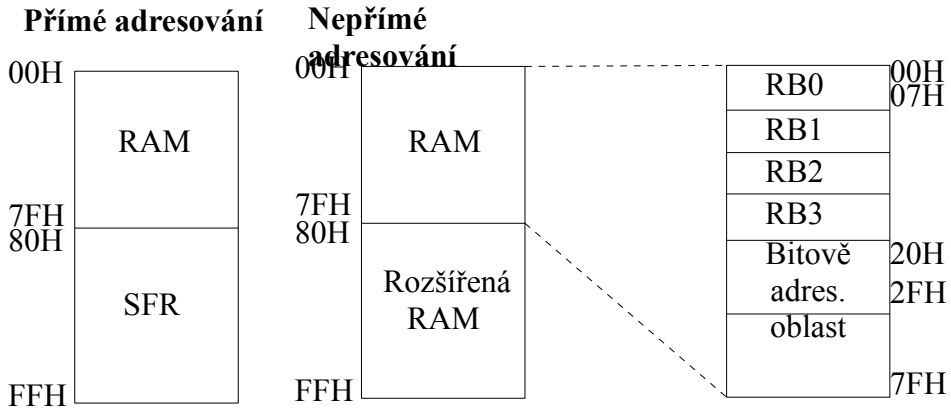
Struktura paměti

Procesory řady x51 mají tři oddělené paměťové prostory:

- programová paměť (označovaná jako CDATA) obsahuje program, může být integrovaná v pouzdře procesoru nebo může být připojena jako externí obvod
- datová paměť (označovaná jako XDATA) není samozřejmá a může mít maximálně 64kB. Může být integrovaná v pouzdře s procesorem (např. 2 kB nebo 4 kB) nebo připojená jako externí obvod. Je určena pro ukládání většího množství dat, práce s ní je pomalejší, než se zápisníkovou pamětí
- zápisníková paměť (označená jako IDATA) – tento paměťový prostor sdílí datová paměť o velikosti 128 nebo 256 bytů a oblast registrů SFR, která zabírá 128 bytů. Protože adresovatelný prostor je pouze 256 bytů, je nutné překrytí datové paměti a

SFR registrů. Struktura paměti je proto různá pokud použijeme přímé nebo nepřímé adresování (viz obrázek)

Na začátku základní oblasti datové paměti jsou umístěny čtyři registrové banky (RB0..RB3). Každá registrová banka má vyhrazenu oblast o velikosti 8 bytů. V rámci této oblasti lze pro adresování buněk používat symboly R0..R7. Která registrová banka má být právě aktivní (ve které oblasti budou registry R0..R7) lze nastavit pomocí bitů RB0 a RB1 v registru PSW.



V oblasti SFR jsou pro potřeby programátora k dispozici tyto nejdůležitější registry:

- Akumulátor = registr A nebo ACC – speciální registr, jediný možný operand některých instrukcí
- Registr B – obecně použitelný registr, slouží jako implicitní operand pro instrukce násobení a dělení
- Vstupně/výstupní porty P0..P3
- Registr DPTR (Data Pointer) – slouží pro nepřímé adresování v datové paměti, je šestnáctibitový ale přistupujeme k němu osmibitově pomocí registrů DPL a DPH
- Registr PC (Program Counter) – tento registr obsahuje adresu aktuálně vykonávané instrukce v programové paměti. Je šestnáctibitový, takže program může mít maximálně 65536 instrukcí
- Registr SP (Stack Pointer) – ukazatel na vrchol zásobníku
- Registr PSW (Program State Word) – stavové slovo procesoru. Struktura PSW:

C	AC	F0	RS1	RS 0	OV	-	P

Některé další registry z oblasti SFR budou uvedeny dále v textu. Oblast SFR ale není využita plně, ale je v ní mnoho rezervovaných oblastí. Ty jsou pak využívány v různých klonech procesorů řady x51 pro ovládání dalších periférií a funkcí, které základní procesor 8051 nemá.

Instrukce

Jednotlivé instrukce jsou uloženy za sebou v programové paměti. Každá instrukce se v paměti skládá z operačního kódu a operandů. V reálném mikroprocesoru může zápis instrukce v paměti zabírat různé množství paměťových buněk. Instrukce, která nemá žádné operandy, zabere pouze jednu buňku. Naopak instrukce, která např. ukládá konstantu do nějakého registru, zabere tři buňky. Registr PC (Program Counter) obsahuje

adresu aktuálně vykonávané instrukce a po startu procesoru je v něm hodnota 0. Po každém vykonání instrukce přičte řadič instrukcí do registru PC velikost instrukce v paměti a tak PC ukazuje na další instrukci, která se má vykonat. Jedinou vyjímkou jsou instrukce skoku, po kterých se může vykonávat i jiná instrukce, než která následuje za touto instrukcí.

Operační kód instrukce je v paměti samozřejmě uložen jako číslo. Instrukcí sice procesor nemá mnoho (jednočipové mikroprocesory mají řádově okolo 100 různých instrukcí), ale stejně je pro člověka nepředstavitelné, že by znal všechny jejich operační kódy zpaměti a pamatoval si jejich význam. Navíc by si musel pamatovat i kódy jednotlivých operandů a v případě skoků by musel při každé změně v programu přepočítat adresy instrukcí v paměti.

Takový program by se nejen obtížně psal, ale i četl a ladil. Proto se procesory neprogramují přímo ve strojovém kódu ale v takzvaném jazyce symbolických adres (JSA), který se někdy označuje jako assembler. V tomto jazyce je každé instrukci přiřazena tzv. mnemonická zkratka. Stejně tak jsou zavedeny jednoduchá jména pro registry procesoru místo jejich čísel. Také je možné používat v takovém programu symbolická návěstí pro označení místa, kam ukazuje instrukce skoku.

Program napsaný v assembleru je pak nutné přeložit pomocí překladače assembleru do strojového kódu. Překladač nahradí jména instrukcí, registrů a návěstí konkrétními čísly a poskládá instrukce za sebe do paměti procesoru. Toto je triviální úloha, která nám ale přináší obrovský komfort při psaní programů na úrovni jednotlivých instrukcí.

Instrukční sada

V následujícím textu si budeme postupně představovat jednotlivé instrukce našeho virtuálního procesoru. Nebudeme se ale již zabývat jejich číselnou interpretací a ukládáním v paměti, ale budeme pracovat na úrovni JSA.

Taková instrukce pak vypadá např. takto

```
MOV A,
R1
```

Zápis začíná vždy mnemotechnickou zkratkou instrukce a následuje seznam operandů, které jsou odděleny čárkou. Operandy mohou být určeny implicitně, nebo explicitně. Implicitní adresování znamená, že adresa je již přímo součástí instrukce. U explicitního adresování můžou být použity následující způsoby určení adresy:

- přímý operand – adresa nultého řádu, immediate address
- přímá adresa – adresa prvního řádu, pořadové číslo registru, portu nebo paměťové buňky
- nepřímá adresa – adresa druhého řádu, adresa není součást operandu ale na adrese, kterou určuje operand, je uložena adresa, se kterou chceme pracovat
- adresování ukazateli – speciální registry, jejichž obsah ukazuje na příslušnou buňku (PC, DPTR, SP)
- relativní adresa – adresová část neurčuje místo v paměti, ale jen posunutí (relativní adresu). Skutečnou adresu pak získáme přičtením relativní adresy ke vztažné adrese

Číselné hodnoty je možné uvádět zcela rovnocenně v dekadické, hexadecimální nebo binární soustavě. Zápis číselné hodnoty musí začínat číslem 0..9 a pokud není v dekadické soustavě, musí končit písmenem, které určuje soustavu (B=binární, H=hexadecimální).

Instrukce procesorů řady x51 lze rozdělit podle účelu do těchto základních skupin:

- prázdná instrukce – NOP
- operace přesunu – MOV, MOVX, MOVC, XCH, XCHD, PUSH, POP
- aritmetické operace – INC, DEC, ADD, ADDC, SUBB, DA, MUL, DIV, DA
- logické operace – ANL, ORL, XRL, CPL
- rotace – RL, RLC, RR, RRC, SWAP
- bitové operace – MOV, CLR, SETB, CPL, ANL, ORL
- instrukce skoků – SJMP, LJMP, AJMP, LCALL, ACALL, RET, RETI
- podmíněné skoky – JB, JNB, JC, JNC, JZ, JNZ
- sdružené instrukce – DJNZ, CJNE

Konstanty je možné ekvivalentně zadávat ve dvojkové, desítkové a šestnáctkové soustavě. Příklad:
10 = 0Ah = 00001010b.
Pro zjednodušení zápisu adres v paměti nebo jiných číselných konstant je možné používat direktivu překladače EQU na začátku souboru se zdrojovým kódem. Příklad použití:

```
MUJBYTE    EQU   15      ; Definice konstanty
NULA       EQU   0
MOV   MUJBYTE,#NULA   ; Použití konstanty
```

Operace přesunu dat

Přesuny v datové paměti

MOV <cíl>,<zdroj>

Instrukce pro přesun dat má mnemotechnickou zkratku MOV, což je zkratka z anglického slova „move“. Tato instrukce provede překopírování dat ze zdroje do cílového registru (resp. paměťové buňky nebo bitu). Podle typu operandů můžeme přesouvat 1bitová, 8mi bitová nebo 16ti bitová data. Podle kombinace operandů zabírá tato instrukce v programové paměti 1 až 3 byty a její vykonání trvá jeden nebo dva instrukční cykly.

Seznam možných kombinací operandů pro osmibitové přesuny:

Instrukce	Příklad
MOV A,Rn	MOV A,R1 – do akumulátoru nahraje obsah registru R1, n=0..7
MOV A,<adresa>	MOV A,10 – přímé adresování, do akumulátoru nahraje obsah paměťové buňky, která má adresu 10
MOV A,@Ri	MOV A,@R0 – nepřímé adresování, do akumulátoru nahraje obsah paměťové buňky, jejíž adresa je uložena v registru R0, i=0..1
MOV A,#<data>	MOV A,#10 – do akumulátoru nahraje přímo hodnotu 10
MOV Rn,A	MOV R5,A – do registru R5 nahraje obsah akumulátoru, n=0..7
MOV Rn,<adresa>	MOV R5,15 – do registru R5 nahraje obsah paměťové buňky, která má adresu 15, n=0..7
MOV Rn,#<data>	MOV A,#10 – do registru R5 nahraje přímo hodnotu 15, n=0..7
MOV <adresa>,A	MOV 10,A – do paměťové buňky s adresou 10 nahraje obsah

	akumulátoru
MOV <adresa>,Rn	MOV 15,R2 – do paměťové buňky s adresou 15 nahraje obsah registru R2, n=0..7
MOV <adresa>,<adresa>	MOV 15,10 – do paměťové buňky s adresou 15 nahraje obsah paměťové buňky s adresou 10
MOV <adresa>,@Ri	MOV 11,@R0 – do paměti na adresu 11 nahraje obsah paměťové buňky, jejíž adresa je uložena v registru R0, i=0..1
MOV <adresa>,#<data>	MOV 12,#10 – do paměti na adresu 10 nahraje přímo hodnotu 10
MOV @Ri,A	MOV @R0,A – do paměťové buňky, jejíž adresa je uložena v registru R0, nahraje obsah akumulátoru, i=0..1
MOV @Ri,<adresa>	MOV @R0,18 – do paměťové buňky, jejíž adresa je uložena v registru R0, nahraje obsah paměťové buňky na adrese 18, i=0..1
MOV @Ri,#<data>	MOV @R1,#20 – do paměťové buňky, jejíž adresa je uložena v registru R1, nahraje hodnotu 20, i=0..1

Seznam možných kombinací operandů pro jednobitové přesuny:

Instrukce	Příklad
MOV C,<bit>	MOV C,P1.0 – do bitového registru C zkopíruje hodnotu ze vstupu P.0
MOV <bit>,C	MOV P2.1,C – na výstup P2.1 zkopíruje obsah jednobitového registru C

Seznam možných kombinací operandů pro šestnáctibitové přesuny:

Instrukce	Příklad
MOV DPTR,#<data16>	MOV DPTR,#1234 – do registru šestnáctibitového DPTR nahraje zadanou hodnotu
MOV <bit>,C	MOV P2.1,C – na výstup P2.1 zkopíruje obsah jednobitového registru C

Přesuny v programové paměti

MOVC A,A+<bázový registr>

Instrukce	Příklad
MOVC A,@A+DPTR	Jako adresa zdroje bude použit součet registru DPTR a akumulátoru
MOVC A,@A+PC	Jako adresa zdroje bude použit součet registru PC a akumulátoru

Přesuny v externí datové paměti

MOVX <cíl>,<zdroj>

Instrukce	Příklad
MOVX A,@Ri	MOVX A,@R0 – do akumulátoru nahraje obsah paměťové buňky z externí paměti, jejíž adresa je uložena v registru R0, i=0..1
MOVX A,@DPTR	MOVX A,@DPTR – do akumulátoru nahraje obsah paměťové buňky z externí paměti, jejíž adresa je uložena v registru DPTR
MOVX @Ri,A	MOVX @R1,A – do paměťové buňky z externí paměti, jejíž adresa je uložena v registru R0, nahraje obsah akumulátoru, i=0..1
MOVX @DPTR,A	MOVX @DPTR,A – do paměťové buňky z externí paměti, jejíž adresa je uložena v registru DPTR, nahraje obsah akumulátoru

Z této tabulky je vidět, že není možné použít jakoukoli kombinaci operandů. Operand R_n znamená, že lze použít libovolný z univerzálních registrů R0.. R7. Pokud jako operand chceme použít číselnou konstantu, pak před ní musíme napsat znak #. Pokud to neuděláme, bude tato konstanta chápána jako adresa v paměti. Takže např. instrukce „mov A , #1" uloží do registru A konstantu 1, kdežto instrukce „mov A , 1" uloží do registru A hodnotu, která je uložena v paměti na adrese 1.
Znak @ slouží pro nepřímé adresování. Takže např. pokud do registru R0 nahrajeme hodnotu 1, pak instrukce „mov A , @R0" uloží do registru A hodnotu, která je v paměti na adrese, která je uložená v registru R0. V tomto případě se do A opět nahraje hodnota z paměti na adrese 1.
Podobně je to i s registrem DPTR. Tento registr je potřeba použít v případě, že chceme pracovat s adresou nad 255. To je totiž maximální hodnota, kterou je možné nahrát do registrů R0..R7, protože jsou osmibitové. Proto místo těchto registrů lze pro nepřímé adresování použít registr DPTR, který je šestnáctibitový a tak umožňuje pracovat s pamětí až do adresy 65535.

Výměna hodnoty s akumulátorem

XCH A,<proměnná>

Instrukce	Příklad
XCH A,Rn	XCH A,R0 – vymění obsah akumulátoru s obsahem registru R0, n=0..7
XCH A,<adresa>	XCH A,10 – vymění obsah akumulátoru s obsahem buňky na adrese 10
XCH A,@Ri	XCH A,@R1 – vymění obsah akumulátoru s obsahem registru, jehož adresa je uložena v registru R1, i=0..1

Výměna dolních čtyř bitů registru s akumulátorem

XCHD A,@Ri

Tato instrukce zjednodušuje některé operace s čísly v BCD kódu. Umožňuje vyměnit spodní čtyři bity akumulátoru se spodními čtyřmi bity paměťové buňky, jejíž adresa je uložena v registru Ri (i=0..1). Horní poloviny obou registrů zůstávají zachovány.

Práce se zásobníkem

Zásobník je oblast v paměti, která slouží k dočasnému ukládání dat. Její velikost je proměnná (podle množství vkládaných dat) a její umístění v paměti je dáno počátečním nastavením registru SP (Stack pointer). Při vkládání hodnoty pomocí instrukce PUSH se nejprve inkrementuje registr SP a poté se na adresu v paměti, která je v registru SP uložena, uloží hodnota ze zvoleného registru. Instrukce POP používá opačný postup – nejprve nahraje hodnotu z paměťové buňky, jejíž adresa je v registru SP. Poté dojde k dekrementaci registru SP. Zásobník tak funguje jako LIFO (Last In First Out) a používá se např. k ukládání návratových adres nebo k předávání hodnot mezi podprogramy. Po resetu procesoru je registr SP nastaven na hodnotu 7. Pokud chceme používat paměť i v oblasti adres osm a výše, je nutné vrchol zásobníku na začátku programu nastavit na vyšší hodnotu. Pokud bychom změnili obsah registru SP za běhu programu (jinak než použitím instrukcí PUSH a POP), může dojít k nevypočitatelnému chování procesoru. Protože zásobník sdílí interní datovou paměť je nutné dbát na dvě pravidla:

- pro zásobník je nutné vyhradit tolik prostoru, aby při svém zvětšování nemohl začít přepisovat ostatní data v paměti
- každé vložení do zásobníku musí být dříve nebo později následováno výběrem ze zásobníku – pokud bychom zapomínali vybírat, bude jeho velikost neustále narůstat a bude přepisovat i ostatní proměnné v paměti

Instrukce	Příklad
PUSH <adresa>	PUSH R0 – uloží obsah registru R0 na vrchol zásobníku
POP <adresa>	POP R1 – nahraje hodnotu z vrcholu zásobníku do registru R1

Aritmetické instrukce

Inkrementace, dekrementace

INC <registr>

Tato instrukce přičte k hodnotě daného registru (nebo paměťové buňky) hodnotu jedna. Neovlivňuje ale žádný stavový bit (carry, zero apod.).

Instrukce	Příklad
INC A	INC A – zvýší obsah akumulátoru o 1
INC Rn	INC R0 – zvýší obsah registru R0 o 1, n=0..7
INC <adresa>	INC 10 – zvýší o hodnotu 1 obsah paměťové buňky, která je na adrese 10
INC @Ri	INC @R1 – zvýší o hodnotu 1 obsah paměťové buňky, která je na adrese, uložené v registru R1, i=0..1
INC DPTR	INC DPTR – zvýší o 1 hodnotu registru DPTR, a to 16ti bitově

DEC <registr>

Tato instrukce pracuje podobně, jako instrukce INC. Také neovlivňuje žádný stavový bit. Rozdíl je pouze ten, že tuto instrukci nelze použít pro přímou dekrementaci registru DPTR.

Instrukce	Příklad
DEC A	DEC A – sníží obsah akumulátoru o 1
DEC Rn	DEC R0 – sníží obsah registru R0 o 1, n=0..7
DEC <adresa>	DEC 10 – sníží o hodnotu 1 obsah paměťové buňky, která je na adrese 10
DEC @Ri	DEC @R1 – sníží o hodnotu 1 obsah paměťové buňky, která je na adrese, uložené v registru R1, i=0..1

Sčítání a odečítání

ADD A,<zdroj>

Tato instrukce přičte k aktuální hodnotě akumulátoru hodnotu ze zdroje a výsledek uloží opět do akumulátoru. Tato instrukce ovlivňuje tyto stavové bity:

- C (Carry) – pokud dojde k přenosu do vyššího řádu (výsledek součtu je větší, než 255), pak bude mít tento bit hodnotu 1, v opačném případě 0. Při sčítání neznaménkových hodnot tento bit indikuje přetečení
- AC (Auxiliary Carry) – tento bit je analogický bitu Carry, jen indikuje přenos ze spodních čtyř bitů do horních čtyř bitů (hodnota akumulátoru se změní z menší než 16 na větší nebo rovnou 16)
- OV (Overflow) – tento bit je nastaven na 1 pokud dojde k přenosu z bitu 6 ale zároveň nedojde k přenosu z bitu 7 nebo pokud dojde k přenosu z bitu 7 a zároveň nedojde k přenosu z bitu 6. V ostatních případech je tento bit nastaven na 0. Při sčítání dvou znaménkových hodnot indikuje tento bit vznik pozitivního čísla ze dvou negativních nebo negativního čísla ze dvou pozitivních

Instrukce	Příklad
ADD A,Rn	ADD A,R0 – přičte k hodnotě akumulátoru hodnotu z registru R0, n=0..7
ADD A,<adresa>	ADD A,10 – přičte k hodnotě akumulátoru hodnotu z paměťové buňky, která je na adrese 10
ADD A,@Ri	ADD A,@R1 – přičte k hodnotě akumulátoru hodnotu z paměťové buňky, která je na adrese uložené v registru R1
ADD A,#<data>	ADD A,#20 – přičte k hodnotě akumulátoru hodnotu 20

Příklady na sčítání znaménkových čísel:
1, provedeme sečtení čísel 100 a -50, čísla chápejme jako znaménková, kódovaná jako dvojkový doplněk

Číslo 100	0 1 1 0 0 1 0 0
Číslo -50	+ 1 1 0 0 1 1 1 0
Přenos	1 1 1 1
Výsledek	1 0 0 1 1 0 0 1 0

Hodnoty stavových bitů na konci výpočtu:
C=1 – došlo k přenosu ze sedmého bitu
AC=1 – došlo k přenosu ze třetího bitu
OV=0 – došlo k přenosu ze šestého i sedmého bitu

2, provedeme sečtení čísel -100 a 50, čísla chápejme jako znaménková, kódovaná jako dvojkový doplněk

Číslo -100	1 0 0 1 1 1 0 0
Číslo 50	+ 0 0 1 1 0 0 1 0
Přenos	1 1
Výsledek	1 1 0 0 1 1 1 0

Hodnoty stavových bitů na konci výpočtu:
C=0 – nedošlo k přenosu ze sedmého bitu
AC=0 – nedošlo k přenosu ze třetího bitu
OV=0 – nedošlo k přenosu ze šestého ani sedmého bitu

3, provedeme sečtení čísel 100 a 100, čísla chápejme jako znaménková, kódovaná jako dvojkový doplněk

Číslo 100	0 1 1 0 0 1 0 0
Číslo 100	+ 0 1 1 0 0 1 0 0
Přenos	1 1 1
Výsledek	1 1 0 0 1 0 0 0

Hodnoty stavových bitů na konci výpočtu:
C=0 – nedošlo k přenosu ze sedmého bitu
AC=0 – nedošlo k přenosu ze třetího bitu
OV=1 – došlo k přenosu ze šestého bitu, ale ne ze sedmého

ADDC A,<zdroj>

Instrukce ADDC pracuje naprosto stejným způsobem, jako instrukce ADD. Má i stejné povolené kombinace operandů. Jediným rozdílem je to, že k součtu hodnot se přičítá ještě hodnota bitu Carry. Pokud má tedy bit Carry hodnotu 1, pak se k výslednému součtu přičte hodnota 1.

SUBB A,<zdroj>

Podobně jako instrukce ADDC provádí součet s hodnotou a bitem Carry, instrukce SUBB odečte od akumulátoru zadanou hodnotu a bit Carry. Výsledek pak uloží zpět do akumulátoru. Tato instrukce ovlivňuje tyto stavové bity:

- C (Carry) – pokud je nutné vzít bit 7 z vyššího bytu (výsledek odečítání je záporný), pak bude mít tento bit hodnotu 1, v opačném případě 0
- AC (Auxiliary Carry) – tento bit je analogický bitu Carry, jen indikuje přenos bitu z horních čtyř bitů do spodních čtyř bitů (hodnota akumulátoru se změní z větší nebo rovné 16 na menší než 16)
- OV (Overflow) – tento bit je nastaven na 1 pokud dojde k přenosu z bitu 6 ale zároveň nedojde k přenosu z bitu 7 nebo pokud dojde k přenosu z bitu 7 a zároveň nedojde k přenosu z bitu 6. V ostatních případech je tento bit nastaven na 0. Při odečítání dvou znaménkových hodnot indikuje tento bit vznik pozitivního čísla ze dvou negativních nebo negativního čísla ze dvou pozitivních

Instrukce	Příklad
SUBB A,Rn	SUBB A,R0 – odečte od hodnoty akumulátoru hodnotu z registru R0 a hodnoty bitu Carry, n=0..7
SUBB A,<adresa>	SUBB A,10 – odečte od hodnoty akumulátoru hodnotu z paměťové buňky, která je na adrese 10 a hodnoty bitu Carry
SUBB A,@Ri	SUBB A,@R1 – odečte od hodnoty akumulátoru hodnotu z paměťové buňky, která je na adrese uložené v registru R1 a hodnoty bitu Carry
SUBB A,#<data>	SUBB A,#20 – odečte od hodnoty akumulátoru hodnotu 20 a hodnoty bitu Carry

DA A

Dekadická korekce akumulátoru po sčítání – tato instrukce upravuje výsledek sčítání dvou čísel (která byla původně v BCD kódu) zpět do BCD kódu. Pokud hodnota spodních čtyř bitů bude větší než 9, nebo pokud bit AC bude mít hodnotu 1, pak přičte se k hodnotě akumulátoru hodnota 6 aby došlo k opravě BCD kódu ve spodní polovině bytu. Pokud tímto přičtením dojde k přenosu i přes vyšší polovinu bytu až do vyššího řádu, nastaví se bit Carry na hodnotu 1. V ostatních případech zůstává jeho hodnota nezměněna. Další krok se provede pokud je bit Carry nastaven na 1 nebo pokud je horní polovina bytu větší, než 9. Pak je i k horní polovině bytu přičtena hodnota 6 aby došlo k jeho korekci. Také při této operaci dojde k nastavení bitu Carry na 1 pouze pokud dojde k přenosu do vyššího bytu, v opačném případě zůstává jeho hodnota nezměněna. Žádný jiný stavový bit není ovlivněn.
Tato instrukce ale nemůže být použita pro převod čísla do BCD kódu, ani pro korekci po odečítání.

Příklad:
Provedeme sečtení čísel 15 a 26 v BCD kódu pomocí instrukce ADD a poté korekci akumulátoru na BCD kód pomocí instrukce DA

```
MOV  A,#15
ADD  A,#26
```

Číslo 15 v BCD kódu	0 0 0 1 0 1 0 1
Číslo 26 v BCD kódu	+ 0 0 1 0 0 1 1 0
Přenos	. 1
Výsledek	0 0 1 1 1 0 1 1

Hodnoty stavových bitů na konci výpočtu:
C=0 – nedošlo k přenosu ze sedmého bitu
AC=0 – nedošlo k přenosu ze třetího bitu
OV=0 – nedošlo k přenosu ze šestého ani sedmého bitu

DA A

Protože hodnota spodní poloviny bytu je 11, ke spodní polovině bytu přičteme hodnotu 6.

Akumulátor	0 0 1 1 1 0 1 1
Číslo 6	+ 0 0 0 0 0 1 1 0
Přenos	1 1 1 1 1
Výsledek	0 1 0 0 0 0 0 1

Výsledkem je správná hodnota součtu čísel v BCD kódu. Stavové bity v tomto případě zůstávají na stejné hodnotě, jako po součtu.

Násobení a dělení

MUL AB

Násobení dvou neznaménkových osmibitových čísel. Jedno je uloženo v akumulátoru a druhé v registru B. Výsledek je šestnáctibitová hodnota, která je opět uložena do akumulátoru a registru B a to tak, že spodní polovina výsledku je v akumulátoru a horní polovina v registru B. Tato instrukce ovlivňuje tyto stavové bity:

- C (Carry) – bit Carry je nastaven na 0
- OV (Overflow) – tento bit je nastaven na 1 pokud výsledek násobení je větší než 255, v opačném případě je vynulován

DIV AB

Tato instrukce provede celočíselné dělení neznaménkového osmibitového čísla z akumulátoru číslem, které je uloženo v registru B. Výsledek dělení je v akumulátoru, celočíselný zbytek po dělení je uložen do registru B. V případě dělení nulou je obsah obou registrů nedefinovaný. Tato instrukce ovlivňuje tyto stavové bity:

- C (Carry) – bit Carry je nastaven na 0
- OV (Overflow) – bit Overflow je také nastaven na 0, jen v případě dělení nulou je nastaven na 1

Logické instrukce

Logické instrukce provádějí základní logické operace pro jednotlivé bity zvolených registrů a neovlivňují žádné stavové bity. Pravdivostní tabulky:

Zdroj	Cíl	ANL	ORL	XRL
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Cíl	CPL
0	1
1	0

ANL <cíl>,<zdroj>

Instrukce ANL provede bitově logický součin zdroje a cíle, výsledek uloží do registru, který je zvolen jako cíl.

Seznam možných kombinací operandů pro osmibitové operace:

Instrukce	Příklad
ANL A,Rn	ANL A,R0 – provede logický součin akumulátoru s registrem R1, výsledek uloží do akumulátoru, n=0..7
ANL A,<adresa>	ANL A,15 – provede logický součin akumulátoru s obsah paměťové buňky, která je na adrese 15, výsledek uloží do akumulátoru
ANL A,@Ri	ANL A,@R1 – provede logický součin akumulátoru s paměťovou buňkou, která je na adrese, uložené v registru R1, výsledek uloží do akumulátoru, i=0..1
ANL A,#<data>	ANL A,#11 – provede logický součin akumulátoru s hodnotou 11, výsledek uloží do akumulátoru
ANL <adresa>,A	ANL 10,A – provede logický součin akumulátoru s hodnotou, která je v paměťové buňce na adrese 10, výsledek uloží do paměťové buňky na adrese 10
ANL <adresa>,#<data>	ANL 10,#20 – provede logický součin hodnoty, která je v paměťové buňce na adrese 10 s hodnotou 20, výsledek uloží do paměťové buňky na adrese 10

Seznam možných kombinací operandů pro jednobitové přesuny:

Instrukce	Příklad
ANL C,<bit>	ANL C,P1.0 – do bitového registru Carry zkopíruje výsledek logického součinu hodnoty ze vstupu P.0 a bitu Carry
ANL C,/<bit>	ANL C,/P1.0 – do bitového registru Carry zkopíruje výsledek logického součinu negované hodnoty ze vstupu P.0 a bitu Carry

ORL <cíl>,<zdroj>
XRL <cíl>,<zdroj>

Instrukce ORL a XRL pracují dle příslušné pravdivostní tabulky naprosto stejně, jako instrukce ANL. Mají i stejné povolené kombinace operandů.

CPL <cíl>

Seznam možných kombinací operandů pro osmibitové operace:

Instrukce	Příklad
CPL A	CPL A – provede negaci všech bitů akumulátoru

Seznam možných kombinací operandů pro jednobitové přesuny:

Instrukce	Příklad
CPL C	CPL C – provede negaci bitu Carry
CPL <bit>	CPL P2.1 – provede negaci bitu P2.1

Instrukce rotací

Všechny instrukce rotací pracují pouze s akumulátorem. Některé instrukce pracují s bitem Carry tak, jako by to byl devátý bit akumulátoru. Ostatní stavové bity zůstávají nezměněné.

RL A – rotace vlevo
RLC A – vlevo přes carry
RR A – rotace vpravo
RRC A – vpravo přes carry
SWAP A – výměna spodní a horní poloviny akumulátoru, odpovídá čtyřem operacím rotace vlevo nebo vpravo

Bitové operace

Pro manipulaci s bity lze používat instrukce přesunu dat nebo logické instrukce, viz výše. Speciálně pro nastavení jednoho bitu lze použít instrukce CLR a SETB.

Instrukce	Příklad
CLR C	CLR C – do bitového registru Carry vloží hodnotu 0
CLR <bit>	CLR P2.1 – na výstup P2.1 nastaví hodnotu 0

Instrukce	Příklad
SETB C	SETB C – do bitového registru Carry vloží hodnotu 1
SETB <bit>	SETB P2.1 – na výstup P2.1 nastaví hodnotu 1

Nepodmíněné skoky

Nepodmíněné skoky se dělí podle způsobu zadání adresy. Některé způsoby adresování omezují vzdálenost skoku nebo oblast, do které může adresa směřovat. Proto určení konkrétní instrukce je ve většině případů lepší ponechat na překladači a použít mnemotechnickou zkratku JMP. Místo ní pak překladač použije jednu z následujících instrukcí:

AJMP <addr11> - adresa je určena pomocí 11ti bitů, horních 5 bitů je použito z registru PC. Proto je možné provádět skoky pouze v bloku o velikosti 2kB, ve které se nachází tato instrukce, v paměti zabírá 2 byty
LJMP <addr16> - tato instrukce umožňuje provádět skoky v rámci celé programové paměti, ale v paměti zabírá 3 byty
SJMP <rel.adr.> - relativní skok, adresa je určena jedním bytem jako znaménková hodnota od -128 do 127 bytů. V paměti také zabírá 2 byty

Adresu lze určit přímo číselnou konstantou:

```
LJMP 1050   - skok na instrukci, která začíná v paměti na adrese 1050
SJMP 10     - skok na instrukci, která je v paměti umístěna o 10 bytů za instrukcí SJMP
```

nebo pomocí návěstí, které je definováno někde v programu:

```
START:
    JMP   START
```

V takovém případě překladač sám vyčíslí adresu návěstí, určí nejvhodnější instrukci a provede zakódování adresy do příslušného formátu.

Podprogramy

Hlavní program obvykle obsahuje i některé krátké rutiny, které se často opakují. Jejich vyčlenění do podprogramu nám pomůže jednak ušetřit programovou paměť, ale také to zpřehledňuje strukturu programu a zjednodušuje jeho údržbu. Podprogramy lze volat z různých míst hlavního programu nebo z jiných podprogramů. Po ukončení podprogramu pak program pokračuje tam, odkud byl podprogram vyvolán.

Pro volání podprogramů platí obdobná pravidla, jako pro nepodmíněné skoky. Existují dvě možnosti zadání adresy, ale pokud použijeme mnemotechnickou zkratku CALL, pak překladač vybere optimální variantu. Oproti prostému skoku navíc ještě uloží do zásobníku návratovou adresu – aktuální obsah registru PC inkrementovaný o délku instrukce CALL (tzn. ukládá se vlastně adresa následující instrukce). Jako první se uloží spodní byte adresy, poté horní byte adresy.

```
ACALL <addr11>   - adresa je určena pomocí 11ti bitů, horních 5 bitů je použito z registru PC. Proto je
                   možné provádět skoky pouze v bloku o velikosti 2kB, ve které se nachází tato
                   instrukce, v paměti zabírá 2 byty
LCALL <addr16>   - tato instrukce volat podprogramy v rámci celé programové paměti, ale v
                   paměti zabírá 3 byty
```

Pro návrat z podprogramu slouží instrukce RET. Tato instrukce nemá žádné operandy. Nejprve vyzvedne ze zásobníku návratovou adresu a poté provede skok na tuto adresu.

Příklad podprogramu:

```
ROZSVIT:
    CLR   P1.0          ; Rozsviti LED
    RET

ZHASNI:
    SETB P1.0           ; Zhasne LED
    RET

START:
    JB    P1.1,$        ;Kdyz je stisknute tlacitko 1
    CALL  ROZSVIT       ; tak rozsvit
    JB    P1.2,$        ; kdyz je stisknute tlacitko 2
    CALL  ZHASNI        ; tak zhasni
    JMP   START
```

Instrukce RETI slouží pro návrat z přerušovací rutiny. Pracuje stejným způsobem, jako

instrukce RET. Navíc pouze povoluje vyvolání přerušení se stejnou, nebo nižší prioritou.

Podmíněné skoky

Podmíněné skoky slouží pro větvení programu na základě splnění některé z podmínek (např. změna hodnoty na pinu nebo výsledek aritmetické operace). Adresa cíle pro skok je vždy uložená jako relativní, proto je možné tyto instrukce použít pouze pro skoky na omezenou vzdálenost.

Instrukce	Příklad
JB <bit>,<rel.adr.>	JB P1.2,STISK – provede skok na návěstí STISK pokud má vstupní pin P1.2 hodnotu 1
JBC <bit>,<rel.adr.>	JBC P1.2,STISK – provede skok na návěstí STISK pokud má vstupní pin P1.2 hodnotu 1 a zároveň nastaví tento bit na hodnotu 0
JNB <bit>,<rel.adr.>	JNB P1.2,STISK – provede skok na návěstí STISK pokud má vstupní pin P1.2 hodnotu 0
JC <rel.adr.>	JC STISK – provede skok na návěstí STISK pokud má bit Carry hodnotu 1
JNC <rel.adr.>	JNC STISK – provede skok na návěstí STISK pokud má bit Carry hodnotu 0
JZ <rel.adr.>	JZ STISK – provede skok na návěstí STISK pokud je v akumulátoru hodnota 0
JNZ <rel.adr.>	JNZ STISK – provede skok na návěstí STISK pokud je v akumulátoru hodnota jiná, než 0

Sdružené instrukce

CJNE <cíl>,<zdroj>,<rel.adr.>

Compare And Jump If Not Equal - Tato instrukce provede porovnání dvou čísel a v případě nerovnosti provede skok na zadanou adresu. Tato adresa je zadána osmibitově jako relativní, takže je možné provádět skoky na omezenou vzdálenost. Tato instrukce také nastaví bit Carry pokud je zdroj větší, než cíl. V opačném případě je bit Carry vynulován. Ostatní stavové bity jsou neovlivněny.

Instrukce	Příklad
CJNE A,<adresa>,<rel.adr.>	CJNE A,10,LOOP – porovná akumulátor s hodnotou paměťové buňky na adrese 10 a v případě nerovnosti provede skok
CJNE A,#<data>,<rel.adr.>	CJNE A,#15,LOOP – porovná akumulátor s hodnotou 15 a v případě nerovnosti provede skok
CJNE Rn,#<data>,<rel.adr.>	CJNE R1,#15,LOOP – porovná obsah registru R1 s hodnotou 15 a v případě nerovnosti provede skok, n=0..7
CJNE @Ri,#<data>,<rel.adr.>	CJNE @R0,#15,LOOP – porovná obsah paměťové buňky, která je na adrese uložené v registru R0, s hodnotou 15 a v případě nerovnosti provede skok, i=0..1

DJNZ <registr>,<rel.adr.>

Decrement and Jump if Not Zero – Tato instrukce nejprve dekrementuje zadaný registr a poté provede porovnání. Pokud je v registru nenulová hodnota, pak provede skok na zadanou adresu. Také u této instrukce je cílová adresa zadána osmibitově jako relativní. Instrukce DJNZ neovlivňuje žádný stavový bit.

Instrukce	Příklad
DJNZ Rn,<rel.adr.>	DJNZ R0,LOOP
DJNZ <adresa>,<rel.adr.>	DJNZ 10,LOOP