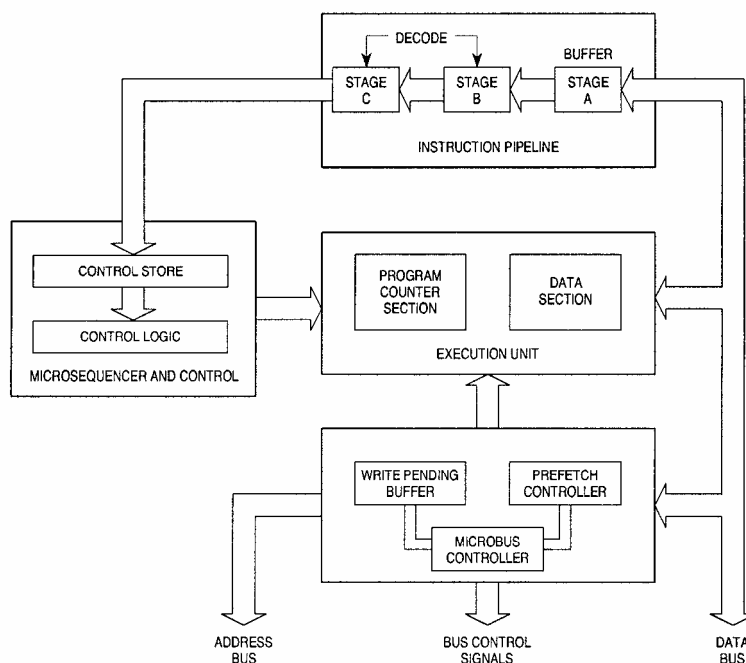


4. CENTRALNA PROCESNA ENOTA CPU32

4.1 Zgradba centralne procesne enote CPU32 mikrokrmilnika MC68332

V tem delu bomo opisali arhitekturo in nabor ukazov 32-bitne centralne procesne enote mikrokrmilnika MC68332. Ta je v mnogočem podobna ostalim CPU, ki so prikazani v poglavju 11.

Centralna procesna enota CPU32 je nastala iz mikroprocesorja MC68020 in je osnovni del vseh mikrokrmilnikov iz serije MC683xx. Njena naloga je izvajanje uporabniškega programa ter koordiniranje delovanja ostalih podmodulov. Navzdol je združljiva z družino procesorjev MC68000. CPU32 lahko programiramo v višjih programskih jezikih HLL (High Level Languages), omogoča različne načine naslavljanja, poleg tega pa vsebuje še nekaj novih ukazov v primerjavi z MC68020. To so predvsem izboljšane (hitrejše) funkcije množenja, deljenja in pomika. Podatkovni registri podpirajo operacije z 8-, 16- ali 32-bitnimi operandi. V tem poglavju se ne bomo ukvarjali z interno arhitekturo CPU (slika 4. 1), saj nas s stališča uporabnika zanimajo le segmenti, ki so nam dostopni pri pisanju programov.



Slika 4. 1: Blokovni diagram CPU32

Za posredovanje informacij med uporabniškim programom in CPU so zadolženi *CPU registri*. Registri so del reduciranega področja hitrega RAM pomnilnika v CPU. Za razliko od navadnega RAM imajo centralne procesne enote le nekaj registrov (sodobnejši običajno 32, nekateri, npr. SPARC, celo 144). Zaradi tega je del kode, ki je potrebna za naslavljanje registrov, sestavljen samo iz nekaj bitov, za naslavljanje pomnilnika pa jih rabimo bistveno več, npr. 24. Čas dostopa do registrov je bistveno hitrejši od časa dostopa do pomnilnika. V

CPU imajo registri imena¹, ki jih uporabljamo za naslavljanje pri programiranju (npr. D0, A7, SP itd.).

CPU32 vsebuje dva skupka internih registrov:

- **registri uporabniškega modela** - (User Programming Model)
- **registri nadzornega modela** - (Supervisor Programming Model)

4.1.1 Registri v uporabniškem modelu CPU

Pri običajnem programiranju praviloma uporabljamo le prvo skupino registrov. Uporabniški registri so (slika 4. 2):

1. **Podatkovni registri** (D0 - D7),
2. **Naslovni registri** (A0 - A7),
3. **Kazalec sklada** (A7 ali USP),
4. **Programski števec** (PC),
5. **Register pogojnih kod** (Condition Code Register) - CCR.

31	23	15	7	
			0	
				D0
				D1
				D2
				D3
				D4
				D5
				D6
				D7
				A0
				A1
				A2
				A3
				A4
				A5
				A6
				A7 (USP)
				PC
		0		CCR

Slika 4. 2: Uporabniški registri CPU32

¹ Registri so praviloma ločeni od pomnilnika, pri nekaterih procesorjih (npr. Texas TMS99XX) pa so locirani v RAM pomnilniku. V MC68332 so nekatere lokacije v RAM pomnilniku vnaprej rezervirane za določanje načina obratovanja in parametriranje podmodulov (npr. MCR je "pseudoregister" za splošno določanje delovanja SIM modula in se nahaja na pomnilniški lokaciji fffa00_{HEX} ali 7ffa00_{HEX}). Takšne registre naslavljamo kot navadne pomnilniške lokacije.

4.1.1.1 Podatkovni registri

Podatkovni registri (Data Registers) - DR se uporabljajo za delo s podatki (npr. za shranjevanje enega ali obeh operandov pri aritmetičnih ali logičnih operacijah ter rezultata te operacije). Naslavljamo jih s simbolom Dx, kjer je x zaporedna številka registra (0-7). Skupaj imamo na voljo osem 32-bitnih registrov. Zato CPU32 štejemo med 32-bitne procesorje. Podatkovni registri so analogni akumulatorjem pri nekaterih drugih CPU (npr. MC68HC11 vsebuje 8-bitna akumulatorja A in B, glej pogl. 11).

Večje število podatkovnih registrov (ali akumulatorjev) je zaželeno zaradi hitrejšega izvajanja programa, ker omogoča shranjevanje vmesnih rezultatov operacij kar v CPU. Na ta način se izognemo shranjevanju in ponovnemu branju podatkov v/iz zunanega pomnilnika. Delo s pomnilnikom je praviloma počasnejše med ostalim tudi zato, ker ima zunanje podatkovno vodilo običajno manj podatkovnih linij od internega vodila v CPU. Npr. pri MC68332 imamo opravka s 16-bitnim zunanjim in 32-bitnim notranjim vodilom.

4.1.1.2 Naslovni registri

32-bitne *naslovne registre (Address Registers) - AR* uporabljamo za naslavljanje znotraj dovoljenega pomnilniškega področja. O njihovi funkciji bomo več govorili v poglavju o načinih naslavljanja (4.3).

4.1.1.3 Kazalec sklada

Uporabniški kazalec sklada (angl. User Stack Pointer) - SP ali *USP* je 32-bitni register, njegova vsebina pa je naslov začetka (vrh) uporabniškega *sklada* (angl. User Stack). Način dela s skladom opisuje poglavje 4.2.7.2.

Kazalec sklada je pri večini procesorjev samostojni register, v CPU32 pa je to kar naslovni register A7. Kadar ni v funkciji naslavljanja sklada je enakovreden ostalim naslovnim registrom. Zaradi te dvojne narave moramo biti pri pisanju programov zelo pozorni, da področje sklada ne poseže v del pomnilnika, kjer so shranjeni podatki in program.

4.1.1.4 Programski števec

Programski števec (angl. Program Counter) - PC je 32-bitni register, ki kaže na naslov naslednje ukaza za izvajanje. Dolžina ukaza je odvisna od tipa (dolžine) in števila operandov ter načinov naslavljanja. Zato mora pri linearnem poteku programa *dekoder inštrukcij* CPU skrbeti za ustrezno povečevanje vsebine PC. Pri skokih znotraj programa oz. na podprograme moramo vsebino PC spremeniti tako, da kaže na izbrano skočno lokacijo, kar dosežemo s posebnimi ukazi (glej tudi pogl. 4.2.7).

4.1.1.5 Register pogojnih kod

Ta register je v bistvu spodnja polovica statusnega registra (glej naslednje podpoglavje).

4.1.2 Registri v nadzornem modelu CPU (Supervisor Programming Model)

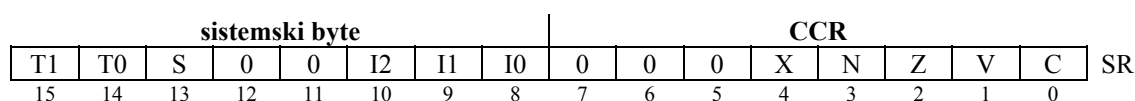
Nadzorne registre pri normalnem režimu izvajanja programa lahko le beremo. Spreminjamo jih lahko le v nadzornem režimu obratovanja (ob setirani zastavici S v statusnem registru). Ti registri so:

- nadzorni kazalec sklada,
- alternativna funkcijska kodna registra,
- register baze vektorjev,
- statusni register.

V tem poglavju bomo opisali le funkciji zadnjih dveh.

Register baze vektorjev (Vector Base Register) - VBR je register, katerega vsebina določa začetni naslov pomnilniškega prostora, v katerem se nahajajo prekinitveni vektorji (več o tem v poglavju 4.2.8.1).

Statusni register (Status Register) - SR je edini 16-bitni register. Njegova funkcija je spremljanje rezultata operacij v procesorju in postavljanje temu ustreznih posebnih bitov. Zgornjih osem bitov je rezerviranih za t.i. sistemski byte, spodnjih osem bitov pa zaseda register pogojnih kod (Condition Code Register) - CCR.



Oznake pomenijo:

T1, T0	...	sledenje (angl. trace) programa omogočeno/onemogočeno
S	...	nadzorni ("1") / uporabniški ("0") režim
I2, I1, I0	...	prekinitvena maska (nivoji dopustnih prekinitiev - pogl. 4.2.8.1.1)
X	...	razširitveni (angl. extend) bit
N	...	rezultat negativen
Z	...	rezultat nič
V	...	bit presežka (angl. overflow)
C	...	bit prenosa (angl. carry).

Stanje bitov X, N, Z, V, C je odvisno od rezultata večine ukazov pri vseh mikroprocesorjih, zato si podrobneje oglejmo njihov pomen.

Bit Z se načeloma postavi, če je rezultat operacije med preznačenimi števili enak nič, če pa je manjši on nič, se postavi bit N. Razširitveni bit X je, podobno kot bit C², pomemben pri seštevanju operandov, katerih format je večji od osnovne 32-bitne besede³. Oba bita sta zelo pomembna tudi pri operacijah pomika (angl. shift). Bit presežka (V) se postavi, ko je rezultat

² Pri nekaterih procesorjih imamo le enega.

³ Če hočemo npr. z 32-bitnimi registri zapisati in sešteti dve 64-bitni števili, moramo pri seštevanju zgornjih 32 bitov upoštevati morebiten bit prenosa kot posledico seštevanja spodnjih 32 bitov. Primer za to je ukaz `ADDX`.

operacije nemogoče izpisati v obsegu operandov. Pri seštevanju (ukaz `ADD`) je npr. bit `V` posledica operacije

$$V = S_m \bullet D_m \bullet R_m + \underline{S_m} \bullet \underline{D_m} \bullet R_m,$$

kjer so S_m , D_m in R_m najbolj pomembni (MSB) biti prvega in drugega operanda ter rezultata⁴. Vidimo, da se bit setira, kadar sta MSB operandov enaka, MSB rezultata pa ima nasprotno vrednost. Pri seštevanju je to indikacija napake, saj pri logiki dvojiškega komplementa seštevek dveh negativnih števil ($S_m = D_m = 1$) ne more biti pozitiven ($R_m = 0$). Tak rezultat je očitno nemogoče zapisati v formatu obeh operandov.

V splošnem velja, da moramo pravila za setiranje omenjenih statusnih bitov preučiti za vsak ukaz posebej!

4.2 Nabor ukazov za CPU32

V tem poglavju si bomo ogledali nabor ukazov za CPU32. Čeprav ima vsak CPU svoj lastni nabor ukazov, je velika večina ukazov skupna vsem procesorjem. Sintakse so si tudi zelo podobne, kar omogoča relativno hiter prehod z enega na drug procesor.

V splošnem je ukaz sestavljen iz

- neobvezne *simbolične oznake naslova ukaza* (npr. `lok`),
- *operacije* (npr. `ADD`),
- *tipa oz. širine operandov* (`B` - byte, `W` - beseda, `L` - dvojna beseda),
- *prvega operanda* (npr. `D0`),
- *drugega operanda* (npr. `D1`) - pri večini ukazov vsebuje tudi rezultat operacije,
- *komentarja* (za zvezdico `*`):

```
lok ADD.L D0,D1      *seštevanje dveh 32-bitnih števil (long)
                      iz
                      *D0 in D1; rezultat se nahaja v D1 -
                      *prejšnja vsebina D1 je izgubljena
```

Poleg teh obstajajo tudi ukazi z enim operandom (npr. `CLR.L D0`) in ukazi brez operanda (npr. `NOP`). Sintaksa ukazov, ki so opisani v tem poglavju, velja za najnižji nivo vpisa ukazov, to je v *zbirniku* (angl. *assembler*).

V naslednjih podpoglavjih so ukazi glede na njihovo funkcijo predstavljeni v skupinah :

- **ukazi za prenos podatkov,**
- **ukazi za celoštevilčno (integer) aritmetiko,**
- **logične operacije,**
- **operacije pomika in rotacije,**
- **operacije za manipulacije z biti,**

⁴ Operator “•” pomeni logično konjunkcijo (logična funkcija `IN` - angl. `AND`), “+” pa logično disjunkcijo (funkcija `ALI` - angl. `OR`). V nadaljevanju bomo negacijo bita, celotnega tipa ali signala označevali s podčrtajem (npr. `CS`).

- operacije med BCD števili,
- ukazi za nadzor nad potekom izvajanja programa,
- ukazi za sistemski nadzor.

V tabelah se uporabljajo številne okrajšave, ki ponazarjajo načine naslavljanja ter funkcije posameznih ukazov:

data	podatek, ki je vsebovan v samem ukazu
Dest	(angl. destination) operand in končni rezultat (cilj)
Source	operand (začetna lokacija) ⁵
vektor	lokacija prekinitvenega vektorja (glej podpogl. 4.2.8.1.1)
An	katerikoli naslovni register (A0 - A7)
Ax, Ay	naslovna registra, ki ju uporabljamo v ukazu
Dn	katerikoli podatkovni register (D0 - D7)
Rc	nadzorni register (VBR, SFC, DFC)
Rn	katerikoli podatkovni ali naslovni register
Dh, Dl	podatkovna registra, ki vsebujeta višjo in nižjo polovico 64-bitnega rezultata množenja
Dr, Dq deljenju	podatkovna registra, ki vsebujeta ostanek in količnik pri celoštevilčnem deljenju
Dx, Dy	podatkovna registra, ki ju uporabljamo pri izračunu v ukazu
Dym, Dyn	podatkovna registra, vrednosti za tabelarično interpolacijo
Xn	indeksni register
[An]	podaljšek naslova
cc	pogojna koda (pri pogojnih operacijah)
d#	odmik pri naslavljanju (angl. displacement), npr. d ₁₆ je 16-bitni odmik
<ea>	efektivni naslov (načeloma pomeni poljubni način naslavljanja; za natančnejše informacije glej [25])
#<data>	takojšnji celoštevilčni podatek
oznaka	oznaka (simbolični naslov) v zbirniku (angl. label)
list	lista, seznam registrov, npr. D3-D0
[...]	biti operanda, npr. [7] označuje 7. bit, [32:24] pa bite od 32 do 24
(...)	vsebina lokacije, npr. (Rn) označuje vsebino registra Rn
CCR	register za pogojno kodo (angl. condition code register) - spodnji byte statusnega registra (sestavljen iz bitov X, N, Z, V, C; glej pogl. 4.1.2)
PC	programski števec
SP	aktivni kazalec sklada
SR	statusni register
SSP	kazalec sklada v nadzornem režimu
USP	uporabniški kazalec sklada
FC	funkcijska koda
DFC	register za funkcijsko kodo končnega naslova
SFC	register za funkcijsko kodo začetnega naslova

⁵ Primer: pri ukazu `ADD.L D0, D1` je D0 Source, D1 pa Dest. To pomeni, da bo seštevek vsebin obeh registrov shranjen v registru D1, vsebina D0 pa ostane nespremenjena.

- Booleova logična funkcija IN
- + Booleova logična funkcija ALI
- ⊕ Booleova logična funkcija EKSKLUZIVNI ALI
- \bar{x} Booleov logični komplement (invertiranje operanda)
- LSW, MSW manj pomembna (desna) in bolj pomembna (leva) beseda

V tabelah ukazov vsebuje prvi stolpec t.i. *mnemonik* ukaza⁶, drugi splošno sintakso oz. dovoljene tipe operanda (ali operandov). Tretji stolpec deklarira možne tipe oz. dolžine operandov (B, W, L: 8, 16 ali 32 bitov), četrti pa opisuje funkcijo ukaza.

OPOZORILO!

Načeloma velja, da nadomešča simbol <ea> katerikoli tip naslova: Dx, Ax, #xxx itd. Za omejitve pri posameznih ukazih glej literaturo [25].

Primeri ukazov so pisani v debug/monitorskem programu CPU32BUG, kjer so z “%” označena binarna števila, z “&” desetiška, operandi s simbolom “\$” oz. brez kakršnekoli oznake pa so pisani v šestnajstiški kodi. Tukaj velja še povedati, da se pri nekaterih drugih zbirnikih sintaksa nekoliko razlikuje od tukaj prikazane.

4.2.1 Ukazi za prenos podatkov

Ukaz	Sintaksa	Dolžina operanda	Operacija
EXG	Rn, Rn	32	Rn \Rightarrow Rn
LEA	<ea>, An	32	<ea> \Rightarrow An
LINK	An, #<d>	16, 32	SP-4 \Rightarrow SP, An \Rightarrow (SP); SP \Rightarrow An, SP+d \Rightarrow SP
MOVE	<ea>, <ea>	8, 16, 32	Source \Rightarrow Dest
MOVEA	<ea>, An	16, 32 \Rightarrow 32	Source \Rightarrow Dest
MOVEM	list, <ea> <ea>, list	16, 32 16, 32 \Rightarrow 32	našteti registri \Rightarrow Dest Source \Rightarrow našteti registri
MOVEP	Dn, (d16, An) (d16, An), Dn	16, 32	Dn[31:24] \Rightarrow (An+d); Dn[23:16] \Rightarrow (An+d+2); Dn[15:8] \Rightarrow (An+d+4); Dn[7:0] \Rightarrow (An+d+6) (An+d) \Rightarrow Dn[31:24]; (An+d+2) \Rightarrow Dn[23:16]; (An+d+4) \Rightarrow Dn[15:8]; (An+d+6) \Rightarrow Dn[7:0]
MOVEQ	#<data>, Dn	8 \Rightarrow 32	takojšnji podatek \Rightarrow Dest
PEA	<ea>	32	SP-4 \Rightarrow SP; <ea> \Rightarrow SP
UNLK	An	32	An \Rightarrow SP; (SP) \Rightarrow An, SP+4 \Rightarrow SP

Osnovna funkcija te skupine ukazov je prenos in shranjevanje podatkov in naslovov (npr. iz registra v register ali pomnilnik).

⁶ Izraz “mnemonik” ali “mnemotehnični” pomeni, da nas že naziv ukaza spominja na njegovo funkcijo, npr. ukaz CLR (iz angleške besede “clear” - izbrisati) postavi vse bite operanda v stanje “0”.

Primeri ukazov:

- `MOVEA.L #12345678, A3`

32-bitno konstanto 12345678_{HEX} naloži v A3.

- `MOVE.B D0, D2`

pred izvršitvijo: (D0)=12345678_{HEX} in (D2)=87654321_{HEX},

po izvršitvi: (D0) = 12345678_{HEX} in (D2)=87654378_{HEX}.

PAZI: le spodnji byte ciljne lokacije se zamenja s spodnjim bytom začetnega naslova!

4.2.2 Ukazi za celoštevilčno (integer) aritmetiko

Ukaz	Sintaksa	Dolžina operanda	Operacija
ADD	$Dn, <ea>$ $<ea>, Dn$	8, 16, 32 8, 16, 32	Source+Dest \Rightarrow Dest
ADDA	$<ea>, An$	16, 32	Source+Dest \Rightarrow Dest
ADDI	$\#<data>, <ea>$	8, 16, 32	takojšnji podatek+Dest \Rightarrow Dest
ADDQ	$\#<data>, <ea>$	8, 16, 32	takojšnji podatek+Dest \Rightarrow Dest
ADDX	Dn, Dn $-(An), -(An)$	8, 16, 32 8, 16, 32	Source+Dest+X \Rightarrow Dest
CLR	$<ea>$	8, 16, 32	0 \Rightarrow Dest
CMP	$<ea>, Dn$	8, 16, 32	(Dest-Source), postavi CCR
CMPA	$<ea>, An$	16, 32	(Dest-Source), postavi CCR
CMPI	$\#<data>, <ea>$	8, 16, 32	(Dest-data), postavi CCR
CMPM	$(An)+, (An)+$	8, 16, 32	(Dest-Source), postavi CCR
CMP2	$<ea>, Rn$	8, 16, 32	Sp. meja $\leq Rn \leq$ Zg. meja, postavi CCR
DIVS/DIVU	$<ea>, Dn$	32/16 \Rightarrow 16:16	Dest/Source \Rightarrow Dest (z ali brez predznaka)
DIVSL/DIVUL	$<ea>, Dr:Dq$ $<ea>, Dq$ $<ea>, Dr:Dq$	64/32 \Rightarrow 32:32 32/32 \Rightarrow 16 32/32 \Rightarrow 32:32	Dest/Source \Rightarrow Dest (z ali brez predznaka)
EXT	Dn Dn	8 \Rightarrow 16 16 \Rightarrow 32	razširitev Dest s predznakom \Rightarrow Dest
EXTB	Dn	8 \Rightarrow 32	razširitev Dest s predznakom \Rightarrow Dest
MULS/MULU	$<ea>, Dn$ $<ea>, D1$ $<ea>, Dh:D1$	16*16 \Rightarrow 32 32*32 \Rightarrow 32 32*32 \Rightarrow 64	Dest*Source \Rightarrow Dest (z ali brez predznaka)
NEG	$<ea>$	8, 16, 32	0-Dest \Rightarrow Dest
NEGX	$<ea>$	8, 16, 32	0-Dest-X \Rightarrow Dest
SUB	$<ea>, Dn$ $Dn, <ea>$	8, 16, 32	Dest-Source \Rightarrow Dest
SUBA	$<ea>, An$	16, 32	Dest-Source \Rightarrow Dest
SUBI	$\#<data>, <ea>$	8, 16, 32	Dest-data \Rightarrow Dest
SUBQ	$\#<data>, <ea>$	8, 16, 32	Dest-data \Rightarrow Dest
SUBX	Dn, Dn $-(An), -(An)$	8, 16, 32 8, 16, 32	Dest-Source-X \Rightarrow Dest
TBLS/TBLU	$<ea>, Dn$ $Dym:Dyn, Dn$	8, 16, 32	Dyn-Dym \Rightarrow Temp (Temp*Dn[7:0]) \Rightarrow Temp (Dym*256)+Temp \Rightarrow Dn
TBLSN/TBLUN	$<ea>, Dn$ $Dym:Dyn, Dn$	8, 16, 32	Dyn-Dym \Rightarrow Temp (Temp*Dn[7:0])/256 \Rightarrow Temp Dym+Temp \Rightarrow Dn

Poleg enostavnejših aritmetičnih operacij z različnimi tipi operandov (seštevanje, odštevanje, brisanje, negiranje...), sodita v to skupino še ukaza za množenje in deljenje, ki običajno nista del nabora ukazov enostavnejših procesorjev.

Ukazi za primerjavo dveh operandov so v osnovu operacije odštevanja, pri čemer se oba operanda ne spremenita. Edina posledica ukaza je postavitve ustreznih bitov v CCR registru.

Primeri ukazov:

• NEG.L D0

pred ukazom: (D0) = 00001234_{HEX},

po ukazu: (D0) = ffffedcc_{HEX}

• ADD.B D3, D0

pred ukazom: (D3) = 12121212_{HEX}, (D0) = 12345678_{HEX},

po ukazu: (D3) = 12121212_{HEX}, (D0) = 1234568a_{HEX} (sodeluje le LSByte).

• SUBI.L #ffffff, D0

pred ukazom: (D0) = ffffffff_{HEX} = -1_{DEC},

po ukazu: (D0) = 0, zaradi: -1_{DEC} - (-1_{DEC}) = 0.

4.2.3 Logične operacije

Ukaz	Sintaksa	Dolžina operanda	Operacija
AND	<ea>, Dn Dn, <ea>	8, 16, 32 8, 16, 32	Source•Dest⇒Dest
ANDI	#<data>, <ea>	8, 16, 32	data•Dest⇒Dest
EOR	Dn, <ea>	8, 16, 32	Source⊕Dest⇒Dest
EORI	#<data>, <ea>	8, 16, 32	data⊕Dest⇒Dest
NOT	<ea>	8, 16, 32	<u>Dest</u> ⇒Dest
OR	<ea>, Dn Dn, <ea>	8, 16, 32 8, 16, 32	Source+Dest⇒Dest
ORI	#<data>, <ea>	8, 16, 32	data+Dest⇒Dest
TST	<ea>	8, 16, 32	Source-0; spremeni CCR

V tej skupini so ukazi za izvajanje logičnih operacij IN, ALI, EKSKLUZIVNI ALI in NE (angl. AND, OR, EOR in NOT) ter njihove inačice z neposrednim operandom. V operacijah se povezujejo biti na istih pozicijah (enake uteži) znotraj operanda.

Primeri ukazov:

- **AND.L (A0), D0**

pred ukazom:

- (A0) = 00008000_{HEX},
- dvojna beseda na naslovu (8000) je 87654321_{HEX} ter
- (D0) = ffffedcc_{HEX};

po ukazu:

- (A0) = 00008000_{HEX},
- (8000) = 87654321_{HEX} ter
- (D0) = 87654100_{HEX} (zaradi lažje ponazoritve pretvori v binarna števila).

- **EORI.W #1212, D0**


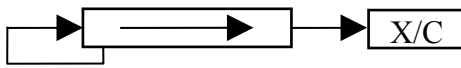

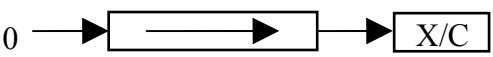
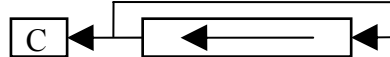
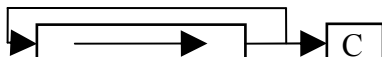
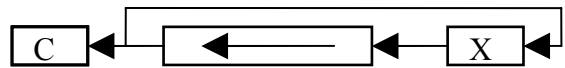
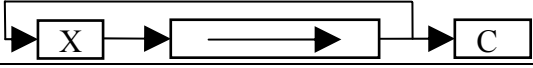
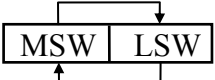
pred ukazom: (D0) = 33333333_{HEX},

po ukazu: (D0) = 33332121_{HEX} (sodeluje le LSW):

- **TST.W D0**

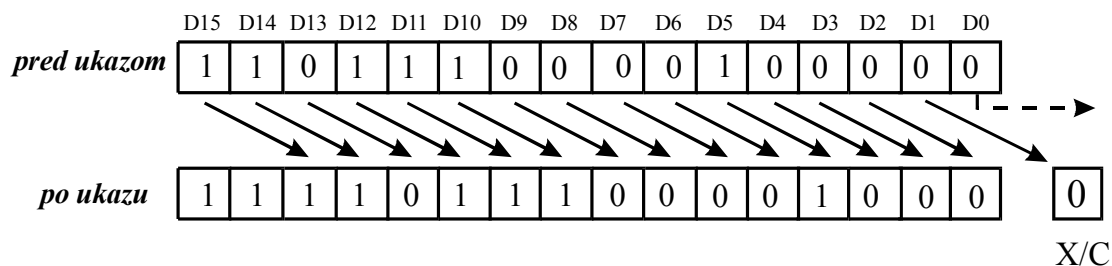
(D0) = 1234abcd_{HEX} pred in po izvršitvi ukaza; ukaz postavi bit N (rezultat negativen) v CCR (del statusnega registra), saj obravnava le LSW (abcd_{HEX}), ki je negativen po logiki dvojiškega komplementa:

4.2.4 Pomične in rotacijske operacije

Ukaz	Sintaksa	Dolžina operanda	Operacija
ASL	Dn, Dn #<data>, Dn <ea>	8, 16, 32 8, 16, 32 16	
ASR	Dn, Dn #<data>, Dn <ea>	8, 16, 32 8, 16, 32 16	
LSL	Dn, Dn #<data>, Dn <ea>	8, 16, 32 8, 16, 32 16	
LSR	Dn, Dn #<data>, Dn <ea>	8, 16, 32 8, 16, 32 16	
ROL	Dn, Dn #<data>, Dn <ea>	8, 16, 32 8, 16, 32 16	
ROR	Dn, Dn #<data>, Dn <ea>	8, 16, 32 8, 16, 32 16	
ROXL	Dn, Dn #<data>, Dn <ea>	8, 16, 32 8, 16, 32 16	
ROXR	Dn, Dn #<data>, Dn <ea>	8, 16, 32 8, 16, 32 16	
SWAP	Dn	16	

V tej skupini so ukazi za pomik (angl. shift) bitov znotraj operanda. Ukazi se razlikujejo glede na smer in število mest pomika, stanje “izpraznjenega” bita (MSB ali LSB) ter položaj ali indikacijo “izpadlega” bita.

Kot primer si lahko ogledamo ukaz `ASR.W #2, D0` (slika 4. 3). Pomik se nanaša le na spodnjih 16 bitov registra D0. Predpostavimo, da obdelujemo le 16-bitne informacije, torej je pred izvršitvijo ukaza vsebina spodnje besede v D0 po predznaku negativna: $1101\ 1100\ 0010\ 0000_{\text{BIN}} = \text{dc20}_{\text{HEX}} = -9184_{\text{DEC}}$ (MSW = 0). Ukaz premakne vse bite v registru za dve poziciji v desno, pri čemer se prazni poziciji na levi naložita z “1” (MSB), skrajnji desni biti pa zapustijo register (razen zadnjega, ki se naloži v X oz. C bit statusnega registra). Nova vsebina registra je $1111\ 0111\ 0000\ 1000_{\text{BIN}} = \text{f708}_{\text{HEX}} = -2296_{\text{DEC}}$ (ob upoštevanju le spodnje besede), torej štirikrat manjša od izhodiščne. Rezultat je pričakovan, saj je pomik bitov v levo (npr. z ukazom ASL) ali v desno za n pozicij enak množenju oz. deljenju z 2^n .

Slika 4. 3: Pomik bitov pri ukazu **ASR.W #2, D0**

Podoben rezultat ukaza ASR dosežemo tudi z ukazom LSR, obstaja pa ena bistvena razlika: pri LSR ukazu se pri vsakem pomiku za en bit MSB napolni vedno z "0", pri ASR pa se preslika prejšnji MSB. Slednji torej ohranja predznak originalnega števila in opravlja deljenja z 2, 4, 8 itd., LSR pa le premika bite, brez upoštevanja predznaka (od tod tudi naziv "logični premik v desno", Logical Shift Right - LSR).

Primeri ukazov:

Naslednja skupina ukazov obdeluje D0, katerega vsebina pred izvršitvijo je enaka $12345678_{\text{HEX}} = 0001\ 0010\ 0011\ 0100\ 0101\ 0110\ 0111\ 1000_{\text{BIN}}$. Komentar kaže spremenjena stanja D0:

- **ROL.B #1, D0**

$123456F0_{\text{HEX}} = 0001\ 0010\ 0011\ 0100\ 0101\ 0110\ 1111\ 0000_{\text{BIN}}$;

biti desnega byta se premaknejo za eno pozijo v levo, MSB pa postane LSB; ostali biti v D0 so nespremenjeni.

- **LSL.L #2, D0**

$48D15BC0_{\text{HEX}} = 0100\ 1000\ 1101\ 0001\ 0101\ 1011\ 1100\ 0000_{\text{BIN}}$;

celotna vsebina D0 se premakne v levo za dva bita; pri vsakem premiku se c LSB naloži 0.

- **SWAP D0**

$5BC048D1_{\text{HEX}} = 0101\ 1011\ 1100\ 0000\ 0010\ 1000\ 1101\ 0001_{\text{BIN}}$; LSW in MSW zamenjata mesti.

4.2.5 Operacije manipuliranja z biti

Ukaz	Sintaksa	Dolžina operanda	Operacija
BCHG	$Dn, \langle ea \rangle$ $\# \langle data \rangle, \langle ea \rangle$	8, 32	$(\langle \text{št. bita} \rangle \text{ Dest}) \Rightarrow Z \Rightarrow \text{bit v Dest}$
BCLR	$Dn, \langle ea \rangle$ $\# \langle data \rangle, \langle ea \rangle$	8, 32	$(\langle \text{št. bita} \rangle \text{ Dest}) \Rightarrow Z$ $0 \Rightarrow \text{bit v Dest}$
BSET	$Dn, \langle ea \rangle$ $\# \langle data \rangle, \langle ea \rangle$	8, 32	$(\langle \text{št. bita} \rangle \text{ Dest}) \Rightarrow Z$ $1 \Rightarrow \text{bit v Dest}$
BTST	$Dn, \langle ea \rangle$ $\# \langle data \rangle, \langle ea \rangle$	8, 32	$(\langle \text{št. bita} \rangle \text{ Dest}) \Rightarrow Z$

Naloga te skupine ukazov je spreminjanje in testiranje posameznih bitov znotraj byta ali dolge besede.

Primeri ukazov:

Vsebina D0 je pred izvajanjem zaporedja ukazov:

$1111\ 1111_{\text{HEX}} = 0001\ 0001\ 0001\ 0001\ 0001\ 0001\ 0001\ 0001_{\text{BIN}}$.

- **BCHG.L #3,D0**

stanje bita 3 v D0 (četrtga z desne strani) je 0, zato setiraj Z bit CCR, nato pa isti bit v D0 negiraj; rezultat je $1111\ 1119_{\text{HEX}}$.

- **BCLR.L #3,D0**

isti bit najprej primerjaj z "0", ker pa je "1", resetiraj Z v CCR; nato briši bit v D0; rezultat je $1111\ 1111_{\text{HEX}}$.

4.2.6 Operacije z BCD števili

Ukaz	Sintaksa	Dolžina operanda	Operacija
ABCD	$Dn, Dn - (An), -(An)$	8	$Source_{10} + Dest_{10} + X \Rightarrow Dest$
NBCD	<ea>	8	$0 - Dest_{10} - X \Rightarrow Dest$
SBCD	$Dn, Dn - (An), -(An)$	8	$Dest_{10} - Source_{10} - X \Rightarrow Dest$

V poglavju 1.1.3 smo videli, da je BCD format nekakšen vmesni zapis med decimalnim in binarnim. Vsak naslednji levi nibble množimo z naslednjo višjo potenco baze 10, saj ustrezajo enicam, deseticam, stoticam itd. Zato veljajo za takšen zapis tudi posebna aritmetična pravila. Pri procesorjih, ki teh ukazov nimajo, je treba napisati posebne podprograme za BCD operacije, kot so seštevanje, odštevanje in negacija. Ukazi obravnavajo le LSByte.

Primer ukaza:

- ABCD D0, D1

Začetni pogoji: (D0) = 12345678_{HEX}, (D1) = 33333333_{HEX}.

Po ukazu: (D1) = 33333311_{HEX}, razširitveni bit X = "1".

Pojasnilo:

Skrajnja desna nibbla seštevamo enako kot decimalna števila. X bit v CCR se postavi v stanje 1, če rezultat presega obseg enega byta:

```

  78DEC
+ 33DEC
  11DEC
  └─ setiraj bit X

```

Rezultat "navadnega" binarnega seštevanja bi bil ab_{HEX}.

4.2.7 Ukazi za nadzor nad potekom izvajanja programa

Ukaz	Sintaksa	Dolžina operanda	Operacija
Pogojno izvajanje			
Bcc	<oznaka>	8, 16, 32	če pogoj <i>TRUE</i> , $PC + d \Rightarrow PC$
DBcc	Dn, <oznaka>	16	če pogoj <i>FALSE</i> , $Dn - 1 \Rightarrow PC$; če $Dn \neq (-1)$, $PC + d \Rightarrow PC$
Scc	<ea>	8	če pogoj <i>TRUE</i> , setiraj bite v Dest; drugače pa jih resetiraj
Brezpogojno izvajanje			
BRA	<oznaka>	8, 16, 32	$PC + d \Rightarrow PC$
BSR	<oznaka>	8, 16, 32	$SP - 4 \Rightarrow SP$; $PC \Rightarrow (SP)$; $PC + d \Rightarrow PC$
JMP	<ea>		$Dest \Rightarrow PC$
JSR	<ea>		$SP - 4 \Rightarrow SP$; $PC \Rightarrow (SP)$; $Dest \Rightarrow PC$
NOP			$PC + 2 \Rightarrow PC$
Vrnitve iz podprogramov			
RTD	#<d>	16	$(SP) \Rightarrow PC$; $SP + 4 + d \Rightarrow SP$
RTR			$(SP) \Rightarrow CCR$; $SP + 2 \Rightarrow SP$; $(SP) \Rightarrow PC$; $SP + 4 \Rightarrow SP$
RTS			$(SP) \Rightarrow PC$; $SP + 4 \Rightarrow SP$

Pogoj za skok pri pogojnih skokih izberemo tako, da oznako "cc" iz prejšnje tabele (ukazi Bcc, DBcc in Scc) zamenjamo z naslednjimi simboli:

CC	- C resetiran	CS	- C setiran
EQ	- enak (angl. equal)	T	- TRUE (trditev resnična)
GE	- večji (angl. greater) ali enak	F	- FALSE (trditev neresnična - le pri DBcc in Scc)
HI	- višji (angl. higher)	GT	- večji od
LE	- manjši ali enak (angl. less or equal)	LT	- manjši od
LS	- nižji ali enak (angl. low or same)	NE	- neenak
MI	- negativen	VC	- bit V v SP (presežek) resetiran
PL	- pozitiven	VS	- V setiran

Tabela 4. 1: Simboli za pogoje pri pogojnih skokih (vpišemo jih namesto cc)

Tukaj moramo opozoriti na razliko med pogojema "večji" in "manjši" ter "višji" in "nižji". Pogoja "večji" in "manjši" (ukazi GE, GT, LT in LE) pomenita, da se vsebini prirejenih operandov interpretirata kot predznačena števila (S). Pri pogojih "višji" in "nižji" (ukaza HI, LS) pa se vsebini operandov obravnavata kot nepredznačena števila (U, glej tudi pogl. 1.2).

Včasih je treba linearno izvajanje programov nadaljevati z ukazom na neki drugi lokaciji. Tak skok je lahko *brezpogojen* ali pa odvisen od rezultata prejšnje operacije oz. od stanja bitov v statusnem registru - *pogojni skok* (slika 4. 4).

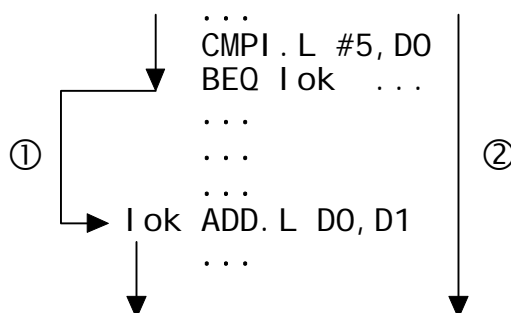
Poleg tega delimo skoke v dve veliki skupini:

- **enostavni skoki na lokacije znotraj podprograma,**
- **skoki na podprograme.**

V prvo skupino sodijo skoki, pri katerih se spremeni samo vsebina PC, vsi ostali registri pa ostanejo nespremenjeni. Druga skupina skokov je predvidena za nadaljevanje programa na ločeni programske sekvenci (t.i. *podprogamu*, angl. subroutine), po koncu katere se bomo vrnili na začetno skočno lokacijo. Pri tem se poleg PC spremeni tudi vsebina SP.

4.2.7.1 Enostavni skoki na lokacije

Slika 4. 4 kaže primer enostavnega pogojnega skoka na lokacijo, ki smo jo označili s simbolom (angl. label). V konkretnem primeru se bo izvajanje programa nadaljevalo z ukazom, ki se nahaja na lokaciji `lok` (sekvenca ①), če bo izpolnjen pogoj, da je setiran bit Z v SR (vsečina D0 je enaka 5). V nasprotnem primeru se bo program izvajal linearno (sekvenca ②). Pomembno je vedeti, da se bo tudi v tem primeru ukaz na oznaki `lok` izvajal po izvršitvi predhodnih ukazov, če med njimi ni takega drugega skoka. Pri brezpogojnih skokih (BRA in JMP) se sekvenca ① izvaja vedno, ne glede na rezultat prejšnje operacije.



Slika 4. 4: Primer enostavnega brezpogojnega skoka na lokacijo

Načeloma obstajata dva tipa ukazov za skoke oz. razvejitve: prvi imajo začetnico “J” (iz angl. jump - skok; npr. `JMP`), drugi pa “B” (iz angl. branch - razvejitev; npr. `BCC`). Pri prvem se naslov skočne lokacije nahaja v registru, ki je določen v 6-bitnem polju $\langle ea \rangle^7$ 16-bitne kode ukaza. V drugem primeru skočno lokacijo dobimo tako, da trenutni vrednosti PC prištejemo odmik (t.i. “displacement” - d). 8-bitni odmik (skočni naslov $PC \pm 127$), je sestavni del kode ukaza, za določanje 16- ali 32-bitnega odmika pa rabimo eno oz. dve dodatni besedi..

Primer ukazov:

⁷ Navadno jo označujemo s simboličnim naslovom.

Vsebina D0 je $c000\ 0000_{\text{HEX}}$. Po primerjavi D0 z 0 (npr. ukaz `CMP.L #0, D0`) bo pogoj za skok `BHI` skok izpolnjen, za `BGT` skok pa ne. Velja namreč:

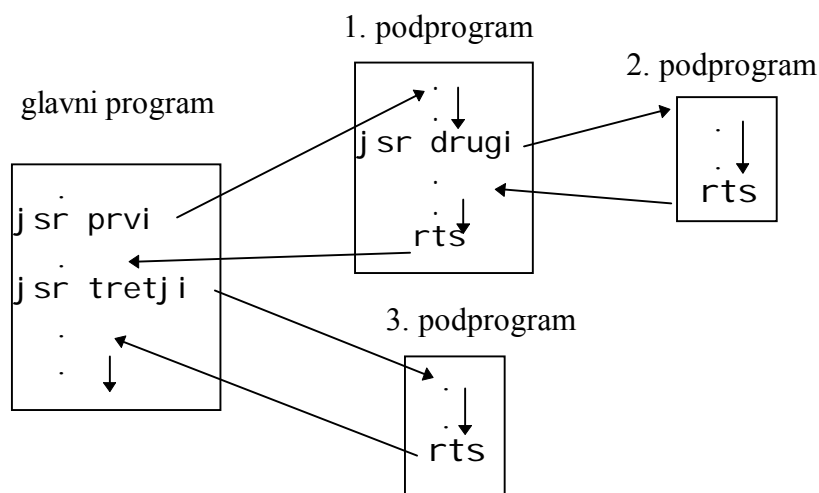
nepredznačeno (U): $c000\ 0000_{\text{HEX}} = 3\ 221\ 225\ 472_{\text{DEC}}$ (večji od nič),

predznačeno (S): $c000\ 0000_{\text{HEX}} = -1\ 073\ 741\ 824_{\text{DEC}}$ (manjši od nič).

4.2.7.2 Skoki na podprograme

Zapletene in dolge programe, kjer želimo ohraniti preglednost nad potekom izvajanja, ali programe, pri katerih med izvajanjem večkrat ponovimo enako programsko sekvenco⁸, je smiselno sestavljati modularno, v podprogramih ali funkcijah. Takrat govorimo o *strukturiranem programiranju* (slika 4. 5).

Za skoke na podprogram je značilna načelna vrnitev v glavni program ali podprogram, iz katerega smo izvršili prvotni skok. Izvajanje programa se nadaljuje z ukazom, ki sledi skočnem ukazu. Iz enega podprograma lahko skočimo na naslednjega, iz njega na naslednjega itd. Ta postopek imenujemo *gnezdenje* (angl. nesting), kar kaže slika 4. 5. Število nivojev gnezdenja je načeloma omejeno z velikostjo sklada.



Slika 4. 5: Blokovna shema strukturnega programiranja

Iz opisanega sledita dva osnovna koraka pri uporabi podprogramov (ne glede na to, ali imamo opravka z razvejitvijo ali s skokom):

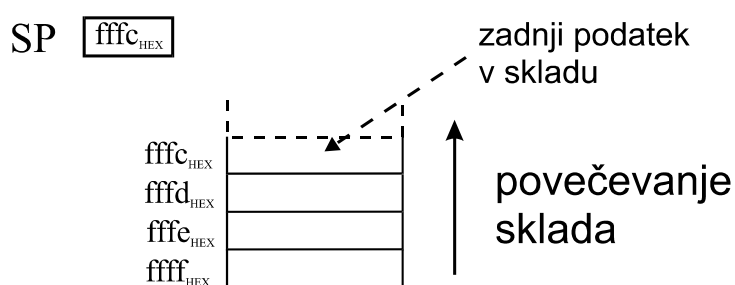
- **skok na podprogram (BSR ali JSR)**

⁸ Pri tem so lahko vhodni podatki v to sekvenco različni, ukazi pa vedno enaki. Npr. sekvenco za PI regulator v regulacijskem algoritmu enosmernega motorja lahko uporabimo za regulacijo hitrosti, fluksa ali toka.

1. kazalec sklada (SP) se zmanjša za štiri (sprosti prostor štirih bytov - 32 bitov - v skladu), ker se
 2. vrednost tekočega PC (32-bitni naslov ukaza za skočnim ukazom v starem podprogramu) naloži v sklad,
 3. v PC se naloži začetna lokacija podprograma (nadaljevanje izvajanja ukazov na podprogramskem modulu).
- **vrnitev iz podprograma** (ukaz RTS - angl. Return from Subroutine):
 1. vsebina lokacije sklada na katero kaže SP se naloži v PC (stara vsebina PC, ki kaže na naslednji ukaz za skokom v glavnem programu),
 2. kazalec sklada se vrne na prejšnjo vrednost (se poveča za štiri).

Kot vidimo, je ena od funkcij sklada začasno shranjevanje sistemskih podatkov (v tem primeru naslov, s katerim nadaljujemo izvajanje programa po vrnitvi iz podprograma). Pri izvajanju ukazov, kot so npr. JSR, BSR, RTS itd., se vrednost kazalca sklada, ki kaže na zadnji naloženi podatek, samodejno zmanjšuje (skok na podprogram) ali povečuje (vrnitev iz podprograma)⁹. Pri tem moramo paziti na naslednje:

- Če med skokom na podprogram in vrnitvijo iz njega (komplementarna ukaza) želimo nekaj naložiti v sklad oz. spremeniti vrednost v SP, moramo pred ukazom RTS vrniti SP v izhodiščno stanje. V nasprotnem se na vrhu sklada ne nahaja povratni naslov, zato ne preberemo pravilnega podatka in je vrnitev onemogočena. Na pravilno manipuliranje s skladom moramo biti še posebej pozorni pri gnezdenju.
- Za sklad moramo rezervirati področje v delovnem RAM pomnilniku. Njegova velikost je odvisna od zahtev programa (npr. od nivojev gnezdenja). Prav zaradi tega je definirana dinamično. Sklad se običajno postavi na **najvišji prosti naslov RAM pomnilnika**. Polnjenje sklada oz. njegovo povečevanje poteka proti nižjim lokacijam, torej se pri polnjenju sklada SP zmanjšuje (slika 4. 6). Edina skrb programerja je, da se sklad in ostali koristni podatki ali program v RAM tudi pri še tako velikem številu lokacij sklada nikoli ne prekrivajo!



Slika 4. 6: Smer polnjenja sklada

4.2.8 Ukazi za sistemski nadzor

⁹ Sklad deluje na LIFO principu (angl. Last In First Out): zadnji vhodni podatek bo tudi prvi izhodni.

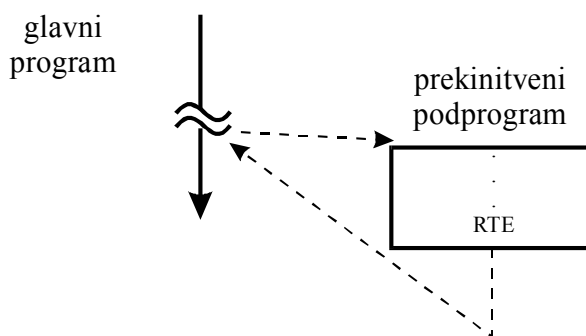
Ukaz	Sintaksa	Dolžina operand a	Operacija
Privilegirani			
ANDI	#<data>, SR	16	Data • SR ⇒ SR
EORI	#<data>, SR	16	Data ⊕ SR ⇒ SR
MOVE	<ea>, SR	16	Source ⇒ SR
	SR, <ea>	16	SR ⇒ Dest
MOVEA	USP, An	32	USP ⇒ An
	An, USP	32	An ⇒ USP
MOVEC	Rc, Rn	32	Rc ⇒ Rn
	Rn, Rc	32	Rn ⇒ Rc
MOVES	Rn, <ea>	8, 16, 32	Rn ⇒ Dest ob uporabi DFC
	<ea>, Rn		Source ob uporabi SFC ⇒ Rn
ORI	#<data>, SR	16	Data + SR ⇒ SR
RESET			aktiviraj linijo RESET
RTE			(SP) ⇒ SR, SP + 2 ⇒ SP; (SP) ⇒ PC, SP + 4 ⇒ SP; ponovno vzpostavi sklad ustrezno formatu
STOP	#<data>	16	Data ⇒ SR; STOP
LPSTOP	#<data>		Data ⇒ SR; prekinitevna maska ⇒ EBI; STOP
Generiranje TRAP funkcije			
BKPT	#<data>		če prekinitveni (breakpoint) cikel potrjen, izvrši vrnjeno ukazno besedo, drugače izvrši trap funkcijo zaradi ilegalnega ukaza
BGND			če background modus omogočen, izvrši background modus, drugače format/vektor odmik ⇒ - (SSP); PC ⇒ - (SSP); SR ⇒ -(SSP); (vektor) ⇒ PC
CHK	<ea>, Dn	16, 32	če Dn < 0 OR Dn < (ea), CHK prekinitev
CHKZ	<ea>, Rn	8, 16, 32	če Rn < nižja meja OR Rn > višja meja, CHK prekinitev
ILLEGAL			SSP - 2 ⇒ SSP; vektor odmik ⇒ (SSP); SSP - 4 ⇒ SSP; PC ⇒ (SSP); SSP - 2 ⇒ SSP; SR ⇒ (SSP); vektor nelegalnega ukaza ⇒ PC
TRAP	#<data>		SSP - 2 ⇒ SSP; format/vektor odmik ⇒ (SSP) SSP - 4 ⇒ SSP; PC ⇒ (SSP), SR ⇒ (SSP); naslov vektorja ⇒ PC
TRAPcc	ni□ #<data>	16, 32	če cc TRUE, TRAP prekinitev
TRAPV			če V setiran, TRAP prekinitev zaradi presežka
Register CCR (spodnji byte statusnega registra)			
ANDI	#<data>, CCR	8	Data • CCR ⇒ CCR
EORI	#<data>, CCR	8	Data ⊕ CCR ⇒ CCR
MOVE	<ea>, CCR	16	Source ⇒ CCR
	CCR, <ea>	16	CCR ⇒ Dest
ORI	#<data>, CCR	8	Data + CCR ⇒ CCR

Zgornja skupina ukazov vpliva na sistemski nadzor. Npr. ukaz **STOP** ustavi nadaljnjo obdelavo programa, enako kot **LPSTOP**, ki dodatno postavi mikrokrmilnik v čakalno stanje

minimalne porabe¹⁰ do nastopa prekinitve, ukaza TRACE ali resetiranja mikroračunalnika. Tukaj bomo največjo pozornost posvetili ukazu RTE (angl. Return from Exception), ki je potreben za vrnitev iz *prekinitvenega podprograma*.

4.2.8.1 *Prekinitve pri MC68332*

Prekinitve (angl. interrupts) so najhitrejši način programskega servisiranja nekega izjemnega dogodka. Po definiciji je to asinhronski dogodek, ki ustavi normalno izvajanje programske sekvence in začasno usmeri potek izvajanja programa na posebno programsko sekvenco (*prekinitveni podprogram*, angl. interrupt handler ali subroutine). Velikokrat se namreč zgodi, da je treba čim hitreje ukrepati na nek dogodek (npr. indikacija nadtokovne zaščite signalizira preobremenjenost motorja). Ukaz, s katerim bi ob normalnem poteku izvajanja programa preverjali obstoj omenjene nevarnosti, bi lahko prišel na vrsto prepozno. Zaradi tega je v tem primeru treba ukrepati s prekinitvijo izvajanja programa takoj ob pojavu signalizacije kritičnega dogodka ter izvršiti pripadajočo programsko sekvenco (slika 4. 7). Šele po tem ukrepu se potek izvajanja programa nadaljuje v točki, kjer je prišlo do prekinitve.



Slika 4. 7: Prekinitve izvajanja programa

Načeloma obstajata dve vrsti prekinitvev z ozirom na izvor zahteve po prekinitvi: softverska in hardverska (glej tudi tabelo 4. 2)¹¹. V prvem primeru imamo opravka s prekinitvami, ki so običajno posledica napak ali nelogičnosti v programu (npr. deljenje z 0, nepravilen ukaz, ukaza CHK in CHK2 itd.). Hardverske prekinitve so posledica reakcije na zunanji signal, ki ga pripeljemo na ustrezno definirane binarne vhode (**RESET**, **IRQ7** do **IRQ1**; glej blokovno shemo na sliki 3.3). Prekinitve izvajanja programa v CPU lahko zahtevajo tudi drugi podmoduli MC68332 (npr. TPU po predhodno določenem številu prešteti pulzov na vhodnem kanalu ali QSPI pri serijskem prenosu).

V grobem je zaporedje dogodkov pri prekinitvah naslednje:

¹⁰ Uporabno npr. pri baterijskem napajanju. S tem preide mikrokontroler v stanje, ko se mu aktivnost in poraba zmanjšata na minimum, potreben za ohranjanje osnovnih informacij. Stanje traja do ponovnega "prebujanja" ob določeni prekinitvi.

¹¹ V Motorolinem izrazoslovju označujejo splošno prekinitve ne glede na njen vzrok z izrazom "exception" (izjema), hardverske prekinitve pa dodatno imenujejo "interrupts" ("prekinitve"). V običajnem žargonu uporabljamo izraza "softverska in hardverska prekinitve".

1. Zahteva po prekinitvi: zahtevnik prekinitve “zaprosi” CPU za prekinitev izvajanja programa.
2. CPU zazna zahtevnika prekinitve in mu ustreže (glej podpoglavje 4.2.8.1.1) ali pa ga ignorira.
3. Izvajanje se nadaljuje na prekinitvenem podprogramu, ki se obvezno konča z ukazom RTE (glej podpoglavje 4.2.8.1.2)!
4. Izvajanje programa se nadaljuje z ukazom v “glavnem programu”, ki je bil na vrsti, preden je prišlo do zahteve po prekinitvi.

4.2.8.1.1 Hierarhija in servisiranje prekinitev

Nekatere prekinitve se izvajajo brezpogojno, druge pa lahko onemogočimo. Primer brezpogojne prekinitve je hardverska prekinitev RESET, ki reinicializira celoten operacijski sistem: izvajanje programa se dokončno prekine in vsi registri se postavijo v začetno stanje. Prekinitve so namreč glede na prednost (prioriteto) razdeljene na več skupin. Tako ima hardverska prekinitev IRQ7 najvišjo, IRQ1 pa najnižjo prioriteto. Vse ostale prekinitve razen IRQ7 lahko onemogočimo s t.i. “maskiranjem”. Izraz pomeni postavitvev področja I2, I1, I0 (prekinitvena maska) v sistemskem zlogu SR v kombinacijo, ki dovoljuje le nekatere prekinitve (podpoglavje 4.1.2). Omogočene so vse prekinitve, katerih zaporedna številka je večja od binarnega števila v prekinitveni maski. Če se v prekinitveni maski nahaja kombinacija $100_{\text{BIN}} = 4_{\text{DEC}}$, sta IRQ5 in IRQ6 omogočeni, IRQ1 - IRQ4 pa ne. Edina izjema je IRQ7, ki je tudi kombinacija $111_{\text{BIN}} = 7_{\text{DEC}}$ ne more preprečiti. Hierarhija prekinitev pride zlasti do izraza v primeru hkratne zahteve po več prekinitvah. V takšnem primeru imajo prekinitve z višjo prioriteto prednost pri izvajanju.

V fazi inicializacije moramo tudi določiti začetni naslov podprograma, ki se bo izvajal ob določeni prekinitvi. V t.i. “vektorski tabeli” se nahajajo kazalci na začetne lokacije podprogramov vseh prekinitev (t.i. vektorji). Programer mora v vektorje predvidenih prekinitev vpisati začetne naslove ustreznih prekinitvenih podprogramov. Tabela 4. 2 kaže vektorje in njihove naslove v vektorski tabeli za nekatere prekinitve

Številka vektorja	Odmik vektorja		Opis
	DEC	HEX	
0	0	0	Reset: začetni SP
1	4	4	Reset: začetni PC
3	12	c	Napaka pri naslavljanju
4	16	10	Nelegalni ukaz
5	20	14	Deljenje z 0
48-58	192 232	c0 e8	Rezervirano za koprocessor
64-255	256 1020	100 3fc	Vektorji, ki jih definira uporabnik (npr. za TPU).

Tabela 4. 2: Vektorska tabela

Položaj vektorske tabele v pomnilniku (njen začetek) določa vsebina registra VBR (glej podpoglavje 4.1.2). Na ta način lahko s spremembo vsebine VBR določimo več vektorskih tabel.

Primer:

Če hočemo napisati lasten podprogram, ki se bo izvajal v primeru operacije deljenja z nič, moramo med inicializacijo na lokacijo (VBR) + 14_{HEX} shraniti naslov začetnega ukaza tega prekinitvenega podprograma.

4.2.8.1.2 Vrnitev iz prekinitvenega podprograma

Ob zahtevi po prekinitvi se vsebina SR shrani v sklad. Po skoku na prekinitveni podprogram se v sklad shrani tudi vsebina PC, enako kot pri skokih na navadne podprograme. Ta “stari” PC kaže na ukaz, pred katerim je prišlo do prekinitve. Temu ustrezno se spremeni tudi SP. To stanje je treba po opravljeni prekinitveni proceduri restavrirati, za kar poskrbi ukaz RTE, podobno kot RTS pri vrnitvi iz navadnih podprogramov.

4.3 Načini naslavljanja

Večina ukazov ima na voljo več možnosti definiranja izvora in končnega cilja operandov. Izbira primerne načina naslavljanja je zelo pomembna, saj neposredno vpliva na hitrost izvajanja in obsežnost programa. V tem podpoglavju si bomo ogledali le najpogostejše načine naslavljanja, ki so skupni večini mikroprocesorjev, podrobnejše informacije o CPU32 pa so v literaturi [25]. Pri ukazih z dvema operandoma lahko načeloma za vsak operand uporabimo drugačen načina naslavljanja.

4.3.1 Neposredno naslavljanje registrov

Oznaka: Dn ali An

Pri tem načinu naslavljanja (angl. direct addressing) se operandi nahajajo v podatkovnem ali naslovnem registru.

Primer:

```
add.l A0,D7      *vsebina A0 + vsebina D7 ⇒ D7
move.b D3,D4     *vsebina byta z najnižjo težo iz D3 ⇒ D4
```

Značilnost tega naslavljanja je, da se eden (ali oba) operanda nahajata v registrih CPU. Na ta način dosežemo najhitrejšo izvajanje operacij, saj kakršnokoli drugačno naslavljanje, ki operira s pomnilnikom (posebej, če le-ta ni na čipu), zahteva več urin ciklov.

4.3.2 Posredno naslavljanje s pomočjo naslovnega registra

Oznaka: (An)

Pri tem načinu naslavljanja (angl. address register indirect) je eden od operandov (ali oba) shranjen v pomnilniku na naslovu, ki ga določa naslovni register.

Primer:

```
MOVE.L (A0),D2    *□e je vsebina A0 effcHEX, vsebina te spominske
                   *lokacije pa takšna kot na sliki 4. 8, bo
                   *vsebina registra D2 po ukazu 23535ACbHEX
```

effc _{HEX}	23
effd _{HEX}	53
effe _{HEX}	5a
efff _{HEX}	c6

Slika 4. 8: Primer vsebine spominske lokacije

4.3.3 Posredno naslavljanje s pomočjo naslovnega registra z naknadnim inkrementiranjem

Oznaka: (An)+

Ta način je podoben prejšnjemu, le da se vsebina naslovnega registra (naslov operanda) po ukazu samodejno poveča in kaže na naslednjo lokacijo. Povečanje je odvisno od uporabljenega tipa operanda (npr. pri "long" operandih so to štirje byti).

Primer:

```
MOVE.L (A0)+,D2    *□e je vsebina registra A0 effcHEX, vsebina
                   *te spominske lokacije pa takšna kot na
                   *sliki 4. 8 bo po ukazu vsebina D2
                   *23535acbHEX, vsebina A0 pa f000HEX
```

Način naslavljanja je zelo uporaben pri dostopu do zapovrstnih podatkov, npr. elementov tabele.

4.3.4 Posredno naslavljanje s pomočjo naslovnega registra ob predhodnem dekrementiranju

Oznaka: -(An)

Zopet imamo opravka z naslavljanjem, ki je podobno predhodnima dvema. Razlika je le v tem, da se pred uporabo naslovnega registra njegova vsebina zmanjša za 1, 2 ali 4. Dekrement je odvisen od podatkovne širine operanda v ukazu. Pri bytnem operandu je enak ena, pri besednem dve, pri dolgi besedi pa štiri.

Primer:

```
MOVE.L -(A0),D2      *□e je vsebina A0 f000HEX, vsebina te
                      *spominske lokacije pa kot na sliki 4. 8,
                      *se pred uporabo v ukazu vsebina A0
                      *spremeni v effcHEX, vsebina D2 pa bo
                      *potemtakem 23535acbHEX.
```

4.3.5 Takojšnje podajanje podatka

Oznaka: #XXX

Pri določenih ukazih (s podaljškom "I" - angl. Immediate - takojšen) lahko enega izmed operandov podamo neposredno v samem ukazu.

Primer:

```
addi.b #23,D0         *k vsebini spodnjega byta registra D0
                      *(stara vsebina f0HEX) prištej 23HEX.
                      *Rezultat: (D0)=13HEX. PAZI! Zaradi bytnega
                      *operanda se presežek ne prenese na višji
                      *byte.
```

4.3.6 Absolutni dolgi naslov

Oznaka: (XXX).L

S tem načinom (angl. absolute long address) neposredno podajamo naslov, na katerem se nahaja operand.

Primer:

```
NEG.L ($123456).L     *negiranje vsebine spominske lokacije z
```



```
*naslovom 123456HEX.  
CLR.W (&8000).L      *brisanje besede na naslovu  
                        *8000DEC = 1f40HEX.
```

4.3.7 Posredno naslavljanje z indeksom (in osnovnim odmikom)

Oznaka: (bd, An, Xn. SIZE*SCALE)

Oglejmo si sedaj primer nekoliko bolj zapletenih načinov naslavljanja. Naslov, na katerem se nahaja podatek, se računa tako, da naslovu v An prištejemo vrednost osnovnega odmika (angl. basic displacement - bd) ter skalirano vrednost indeksnega registra:

$$ea = bd + (An) + (Xn * SCALE).$$

Kot indeksni register lahko uporabljamo podatkovni ali naslovni register. V njem lahko definiramo 16-bitni (sufiks W) ali 32-bitni indeks (sufiks L). Vrednost indeksnega registra lahko množimo s faktorjem SCALE, katerega možne vrednosti so 1, 2, 4 ali 8.

Kot primer vzemimo ukaz: `MOVE.W D0, (8000, A0, D1.L*2).`

Pred ukazom so vrednosti posameznih registrov:

```
(d0) = 1234HEX ,  
(d1) = 1000HEX ,  
(a0) = 2000HEX.
```

Efektivni naslov, na katerega shranimo besedo iz D0, izračunamo po zgornji enačbi:

$$8000_{HEX} + 2000_{HEX} + 1000_{HEX} \cdot 2 = b000_{HEX},$$

torej bo po izvršitvi ukaza:

$$(b000) = 1234_{HEX}$$

4.4 Čas izvajanja posameznega ukaza

Čas, ki je potreben za izvajanje posameznega ukaza, je vsekakor eden od pomembnih parametrov pri presojanju zmogljivosti procesorja. Pri nekaterih procesorjih (npr. pri digitalnih signalnih procesorjih - DSP) je ta čas ob enakem načinu naslavljanja enak za vse ukaze¹², razen pri operaciji deljenja, kjer traja nekajkrat več (npr. 16-krat pri 16-bitnem deljenju).

Pri običajnih procesorjih, kamor sodi tudi CPU32, je izračun trajanja določene programske sekvence zapleten, saj je čas izvajanja posameznega ukaza odvisen od naslednjih faktorjev:

¹² Npr. pri TMS320F240 je to 50 ns pri najhitrejšem načinu naslavljanja.

- tipa ukaza,
- načina naslavljanja,
- formata operandov,
- urinega takta, saj lahko isti procesor dela na različnih frekvencah.

Izračun potrebnih časovnih ciklov je sestavljen iz več segmentov¹³. V naslednji tabeli so za ilustracijo prikazani faktorji za izračun urinih ciklov, ki so potrebni za izvajanje nekaterih značilnih ukazov. Samega algoritma izračuna in dodatnih ciklov, ki so odvisni od načina naslavljanja in formata operandov, ne bomo navajali. Že sama razmerja med različnimi ukazi pa ilustrirajo velike razlike v trajanju izvajanja.

Ukaz	Začetni cikli (head)	Končni cikli (tail)	Cikli
ADD R _n , R _m	0	0	2(0/1/0)
OR <FEA>, D _n	0	0	2(0/1/0)
MOVE R _n , R _n	0	0	2(0/1/0)
MOVE.L <FEA>, (A _n)	2	2	6(0/1/2)
ADDI #, R _n	0	0	2(0/1/0)
ADDI #, <FEA>	0	3	5(0/1/2)
BSET D _n , D _m	4	0	6(0/1/0)
RTE	1	-2	24(4/2/0)
MULS(U).W <FEA>, D _n	0	0	26(0/1/0)
DIVS.W <FEA>, D _n	0	0	2(0/1/0)

Slika 4. 9: Tabela za izračun časovnih ciklov potrebnih za izvajanje nekaterih ukazov CPU32

Iz tabele je razvidno, da so razlike med časi izvajanja posameznih ukazov zelo velike, še zlasti pri aritmetičnih operacijah seštevanja, odštevanja, množenja in deljenja.

Izkušen programer se bo v časovno kritičnih programih skušal izogniti uporabi časovno zahtevnejših ukazov (kot so npr. deljenja in množenja). Oglejmo si to na primeru množenja spremenljivke, ki se nahaja na naslovu 8000_{HEX}, s faktorjem 7.

V prvem trenutku se najenostavnejša zdi uporaba ukaza za množenje:

```
LEA      $8000,a5    * naslov spremenljivke v a5
MOVE.W   (a5),d1     * nalaganje spremenljivke v D1
MULS.W   #7,d1       * množenje s 7
MOVE.W   d1,(a5)     * shranjevanje rezultata
```

Naslednja rešitev je nekoliko hitrejša, čeprav rabimo več ukazov:

¹³Število ciklov za izvajanje inštrukcije, cikli za zaključek predhodne inštrukcije, cikli za bralni in vpisovalni pristop, zajemanje inštrukcije (angl. fetch) itd.

```

LEA      $8000,%a5    * naslov spremenljivke v a5
MOVE.W   (%a5),%d1     * nalaganje spremenljivke v D1
MOVE.W   %d1,%d0       * preslikava spremenljivke v D0
LSL.W    #3,%d0        * pomik vsebine D0 za 3 mesta (bite) v
levo
                        * (množenje z 8)
SUB.W    %d1,%d0       * odštevanje (D0) od (D1): D0=(8 - 1)*D1
MOVE.W   %d0,(%a5)     * shranjevanje rezultata

```

Zgornjo rešitev bi dobili, če bi množenje s konstanto izvedli v višjem jeziku (npr. C) ob časovni optimizaciji.