SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Doc. Ing. PETER FUCHS PhD., Ing. ALEK LICHTMAN

VÝVOJOVÉ PROGRAMOVACIE PROSTREDIA PRE MECHATRONICKÉ SYSTÉMY - SIGNÁLOVÉ PROCESORY

BRATISLAVA 2017

VÝVOJOVÉ PROGRAMOVACIE PROSTREDIA PRE MECHATRONICKÉ SYSTÉMY - SIGNÁLOVÉ PROCESORY

Doc. Ing. PETER FUCHS PhD.
Ing. ALEK LICHTMAN

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
2017

OBSAH

Kapitola 1

Úvod a prehľad

Odsek 1-1. Učebné ciele:

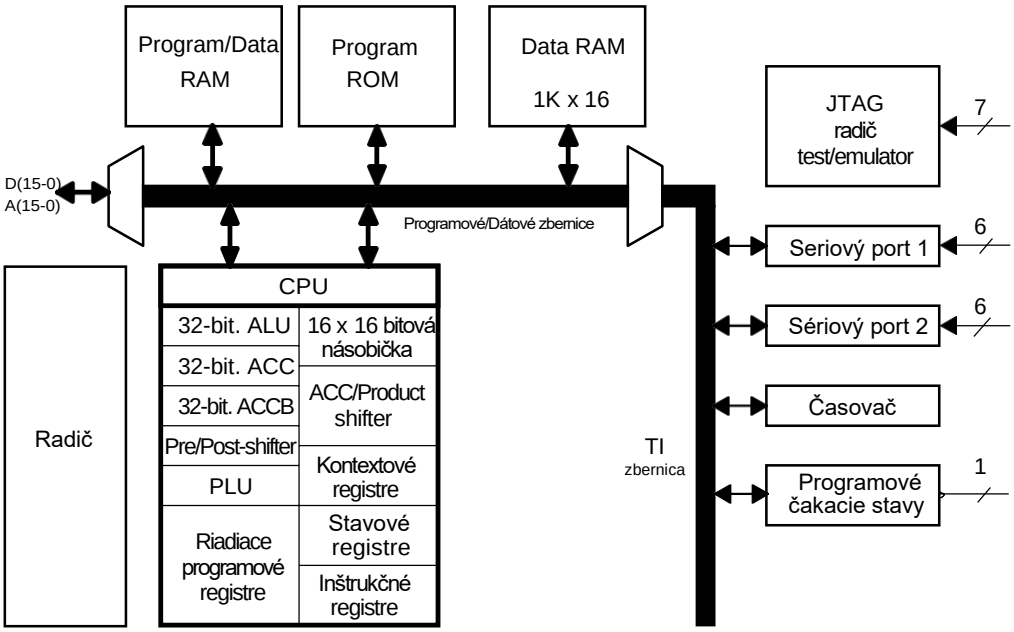
Na záver tejto kapitoly by ste mali byť schopní:

- Nakresliť základný blokový diagram TMS320C5x
- Urobiť zoznam kľúčových bodov pamäťovaj mapy a zbernicových štruktúr TMS320C5x
- Opísať rozdiely medzi zariadeniami TMS320C5x
- Rozumieť informáciám zobrazeným v každom z okien simulátora
- Usporiadať okná simulátora, aby zobrazili informácie, ako sú žiadané
- Zaviesť a spustiť program simulátora
- Vykonať základné simulačné príkazy priamo, cez skratkové klávesy a pomocou myši

1.1 Základy architektúry piatej generácie TMS320

Procesor TMS320C5x používa modifikovanú *Harvardskú architektúru*. Táto modifikácia pozostáva z rozšírenia Harvardskej architektúry o prvky von Neumannovej architektúry. Hlavnou výhodou je schopnosť inicializovať dáta z programovej pamäte. Táto schopnosť dovoľuje 'C5x-ke časový multiplex pamäte medzi úlohami, takisto ako vyvolať dáta s konštantami (napr. koeficienty) uchované v systémovom programe ROM. To zmenšuje systémové náklady eliminovaním potreby pre dáta ROM a maximalizuje dátovú pamäť dovolením dynamického predefinovania dátovej pamäťovej funkcie.

Odsek 1-2. Blokový diagram TMS320C5x



Najväčším dôvodom pre stavbu 'C5x na Harvardskej architektúre je rýchlosť. Oddelený dátový a programový priestor umožňuje simulačné vyvolanie programových inštrukcií a dát. V matematicky intenzívnych aplikáciách toto efektívne zdvojuje priepustnosť algoritmu v porovnaní so (štandardným) von Neumannovým typom procesorov.

1.1.1 Verzie TMS320C5x

Zariadenia 'C5x sa líšia hlavne v pamätiach na čipe. Ďalšie variácie zahŕňaju typ zapúzdrenia a operačné napätie. Pre čo najnižšiu cenu, 'C52 je prístupná v 100-pinovom púzdre s jednoduchým sériovým portom (nie TDM).

Odsek 1-3. Verzie TMS320C5x

Zariadenie	RAM	ROM	Púzdro	Poznámka
50	10K	2K	132 PQFP	
51	2K	8K	132 PQFP, TQFP	
52	1K	4K	100 TQFP	PLLx1,2,3 3/5v
53	4K	16K	132 PQFP	
56	7K	32K	100 TQFP	*1
57	7K	32K	128 TQFP	*2

*1: Rozšírený sériový port s auto-DMA a použitie oddelitelného PLL (x1 cez x3)
*2: 'C57 je 'C56 plus 8-bitové hostiteľské rozhranie (port interface)

Každé z týchto zariadení je ponúkané v rozsahu rýchlostí od 50 do 25 ns. Sú prístupné s 5V alebo 3,3V napájaním, dovoľujúcim väčší výber s rešpektovaním výkonu, spotreby, ceny, atď.

1.1.2 Rozšírenia piatej generácie

Produkty 'C5x majú mnoho vylepšení oproti zariadeniam prvej a druhej generácie ako ukazuje nasledujúci odsek.

Odsek 1-4. Vylepšenia piatej generácie:

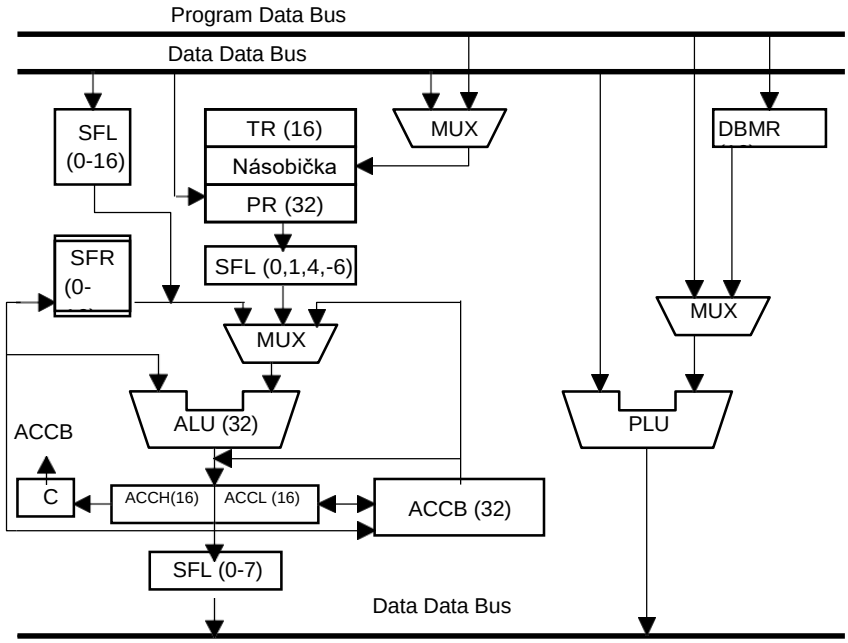
- Rýchlejší čas cyklu
- Rýchlejšia odpoveď na prerušenie
- Bitové operácie cez paralelnú logickú jednotku
- Väčšia pamäť na čipe
- Snímacia zbernica JTAG podľa normy IEEE
- Hodiny pre redukovanú EMI
- Blokové odpovede
- Softwarové generovanie čakacích stavov
- Rýchlejšie vetvenie, volanie a návrat
- Kruhové vyrovnávacie pamäte (circular buffers)
- Druhý akumulátor
- ALU 0- až 16-bitový posuv

Architektúra 'C5x môže byť rozdelená do troch kategórií: CPU, pamäť a periférie.

1.2 CPU

Centrálna procesorová jednotka (CPU) zlučuje násobičku, centrálnu aritmeticko-logickú jednotku (CALU), paralelnú logickú jednotku (PLU) spolu s piatimi posuvnými registrami (shifters), a 28 programom riadených registrov.

Odsek 1-5. Centrálna procesorová jednotka



Vylepšenia CPU 'C5x oproti CPU 'C2x zahŕňajú 32-bitovú vyrovnávaciu pamäť (buffer) akumulátora, schopnosť meniť mierku (scaling) a množstvo nových inštrukcií, ktoré operujú nad novým hardwarom. Nové riadiace funkcie obsahujú nezávislú PLU pre vykonanie boolovských operácií a sadu kontextových registrov pre okamžité uchovanie systémových informácií (context-saving) pri obslužnom programe prerušenia (ISRs). Správa dát bola vylepšená použitím nových inštrukcií presúvajúcich bloky a inštrukcií pamäťovo mapovaných registrov.

1.2.1 Násobička

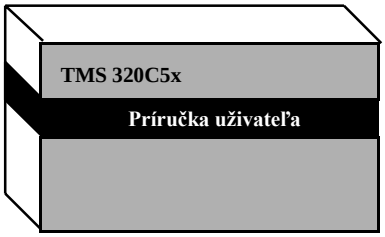
Hardwarová násobička je navrhnutá tak, aby vykonávala násobenie v jednom strojovom cykle. Jeden vstup prichádza z dočasného registra (TR) a ostatné z dátovej alebo programovej zbernice. 32-bitový výsledok je uchovaný v produktovom registri (PR), ktorý je potom použiteľný pre CALU.

'C5x má násobiť/nahromadiť inštrukcie, ktoré sa vykonávajú v jednoduchom cykle, keď sú použité v opakovaní inštrukcií.

1.2.2 CALU

Centálna aritmeticko logická jednotka

Odsek 3-12, str. 3-23



CALU obsahuje ALU a štyri oddelené posuvné registre. ALU vykonáva jednoduchý cyklus, 16- alebo 32-bitové aritmetické alebo logické operácie. Výsledky sú uložené v 32-bitovom akumulátore (ACC). Hodnoty v ACC a ACCB môžu byť spojené počas aritmetických alebo logických operácií a môžu tiež uskutočniť široký 65-bitový prírastkový posuv (vrátane carry).

'C5x má vstupný posuvný register z dátovej zbernice do ALU, výstupný posuvný register z ACC na dátovú zbernicu, produktový posuvný register pre menenie rozmeru výsledkov násobičky a akumulátorový posuvný register schopný 0- až 16-bitového posuvu vpravo.

Inštrukcia ALU je vždy vykonávaná nasledovne:

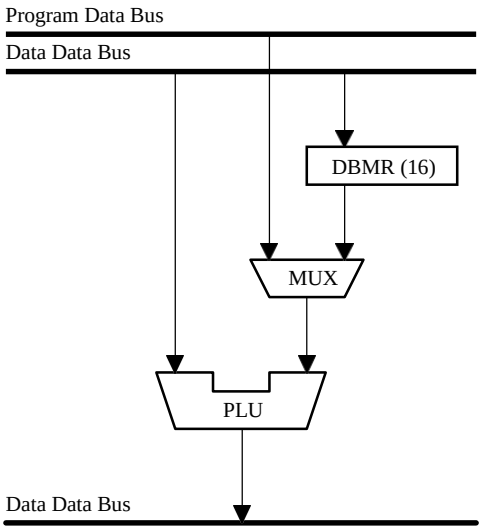
1. Dáta sú čítané z RAM na dátovú zbernicu.
2. Dáta prechádzajú cez škálovací posuvný register (scaling shifter) a ALU, kde sú operácie vykonávané.
3. Výsledok sa nachádza v akumulátore.

Jeden vstup do ALU je vždy obstarávaný akumulátorom. Druhý môže byť premiestnený z produktového registra (PR), alebo zo škálovacieho posuvného registra tak, že je zavedený z dátovej pamäte alebo ACC.

1.2.3 PLU

Paralelná logická jednotka (PLU) obstaráva druhú cestu pre boolovské funkcie alebo bitové manipulácie. Je to nezávislá jednotka, ktorá operuje oddelene (paralelne) od ALU. PLU sa používa na nastavenie, nulovanie, testovanie alebo prepínanie bitov v riadiacom/stavovom registri alebo na ktoromkoľvek mieste dátovej pamäte.

Odsek 1-6. Paralelná logická jednotka



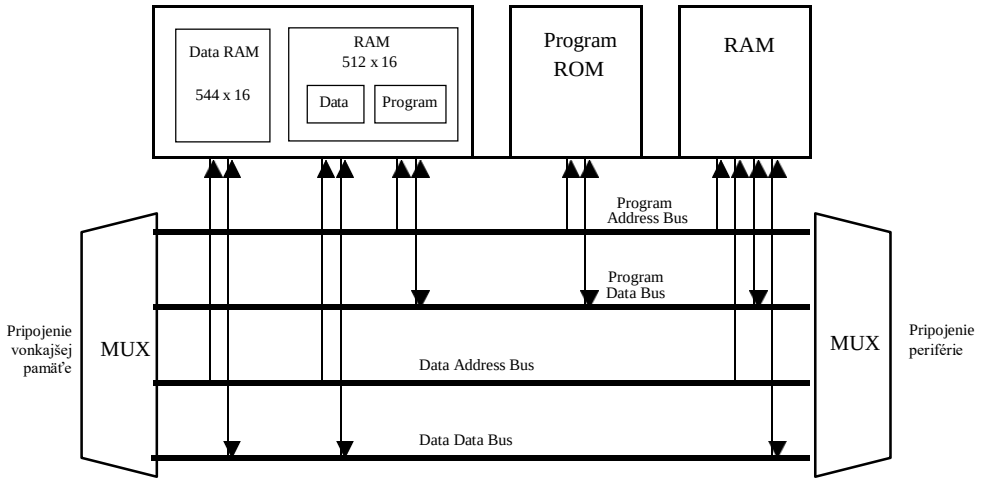
PLU obstaráva priamu logickú operačnú cestu k hodnotám dátovej pamäťe bez zmeny obsahu akumulátora alebo produktového registra. PLU číta dátové hodnoty zo špecifického miesta v dátovej pamäti. Táto hodnota je spracúvaná podľa hodnoty masky, ktorou je každá konštanta vložená v inštrukčnom slove alebo obsah dynamického bitovo-manipulačného registra (DBMR). Výsledky PLU sú zapísané späť na pôvodné miesto v dátovej pamäti.

1.3 Pamäť

'C5x má 224K x 16-bitov úplného adresného priestoru. Pamäťový priestor je rozdelený do štyroch oblastí:

- 64K programová
- 64K dátová
- 64K I/O
- 32K všeobecná

Odsek 1-7. Organizácia pamäte



Všetky zariadenia 'C5x majú 1K RAM, ktorá je niekedy popisovaná ako RAM s dvojitým prístupom. Môže uskutočniť oneskorenie bez potreby "kruhového vyrovnávania" (circular buffering) ako u iných pamätí. Pri resetovaní sa táto pamäť nachádza v dátovom priestore, ale časť môže byť premiestnená pod kontrolou softwaru do programového priestoru.

Druhou pamäťou na čipe je ROM, ktorá sa nachádza na začiatku programového priestoru. Môže byť použitá alebo obídená pod kontrolou pinu pre reset a použitého programu. Veľkosť ROM sa mení podľa typu zariadenia od 2K až po 16K.

Druhý typ RAM je niekedy volaný "RAM s jednoduchým prístupom". Podľa typu zariadenia sa mení od 0K až po 9K. Pri resetovaní je táto pamäť nezávislá a môže byť usmernená časťou dátovej a/alebo programovej pamäte pod kontrolou použitého programu. Hoci je nazvaná RAM s jednoduchým prístupom, táto pamäť môže byť použitá pre oba, programový aj dátový prístup v jednoduchom cykle, keď je ťahaný z rozdielnych 2K blokov. Tak ako všetky pamäťové prístupy, duálne prístupy k podobným poliam operujú bez chyby automatickým pridaním extra cyklu.

1.4 Zbernicové štruktúry

Veľká časť vykonávania príkazov 'C5x je zviazaná s vnútorným "busovaním" a výsledným paralelizmom.

Dvomi hlavnými zbernicami sú programová zbernica a dátová zbernica:

- Programová zbernica
  - Pridáva kód
  - Číta operandy
- Dátová zbernica
  - Číta operandy
  - Zapisuje výsledky

Paralelizmus dovoľuje súčasne privádzať oddelené operácie:

- Operácie s dátami v CALU, napr. matematické

  or
- Logické operácie v PLU

  plus
- Aritmetické operácie v ARAU, napr. zmena smerníka

  plus
- Privedenie novej inštrukcie z programovej pamäte

1.5 Reťazenie (pipeline)

Zariadenia 'C5x prevádzajú reťazenie inštrukcií, ktorého výsledkom je účinejšie vykonanie operácie a programu. Inštrukčné fázy privedenie-dekódovanie-čítanie-vykonanie sú navzájom nezávislé. Inštrukcie sa prekrývajú tak, že v ľubovolnom cykle môžu byť aktívne maximálne štyri inštrukcie, každá v inom stave uskutočnenia.

Odsek 1-8. Proces posuvu

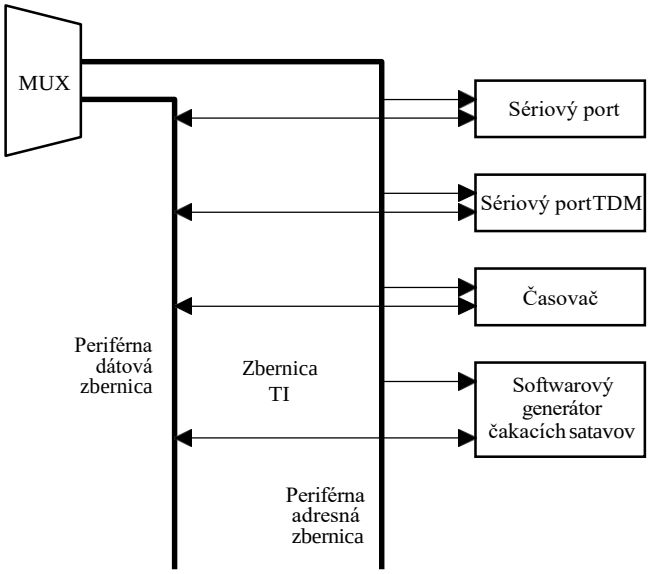
Čas:	100	101	102	103	104	105	106
Add	F_1	D_1	R_1	E_1			
Sub		F_2	D_2	R_2	E_2		
Mpy			F_3	D_3	R_3	E_3	
Store				F_4	D_4	R_4	E_4

Plne nahratý pipeline
(normálna operácia)

1.6 Periférie

Periférie 'C5x sú obsluhované cez registre mapované v dátovom pamäťovom priestore. Tieto periférie sú prístupné cez neviditeľnú, vnútornú perifernu zbernicu (TI BUS).

Odsek 1-9. Periférne moduly



1.6.1 Sériové porty

'C5x má dva sériové porty:

- Obojsmerný synchrónny port programovateľný pre množstvo módov a prenosových rýchlostí
- Obojsmerný port, ktorý môže byť konfigurovaný ako každý synchrónny sériový port, alebo ako port s časovo-deleným viacnásobným prístupom (TDM). TDM mód dovoľuje až ôsmim zariadeniam 'C5x komunikovať cez jednoduchý sériový port.
- Najnovšie zariadenia 'C5x, 'C56 a 'C57 pridávajú DMA ako interface k vyhradenému poľu v pamäti.

1.6.2 Hardwarový časovač

Hardwarový intervalový časovač zaobstaráva 16-bitové odpočítavanie cez periodický a riadiaci register. Funkcie start, stop, restart, reset a disable (zablokovanie) sú riadené cez riadiaci register časovača.

1.6.3 Softwarové generátory čakacích stavov

Softwarové generátory čakacích stavov dovoľujú pripojenie k pomalšej pamäti mimo čipu a I/O zariadeniam. Táto periféria môže eliminovať potrebu externej logiky k vytvoreniu čakacích stavov cez READY pin. Obvod pozostáva zo 16-tich obvodov generujúcich čakanie a je programovateľný tak, aby vykonával 0, 1, 2, 3, alebo 7 čakacích stavov. Programová aj dátová pamäť môžu byť delené do štyroch 16K slovných blokov, s jedným generátorom čakacích stavov na blok. Podobne I/O pamäť je rozdelená do ôsmich 8K blokov.

1.6.4 Paralelné I/O porty

'C5x má celkovo 64K I/O adries, z ktorých 16 je tiež použiteľných ako pamäťovo mapované registre. Tieto porty sú multiplexované s dátovými linkami a sú identifikované cez I/O Space Select signál.

1.6.5 JTAG logika snímania

'C5x podporuje normu IEEE 1149.1 pre snímaciu zbernicu testovania JTAG. Táto 5-pinová sériová zbernica je použitá k snímaniu stavov ľubovolného registra alebo pamäťového miesta vo vnútri alebo mimo zariadenia. Sériová snímacia cesta je použitá k vytvoreniu vlastnej emulácie, ktorá nezasahuje do cieľovej systémovej operácie. Toto môžeme využiť pri zastavení a krokovaní procesora, kým registre na čipe sú použité pre breakpoint a trace.

1.6.6 Prerušenia

Každé zo štyroch vonkajších prerušení 'C50-ky a 'C51-ky je vnútorne zachytávané, dovolujúc asynchrónnu funkciu prerušení. Je tu päť vnútorných prerušení: jedno z časovača a štyri zo sériových portov.

Okamžité uchovanie kontextu je dosiahnuté použitím tieňových registrov. Jedenásť strategických registrov CPU má jeden zásobník, ktorý slúži k uchovaniu obsahov registrov CPU, keď sa vykonáva obsluha prerušenia. To redukuje alebo eliminuje čas obyčajne strávený prioritným uchovávaním kontextu pred začatím obsluhy prerušenia. Výsledkom je rýchlejšia odpoveď na prerušenie a jednoduchšie, kratšie kódovanie prerušenia.

1.7 Inštrukčná sada

Inštrukčná sada je nadstavbou inštrukčnej sady 'C1x-ky a 'C2x-ky. Assembler 'C5x-ky bude akceptovať inštrukcie 'C2x-ky takisto ako inštrukcie 'C5x.

Odsek 1-10. Inštrukčná sada TMS320C5x

Pamäťové referencie cez akumulátor

ABS	NEG	SFRB
ADCB	NORM	SUB
ADD	OR	SUBB
ADDB	ORB	SUBC
ADDC	ROL	SUBS
ADDS	ROLB	SUBT
ADDT	ROR	XOR
AND	RORB	XORB
ANDB	SACB	ZALR
BSAR	SACH	ZAP
CMPL	SACL	
CRGT	SAMM	
CRLT	SATH	
EXAR	SATL	
LACB	SBB	
LACC	SBBB	
LACL	SFL	
LACT	SFLB	
LAMM	SFR	

Pomocný register a smerník dátovej stránky

ADRK	MAR
CMPR	SAR
LAR	SBRK
LDP	

PLU

APL	SPLK
CPL	XPL
OPL	

Vetvenie

B [D]	CALL [D]
BACC [D]	CC [D]
BANZ [D]	INTR
BCND [D]	NMI
CALA [D]	RET
RETE	RETC [D]
RETI	TRAP
	XC

I/O a dátové pamäťové operácie

BLDD	LMMR
BLDP	OUT
BLPD	SMMR
DMOV	TBLR
IN	TBLW

Riadiace registre

BIT	LST
BITT	NOP
CLRC	POP
IDLE	POPD
IDLE2	PSHD
RPT	PUSH
RPTB	SETC
RPTZ	SST

Násobenie, T, P

APAC	MPYA
LPH	MPYS
LT	MPYU
LTA	PAC
LTD	SPAC
LTP	SPH
LTS	SPL
MAC	SPM
MACD	SQRA
MADD	SQRS
MADS	ZPR
MPY	

1.8 Programovanie v jazyku C

C-kompilátor plne vyhovuje norme ANSI a výstupom je zdrojový kód assemblera. C-kompilátor podporuje vnútorne assemblerovský kód; volanie assemblera z C a volanie C z assemblera. Premenné definované v zdroji C môžu byť adresované v kóde assemblera a naopak. Tým, že je dovolené miešať C a assembler, programátor môže využiť C k dosiahnutiu rýchlejšieho rozvoja programu a assembler k písaniu tých častí, ktoré si žiadajú optimálne vykonanie.

C kód na TMS320 je relatívne účinejší tým, že sprístupní optimalizáciu počas volania kompilácie. Špeciálny vnútorný hardware ako aj osem pomocných registrov skvele zvyšuje rýchlosť zásobníkových a smerníkových operácií.

1.9 Lab 1: Prostredie debuggera

Rodina Texas Instruments DSP používa spoločné uživateľské prostredie volané Source Debegger. Takmer všetky nástroje TMS320, vrátane simulátora TMS320C5x, používajú toto prostredie.

Táta kapitola vás bude viesť cez základné príkazy zdrojového debuggera.
Po prejdení celej kapitoly budete schopní:

- Nastaviť a manipulovať s oknami zobrazujúcimi premenné a dátové štruktúry
- Krokovať inštrukcie C a/alebo assemblera
- Nastaviť breakpointy a benchmark kód
- Zadávať príkazy debaggera cez príkazové menu, klávesnicu alebo myš

Poznámka

Tento prehľad je určený len na demonštrovanie prostredia debuggera. Nie je určený na učenie assemblera 'C5x alebo jazyka C. Prosím, nezastavujte sa nad tým, ak to zaberie veľa času (a úsilia). Jazyk assembler bude plne prezentovaný v nasledujúcich kapitolách.

1.9.1 Simulačné súbory a adresáre

Demo program `lab 1` je uchovaný v adresári . Aby ste zmenili adresár, napíšte

```
cd c:\dsp5 ↵
```

Demo programom je C súbor, ktorý jednoducho zavedie inkrementujúcu hodnotu do radu dátového typu. Je to užitočná platforma pre cvičenie príkazov a prostredia debuggera.

Odsek 1-11. Príklad programu pre Source Debugger

```
/*--------------------------------------------------------------------*/
/* Príklad programu pre Source Debugger                               */
/*--------------------------------------------------------------------*/

                /* deklarovanie globálnych premenných: int,         */
                /*          float array, zmiešaná typová štruktúra  */
int i;
float a[10];
struct  { int    i;
          float  j;
          int    k[4];
          int    *p;
        } example ;
void init ();

                /* inkrementovanie od 0 do 1000,                    */
                /* volanie init po každom pričítaní                 */
main () {
          int count;

          for (;;)
           for (count=0; count<1000; count++)
            init (count);
        }

                /* zavedenie všetkých globálnych premenných          */
                /*          s priebežným stavom hodnoty              */
void init (x)
          int x;
        {
          for (i=0; i<10; i++)
           a[i] = x;

          example.i = x;
          example.j = x;

          for (i=0; i<4; i++)
           example.k[i] = x;

          example.p = (int *) (0x0200 + x);
        }
```

1.9.2 Štart simulátora

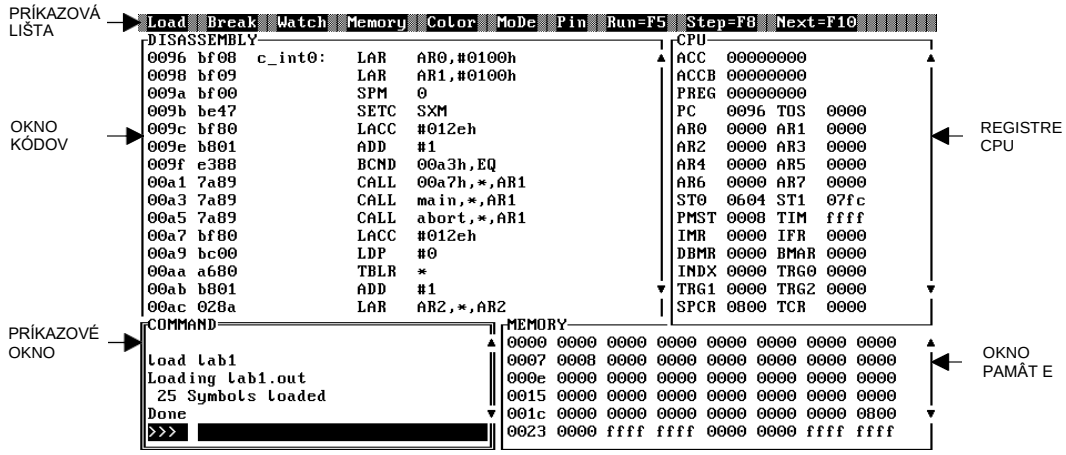
Na spustenie debuggera a zavedenie vášho výstupného súboru napíšte:

```
SIM5x lab1 ↵
```

Debugger predpokladá, že súbor, ktorý má byť zavedený, má príponu `.out`.

V druhej kapitole sa budeme učiť, ako vytvoriť výstupný súbor. Teraz by ste mali vidieť obrazovku debuggera, ktorá vyzerá podobne ako nasledujúci obrázok.

Obr. 1-12. Obrazovka debuggera



Poznámka

Ak sa počas tohto laboratórneho cvičenia dostanete do bodu, kde systém dlhšie neodpovedá, alebo je inak pokazený, môžete znovu zaviesť súbor napísaním `LOAD lab1` priamo v príkazovom riadku. Vo výnimočnom prípade môžete simulátor opustiť úplne napísaním `QUIT` a znova spustiť.

1.9.3 Výber aktívneho okna

Aktívne okno je zobrazené s vysvietenými okrajmi. Pre zmenu aktívneho okna stlačte:

```
<F6>
```

Niekoľkokrát opakujte cyklovanie cez aktívne okná.

1

Urobte okno DISASSEMBLY aktívnym. Môžete sa pohybovať v okne po obsahoch zobrazených kódov použitím kláves PgUp, PgDn, šipka hore, šipka dole. Pokúste sa pohybovať po okne.

Ak sa chcete vrátiť na jednotlivú značku alebo adresu, použite príkaz `addr nnnn`, kde `nnnn` je značka alebo adresa návratu. Napríklad napíšte:

```
addr c_int0 ↵
```

Presuňte sa na absolútnu adresu napísaním:

```
addr 0x0005 ↵
```

Presuňte sa na funkciu napísaním:

```
addr main ↵
```

1.9.4 Veľkosť a presúvanie okien

Môžete meniť veľkosť a presúvať ľubovolné okno. Urobte okno CPU aktívnym cez <F6> alebo napíšte:

```
win CPU ↵
```

Uistite sa, že CPU je napísané veľkými písmenami. Mená príkazov ako "win" nie sú citlivé na veľké alebo malé písmená, takže ich môžete vzájomne zamieňať. ***Ale meno okna musí byť napísané veľkými písmenami.***

Skratky pre mená okien sú dovolené.

Ak chcete okno zväčšiť alebo zmenšiť, napíšte:

```
size ↵
```

a potom šipkami určťe veľkosť. Potvrďte klávesou <Enter>.

```
↵
```

Ak chcete okno presunúť, napíšte:

```
move ↵
```

a potom šipkami okno presuňte. Obrys okna sa bude posúvať po obrazovke. Ak ste dosiahli želanú polohu, stlačte:

```
↵
```

Aby ste obnovili pôvodný stav obrazovky, použite príkaz "konfigurácie obrazovky" bez argumentov:

```
sconfig ↵
```

Aby ste zaviedli jednotlivú konfiguráciu obrazovky, môžete špecifikovať želaný súbor príkazom SCONFIG:

```
sconfig tc.clr ↵
```

Ak chcete uchovať konfiguráciu, použite funkciu Color, Save prístupnú stlačením <Alt> C , potom S, a napíšte želané m definovaná prípona, hoci .CLR (pre farbu) je prípona zvyčajne používaná u TI.

Ak napíšete opäť sconfig, vrátite sa do pôvodnej konfigurácie. Môžete použiť každú z týchto konfigurácií, alebo ľubovolné vlasné nastavenie, hocikedy používate debbuger.

1.9.6 Spustenie programu

Príklad programu začína vykonaním funkcie C: reset s návestím `c_int0`. Aby ste zmenili pozíciu v okne DISASSEMBLY, napíšte:

```
addr c_int0 ↵
```

Assemblerovský kód zobrazený na `c_int0` môže byť krokovaný stlačením:

```
<F8>
```


Skúste spustiť niekoľko inštrukcií stlačením `< F 8 >` a všimnite si zmeny hodnoty PC (v okne CPU) zodpovedajúce vykonávanej (vysvietenej) inštrukcii. Menený register a obsah pamäte je tiež vysvietený. Aby ste preskočili túto funkciu resetu, napíšte:

```
go main ↵
```

Všimnite si, že obrazovka mení zobrazenie C programu v okne FILE. Všimnite si tiež, že CPU registre nie sú dlhšie zobrazené. Okno CALLS ukazuje, ktorá C funkcia bola volaná.

Schopnosť vidieť zdrojový kód C v jeho pôvodnej forme je dôvod, prečo náš debbuger je označený ako "zdrojový" debbuger.

1.9.6 Okno Watch

Predpokladajme, že chcete vidieť hodnotu premennej C, kým krokujete program. Napíšte:

```
wa count ↵
```

Toto vytvorí okno watch s hodnotou premennej `count`. Hodnota zobrazená pre count nie je plnovýznamová, kým ešte nebola inicializovaná. Môžete skúsiť, že otvorenie okna watch pre premennú doposial nepoužitú bude generovať varovavanie.

Krokujte program (vykonanie jedného príkazu v čase) stlačením

```
<F8>
<F8>
```

Všimnite si, že premennej `count` bola pridelená hodnota nula. Teraz by ste mali byť na volaní funkcie `init()`. Stlačte:

```
<F8>
```

a pôjdete do funkcie. Všimnite si zmenu v okne CALLS.

Pridajte ďalšiu premennú do okna watch napísaním:

```
wa I ↵
```

Krokujte niekoľko ďalších C príkazov stlačením <F8>.

Aby ste videli prvok poľa, napíšte:

```
wa a[0] ↵
```

1

Všimnite si, že display automaticky ukazuje a[0] ako premennú s pohyblivou čiarkou. Debbuger zobrazuje hodnoty podľa definovaného typu.

Keď watch alebo display nie sú dlhšie používané na obrazovke, môžu byť zatvorené ich aktiváciou (použitím <F6> alebo kliknutím myši) a následným použitím príkazu na zatvorenie okna: klávesa <F4>. <F4> nie je použiteľná pre hlavné okná simulátora (CPU, MEM, Disassambly, atď.).

Zobrazenie poľa a štruktúr je veľmi dobrá vlastnosť debuggovania. Napíšte:

```
wa a ↵
```

Prímate chybovú správu Invalid watch expression, pretože ste dovolili sledovať len jednoduché skalárne hodnoty. Ak ste zabudli akým typom premennej je a, napíšte:

```
whatis a ↵
```

Aby ste zobrazili celé pole reálnych hodnôt, použite zobrazovací príkaz:

```
disp a ↵
```

Môžete chcieť presunúť okno DISP doprava obrazovky. Ak je tak, použite príkaz move a klávesy šipiek.

Zobrazte štruktúru s menom example napísaním:

```
disp example ↵
```

Táto štruktúra má štyri prvky s označením i, j, k a p. Všimnite si, že sú zobrazené podľa typu. Presuňte toto okno doprava hneď pod okno DISP: a.

Aby ste zobrazili obsah poľa example.k, presuňte kurzor vysvietenému riadku ukazujúcemu k: [...] a vybete ho stlačením:

```
<F9> alebo ľavé tlačítko myši
```

Novootvorené okno ukazuje prvky poľa. Ak toto malo inú štruktúru (namiesto poľa), ukáže sa to ako `k: { . . . }` . **Hranaté zátvorky vyznačujú polia a zložené zátvorky určujú štruktúry.**

Ak toto nové okno ukazujúce pole `k` je otvorené priamo na vrchu predchádzajúceho okna, môžete ho presunúť nižšie, aby `example` bolo viditeľné. Použite príkaz move so šipkami k presunutiu okna.

1.9.7 Inštrukcie STEP a NEXT

Teraz, keď zobrazovacie okná sú otvorené, začnime znovu od začiatku a odkrokujme niekoľko inštrukcií. Napíšte:

```
reset ↵
go main ↵
```

Stlačte:

```
<F8>
```

Pokračujte vykonávaním príkazov opakovaným stláčaním `<F8>` . Všímajte si, ako sa menia hodnoty v oknách watch a display. Pokračujte krokovaním cez funkciu `init`, kým sa nevráti na funkciu `main`. Ak si neprajete vidieť zvyšok funkcie v krokovom móde, môžete funkciu ukončiť a okamžite sa vrátiť vložením:

```
ret ↵
```

Poznámka

Ak ste v tomto bode neboli v podfunkcii, simulátor sa nikdy nevráti, a preto sa nikdy nezastaví. K zastaveniu simulátora v takomto prípade jednoducho stlačte `<Esc>`.

Predpokladajme, že chcete krokovať bez poznania detailov jednotlivých volaní funkcie. Môžete krokovať cez volania funkcie použitím:

```
Next ↵
```

Poprípade môžete stlačiť `< F 1 0 >` . Všimnite si, že ďalší C príkaz b vykonaný bez zobrazenia volaní funkcie. (Volania funkcie nie sú preskočené,
sú len vykonané v krokovom móde.)

Obidva, step príkaz `<F8>` aj next príkaz `<F10>` môžu byť vykonanné u príkazového riadku spolu s argumentom špecifikujúcim číslo vykonania
inštrukcie.

Napríklad napíšte:

```
step 10000 ↵
```

Na zastavenie vykonávania, stlačte:

```
<Esc>
```

Poznámka

Takisto ako numerické príklady, môžete použiť boolovské výrazy spolu s príkazom `step`; napr. `step (AR0 !=0)`

Ak vykonávate funkciu `init ()` a chcete sa vrátiť, napíšte:

```
ret ↵
```

Teraz vyskúšajte príkaz `next` so súčtovou hodnotou:

```
next 10000 ↵
```

Môžete ísť späť a pozorovať operáciu po krokoch.

K zastaveniu vykonávania stlačte

```
<Esc>
```

a budete vidieť odkaz `User halt` zobrazený v príkazovom okne.

1.9.8 Debuggovanie assemblera a programov C

Táto časť výuky predpokladá, že ste prešli prvou časťou tohoto prehľadu a máte zavedený program `lab1.out` v debuggeri.

K opätovnému začatiu vykonávania, napíšte:

```
reset ↵
go main ↵
```


1.9.8.1 Zmiešaný mód

K debbugovaniu v zmiešanom móde, ktorý dovoluje pozorovať inštrukcie assemblera a C súčasne, napíšte:

```
mix ↵
```

Mali by ste vidieť zdrojový kód C a k nemu zodpovedajúci kód assemblera. Okno DISASSEMBLY ukazuje vysvietené pamäťové miesta, ktoré sú spojené s práve prebiehajúcim príkazom C.

Môžete premiesniť a zmeniť veľkosť okien display a watch, aby ste videli okná CPU a REGISTER. Je vhodné vybrať (resetnúť) okno watch použitím príkazu:

```
wr ↵
```

Skúste krokovať opakovaným stláčaním:

```
<F8>
```

Všimnite si, že inštrukcie assemblera sú krokované. Ak ste v práve vykonávanej funkci `init ()` a chcete sa vrátiť, napíšte:

```
ret ↵
```

Vyskúšajte príkaz `next` opakovaným stláčaním:

```
<F10>
```

Pokračujte, až kým nespozorujete, že inštrukcia `CALL init` je ukončená.

Ak chcete krokovať príkazy C počas zmiešaného módu, zakaždým napíšte:

```
cstep ↵
```

alebo

```
cnext ↵
```

Tak isto ako `step` a `next`, môžete vykonať určité množstvo inštrukcií. Napríklad:

```
cstep 10 ↵
```

vykoná 10 príkazov jazyka C.

1.9.8.2 Mód ASM

Ak sa zaujímate len o debuggovanie assemblerovského programu, môžete zapnúť tento mód napísaním:

```
asm ↵
```

Všimnite si, že okno zobrazujúce dátové štruktúry C zmizne. Toto je vhodná cesta vyčistiť obrazovku, ak chcete pozorovať hodnoty CPU registra alebo zobraziť obsah pamäte. Vyskúšajte krokovanie opakovaným stláčaním:

```
<F8>
```

a pozorujte zmeny hodnôt registra v okne CPU. Zmenené hodnoty sú vysvietené, takže si môžete všimnúť, kedy zmena nastala.

Do MIX módu môžete ísť napísaním:

```
mix ↵
```

Všimnite si, že okno DISP sa znovu objavilo.

1.9.8.3 Opakovanie módov

V zhrnutí: sú tu tri módy operácie.

- Zmiešaný mód (príkaz `mix`) zobrazuje assembler a C (ak C zdroj existuje).
- Assembly mód (príkaz `asm`) zobrazuje len assemblerovský kód.
- C mód (príkaz `c`) automaticky prepína zobrazenie C do assemblera podľa toho, aký typ zdrojového kódu je vykonávaný.

1.9.9 Breakpointy a benchmarking

Opäť naštartujte váš program a vykonajte prvé volanie funkcie `init()`. Napíšte:

```
reset ↵
mix ↵
go init ↵
```


K nastaveniu breakpointu musíte vždy použiť príkaz `ba xxxx`, kde `xxxx` je absolútne miesto pamäte alebo platné návestie. Táto metóda vyžaduje poznanie adresy (alebo návestia). Napríklad napíšte:

```
ba init ↵
```

Toto nastaví breakpoint na vstupný bod funkcie. Všimnite si, že inštrukcia sa vysvieti, keď je breakpoint nastavený.

Aby ste mohli listovať nastavenými breakpointami, napíšte:

```
bl ↵
```

Aby ste vymazali všetky breakpointy, použite príkaz breakpoint reset. Napíšte:

```
br ↵
```

Overte proces opätovným listovaním. Napíšte:

```
bl ↵
```

Dodatok k príkazu `ba`. Môžete oveľa jednoduchšie vybrať a nas inštrukciu pre breakpoint. Poďte si vyskúšať túto techniku. Najprv vyberte okno FILE. Môžete použiť klávesu `<F6>` k cyklovaniu okien, kým nie je okno FILE aktívne. Rýchlejšou cestou je napísať príkaz `win`. Napíšte (buďte si istí, že píšete FILE veľkými písmenami):

```
win FILE ↵
```

Teraz presuňte kurzor v okne FILE na prvý príkaz `for` vo funkcii `init`. Použite klávesy šipiek k presúvaniu hore a dole. Aby ste vybrali inštrukciu pre breakpoint, stlačte:

```
<F9>
```

Príkaz `for` by mal byť vysvietený, aby ukazoval, že breakpoint je nastavený. K odstráneniu breakpointu opäť stlačte `<F9>` . Funkčná klávesa `<F9>` (alebo

kliknutie myšou) urobí to isté, ako prepínanie medzi vyznačením alebo odznačením breakpointov.
Aby sa vykonal váš program až po breakpoint, napíšte:

```
run ↵
```

Program by sa mal zastaviť na breakpointe. Ak breakpoint nebol dosiahnutý, stlačte <Esc> a skontrolujte, či breakpoint bol nastavený (použite príkaz `bl`
alebo pozrite na okno FILE/DISASSEMBLY, či inštrukcia je vysvietená).

K použitiu predchádzajúceho príkazu debuggera (pre lenivých pisárov) stlačte:

```
<Tab>
```

Všimnite si, že stlačenie <Tab> spôsobí návrat do predchádzajúceho spusteného príkazu. Stlačenie spôsobí, že príkaz bude opäť vykonaný. Môžete tiež prechádzať späť cez všetky predchádzajúce spustené príkazy stlačením <Tab>. Stlačením <Shift><Tab> vás presunie na začiatok tohoto príkazového bufferu.

Predpokladajme, že máte stále nastavený breakpoint na inštrukciu `for`. K odmeraniu času (benchmarking) potrebného na vykonanie programu od jedného breakpointu k druhému, potrebujete nastaviť druhý breakpoint. Choďte priamo a vyberte ďalšiu inštrukciu pre breakpoint použitím <F9>, kliknutím myši alebo príkazom `ba`. Na benchmarking, napíšte:

```
run ↵
runb ↵
? clk ↵
```

Príkaz `run` vykoná prvý breakpoint. P "bež-s-benchmarkingom". Príkaz `?` povie debuggeru, aby zhodnotil nasledujúci výraz C a zobrazil výsledok. Premenná debuggera `clk` je správna len po príkaze `runb` a je nastavená na hodnotu počtu hodinových cyklov medzi príkazmi `run` a `runb`.

1.9.10 Hodnotenie výrazov

K zhodnoteniu C výrazov, môžete použiť príkaz `?`. Toto je jedna cesta, ako zmeniť hodnoty registrov, pretože výrazy C môžu mať vedľajší efekt ako napríklad priradenie. Napíšte:

```
? pc ↵
```

Mohli by ste tiež vidieť zobrazenú hodnotu `pc`. K zmene terajšieho `pc` napíšte:

```
? pc = main
```

K zmene registra, napíšte:

```
? ar0 = 0
```

K zhodnoteniu výrazu bez zobrazenia výsledku v okne COMMAND, použite príkaz `eval` namiesto príkazu `?`. Napíšte:

```
eval pc = 0 ↵
eval pc = main ↵
```

1.9.11 Zobrazenie súborov

V okne FILE môžete zobraziť ľubovolný súbor. Napíšte:

```
file siminit.cmd ↵
```

Mali by ste vidieť zobrazený inicializačný príkazový súbor debuggera. V tomto bode môžete ísť späť k debuggovaniu a predchádzajúci zdrojový C súbor bude automaticky zobrazený, keď začnete vykonávať inštrukcie.

S COMMAND oknom debuggera môžete vykonávať príkazy ako v DOSe, napr. prezrieť a zmeniť súčasný adresár. Použite príkaz `dir nnnn`, kde `nnnn` je meno adresára, k zobrazeniu zoznamu adresárov. Napíšte:

```
dir ↵
```

k zobrazeniu terajšieho adresára.

Príkaz `cd nnnn`, kde `nnnn` je nové meno adresára, zmení terajší adresár.

Aby ste zmazali okno COMMAND, napíšte:

```
cls ↵
```

Niektré ďalšie príkazy sú:

- `quit`, ktorý opustí debugger a vráti vás späť do DOSu
- `restart`, ktorý nastaví PC na vstupný bod kódu

1.9.12 Roletové menu

Aby ste spustili roletové menu z lišty na vrchu obrazovky, stlačte <Alt><key> , kde <key> je vysvietené písmeno menu (L, B, W, M, alebo D). Keď je menu zobrazené, môžete vždy vykonať príkaz napísaním určitého písmena, alebo použitím šipiek k presunutiu výberovej lišty na želaný príkaz a stlačením <Enter>. Napríklad, stlačte:

<Alt>L

potom opakovane stlačte klávesu pravej šipky k zobrazeniu roletových menu.

1.9.13 Použitie myši

Použitím myši je prístup k menu a oknám oveľa jednoduchší ako pomocou klávesnice a šipiek. K vybratiu aktívneho okna presuňte kurzor myši na okraj okna, potom stlačte a uvoľnite ľavé tlačítko. (Stlačenie a rýchle pustenie tlačítka myši sa nazýva kliknutie. Kliknutie ľavým tlačítkom myši zodpovedá stlačeniu <F9>.)

Poznámka

Buďte opatrní. Ak kliknete, kým ste nad prvkom okna, môžete nastaviť breakpoint (ak ste vo FILE alebo DISASSEMBLY okne), alebo vyberiete register alebo pamäťové miesto, ktoré sa bude meniť.

Presúvanie okna je ľahšie s myšou. Umiestnite kurzor myši na vrchnú lištu okna. Stlačte a držte ľavé tlačítko, presuňte myš, čím potiahnete okno na novú pozíciu, potom pustite ľavé tlačítko.

Zmena veľkosti okna je tiež jednoduchšia s myšou. Umiestnite kurzor myši na tlačítko veľkosti okna (v pravom dolnom rohu), stlačte a držte ľavé tlačítko, posuňte myš podľa toho, či si želáte okno zväčšiť alebo zmenšiť, potom ľavé tlačítko pustite.

1.9.14 Zmena veľkosti zobrazenia

Ak máte displej schopný znakového rozlíšenia väčšieho než 80x25, môžete dať na obrazovku viac informácií použitím voľby `-b[bbbb]` debuggera, keď ho vyvolávate. Skúsme to. Opustite debugger napísaním:

```
quit ↵
```

Priamo z DOSu otvorte:

```
sim5x lab1 -bb
```

a mali by ste dostať displej, ktorý zobrazuje viac detailov, ale môže tiež spôsobiť viac problémov s očami. Väčší monitor vám dovoľuje využívať výhody vysokorozlišovacieho módu zdrojového debuggera.

Zapnutie `-bb` vytvára 50-riadkový displej. Ďalšia voľba `-b` ponúka strednú veľkosť, 43-riadkový displej. Vaša preferovaná veľkosť displeja môže byť uchovaná v konfigurácii obrazovky ako `int.clr` s príkazom `ssave` opísaným skôr. Potom je potreba explicitného použitia voľby `-b` eliminovaná.

1.9.15 Batch operácia debuggera

Príkazy debuggera môžete vykonať z batch súboru. To môže byť užitočné, ak je určitý sled príkazov, ktoré chcete spustiť vždy, keď štartujete debugger pre aplikácie. Meno súboru by malo mať príponu `.log`. Aby sa vykonal súbor `.log` kým ste v debuggeri, použite príkaz `take <filename>.log` Napríklad, skúste príkazový súbor:

```
take lab1.log ↵
```

Gratulujeme, práve ste skončili. Aby ste opustili debugger, stlačte:

```
<Esc>
```

Napíšte:

```
quit ↵
```

Odsek 1-13. Rýchle odkazy simulátora

Rýchle odkazy simulátora

Organizácia okien

Vybratie okna
- F6 rotuje k ďalšiemu oknu
- WIN <name> vyberie okno <name>
- Kliknite na rám želaného okna
- F4 zavrie vybrané okno

Presúvanie vo vnútri okna
- Šipka hore / Šipka dole
- Page Up / Page Down
- Kliknite na šipku rámu okna
- Pre DISASSEMBLY; napíšte ADDR <hodnota>
- Pre okno MEMORY; napíšte MEM <hodnota>

Presúvanie okna
- Kliknite na vrch rámu; potiahnite na nové miesto
- Napíšte MOVE a použite šipky/napíšte koordináty

Zmena veľkosti okna
- Kliknite na pravý roh; potiahnite na nové miesto
- Napíšte SIZE a použite šipky/ napíšte koordináty
- ZOOM kliknite na ľavý horný roh
- UNZOOM kliknite opäť na ľavý horný roh

Konfigurácia obrazovky
- SCONFIG <name> zavedenie konfigurácie
- SSAVE <name> uloženie konfigurácie <name>

Módy
- ASM zobrazí ASM info alebo <Alt> D,A
- C zobrazí C info alebo <Alt> D,C
- MIX zobrazí ASM aj C alebo <Alt> D,M

Vstup / Výstup

- SIM5X <file> štartuje simulátor s <file>.out
- SIM5X -bb vysokorozlišovací mód
- QUIT opustenie simulátora
- SYSTEM môže vykonať DOS shell

Spustenie programu

Reset
- Napíšte RESET nastaví PC na nulu
- Napíšte RESTART vráti sa na "spúšťací bod"

Krokovanie
- F8 alebo napíšte STEP pre jeden krok
- F10 alebo napíšte NEXT zhustí podrutiny
- Napíšte STEP <n> pre <n> krokov
- Napíšte NEXT <n> pre <n> nextov

Spustenie
- RUN beží kým nie je <Esc> alebo breakpoint
- RUNB beží s benchmarkom
- GO <návestie> beží po <návestie>

Watch a breakpointy

Operácia	Watch	Breakpoint
• ADD	WA	BA
• RESET	WR	BR
• LIST	WL	BL
• DELETE	WD #	BL #

alebo skratkové klávesy, alebo kliknutie myšou

Ďalšie operácie

- ? <návestie> zobrazuje hodnotu <návestie>
- ? <návestie> = <n> natiahne <návestie> s <n>
- súbor <meno> natiahne súbor <meno> do okna
- TAB scrolluje prioritné príkazy
- SHIFT TAB scrolluje pod príkazy
- F9 alternuje kliknutie myšou
- TAKE <meno> simulátor batch file
- LOAD <meno> download file <meno>

1.10 Opakovanie

Odsek 1-14. Opakovanie

1. Opíšte vnútorné pamäťové moduly TMS320C5x.
2. Koľko sériových portov je na TMS320C5x ?
3. Čo je cieľom PLU ?
4. Čo je cieľom jednotky JTAG ?
5. Ktoré sú dve hlavné zbernice na TMS320C5x ?

Kapitola 2

COFF TOOLS: PROSTREDIE JAZYKA SYMBOLICKÝCH ADRIES

Odsek 2-1 Učebné ciele:

Na záver tejto lekcie by ste mohli porozumieť a používať COFF tools pre súbory v jazyku symbolických adries (JSA).

Teda mali by ste vedieť:

- popísať úžitok COFF tools
- používať jednoduché direktívy, symboly a syntax COFF-u
- vybrať a používať vhodné typy sekcií v ASM súboroch
- identifikovať zložky a " popisovače" v príkazovom súbore LINKER-a
- zmeniť príkazové súbory LINKER-a podľa toho, o aký súbor, pamäť a možnosti výberu sekcií ide
- na vytvorenie objektového a listing súboru vyvolať assembler
- na vytvorenie vykonateľného výstupného súboru vyvolať linker s použitím .CMD súboru, ktorý usmerňuje proces

2.1 Pojem COFF

V úsilí štandardizovať proces vývoja softwaru (SW), TI vyselektoval SPOLOČNÝ FORMÁT OBJEKT-ového SÚBOR-u - COFF. COFF mániekoľko rysov, ktoré ho robia vel'mi užitočným a silným systémom pre vývoj SW-u. Je najužitočnejší, ked´ je vývojová úloha rozdelená medzi niekoľko programátorov.

Každý súbor, nazývaný *modul*, môže byť napísaný nezávisle, vrátane špecifikácie všetkých zdrojov potrebných pre správnu činnosť modulu.

Moduly sú písané v mnemonikách na úrovni assemblera s použitím hocijakého textového editora schopného pripraviť jednoduchý ASCII súbor. Očakávaná prípona súboru je .ASM, ktorá je od slova assembly-skladať.

Ďalej, každý z týchto modulov je*prekladaný* preto, aby sa preložil z JSA do binárneho zobrazenia, ktorý ´320 rozpoznáva. Listing súbor je produkovaný nezáväzne, aby dokumentoval výsledok prekladania.

Veľa modulov môže byť spájaných tak, aby vytvárali kompletný program. Linker je SW-ový nástroj schopný výkonne prideliť každému modulu v systéme zdroje prístupné na ´320-ke. Linker môže odkázať na príkazový .CMD súbor, ktorý identifikuje mená všetkých vstupných a výstupných súborov, zdroje prístupné na ´320 a tiež kam majú ísť rozličné sekcie každého modulu. Výstupy linkovania môžu obsahovať linkovaný objektový súbor .OUT, ktorý beží na ´320-ke a .MAP súbor, ktorý identifikuje, kde bola umiestnená každá linkovaná sekcia.

Vysoká úroveň modularity a prenosnosti vyplývajúca z tohto systému, zjednodušuje procesy overovania, ladenia a údržby. Proces vývoja COFF-u je bližšie prezentovaný v nasledujúcich paragrafoch.

Koncepcia COFF tools je umožniť, aby modulárny vývoj SW-u išiel nezávisle od záležitostí hardwaru (HW). Jednotlivý súbor v JSA je písaný tak, aby vykonával jednotlivú úlohu, a aby mohol byť linkovaný s niekoľkými inými úlohami. Tým je dosiahnuté, že celkový systém je komplexnejší .

Písaním programu v štruktúrovanej-modulárnej forme je možné vyvíjať program niekoľkými, paralelne pracujúcimi ľuďmi, čím sa skráti vývojový cyklus. Ladenie a zlepšovanie programu je rýchlejšie, pretože sa nepracuje nad celým systémom, ale nad jeho časťami. Teda nové systémy môžu byť vyvíjané rýchlejšie, ak sa v nich môžu použiť už skôr vyvinuté moduly.

Ak je program vyvíjaný nezávisle od záležitostí HW-u, rastie úžitok modularity, a to preto, lebo to dovoľuje programátorovi, aby sa sústredil na tvorbu programu a neplytval čas obsluhou pamäti a premiestňovaním programu,ak niektoré jeho zložky rastú, alebo sa zmenšujú. Linker je vyvolaný na to, aby prideľoval systémový HW modulom, ktoré tvoria systém. Zmeny v niektorom, alebo všetkých moduloch pri re-linkovaní (opakovanom linkovaní), vytvárajú nové prerozdelenie HW-u, s vyvarovaním sa možnosti vzniku konfliktov pri prideľovaní pamäťových zdrojov.

2.2 Syntax COFF-u

2.2.1 Formát assemblera

Formát pre inštrukcie, alebo direktívy v JSA je:

```
[<label>[:]]   <mnemonic>   [<operand list>]   [;<comment>]
```

< l a b e l > je nezáväzný, ale ak je použitý, musí začínať v prvom stĺpci. Dvojbodka nasledujúca po < l a b e l > je tiež nepovinná, assembler ju aj tak ignoruje. Nepoužívajte dvojbodku, keď odkazujete na takto vytvorené návestia.
<mnemonic> je inštrukcia v JSA, direktíva assemblera, alebo meno makra. Mnemoniky musia byť od ľavého stĺpca oddelené medzerou, aby neboli považované za návestie.
<operand list> je zoznam operandov, ktoré nasleduje po poli mnemoník. Operand môže byť konštanta, symbol, alebo ich kombinácia vo výraze. Operandy musia byť oddelené čiarkami.
< c o m m e n t > sa môže objaviť hocikde v riadku, ak je pred ním bodkočiarka. Ak je v prvom stĺpci (*), celý riadok je pokladaný za komentár.

2.2.2 Konštanty

Assembler používa 7 typov konštánt uvedených v nasledujúcej tabuľke:

tabuľka 2-2. Typy konštánt

Typ	Príklad
Binárne číslo	11111000b alebo 11111000B
Osmičkové číslo	226q alebo 226Q
Desiatkové číslo	1234 alebo +1235/-1234 (+ je štandardné)
Hexadecimálne číslo	37ACh alebo 37ACH. Táto konštanta sa nemôže začínať písmenom, začínajte ho 0. Napr. 0FFFFh, nie FFFFh.
Číslo s plávajúcou čiarkou	1.623e-23 (znamienko a desatinná čiarka sú voliteľné), veľké/malé "e" povolené.
Znaková konštanta	´D´
Znakový reťazec	"toto je reťazec"

tabuľka 2-3 Operátory používané vo výrazoch (priorita)

Skupina 1(najvyššia priorita) vyhodnocovanie sprava doľava		Skupina 3 vyhodnocovanie zľava doprava	
+	unárny plus (kladný výraz)	+	sčítanie
-	unárny mínus (záporný výraz)	-	odčítanie
~	jednotkový doplnok	^	bitový exlusive-OR
		|	bitový OR
		&	bitový AND
Skupina 2 vyhodnocovanie zľava doprava		**Skupina 4(relačné operátory) vyhodnocovanie zľava doprava**	
*	násobenie	<	menší než
/	delenie	>	väčší než
%	modulo	<=	menší než alebo rovný
<<	posuv vľavo	=	(==) rovná sa
>>	posuv vpravo	|=	nerovná sa

Poznámka: Operátory v ( ) sú alternatívne.

2.2.3 Symboly

Symboly sú reťazce z max. 32 znakov (A-Z, a-z, $ a _). Symbol nemôže začínať číslom 0 až 9.
$ vyjadruje hodnotu v počítadle aktuálnej polohy (adresy) v sekcii.
Assembler má predurčené nasledujúce:

mená registrov (vrátane)	AR0-AR7
adresy portov	PA0-PA15

Použitie direktívy .mmregs (dole) zväčšuje počet horeuvedených definovaných symbolov.

2.2.4 Základné direktívy assemblera

.set

Táto direktíva sa používa na definovanie časovej konštanty prekladania. Hodnota konštanty ide do poľa operandov. Návestie direktívy .set môže byť použité na mieste, kde sa smie použiť konštanta. Direktíva *set* môže byť použitá na definovanie nového mena registra. Napríklad:

```
LENGHT  .set          20 h        ;konštanta  "lenght"
                                  ;sa rovná 20h

COUNTER .set          AR0         ;"counter" je to isté
                                  ;ako AR0
```

POZNÁMKA:
Direktíva .set nevytvorí nový objekt v rámci TMS320, ale jednoducho len definuje nové meno registra.

.global

Táto direktíva vyhlasuje, že symbol je definovaný ako globálny symbol. Symbol je definovaný buď v rámci modulu; alebo sa v module vyskytuje, ale definovaný je niekde inde. Linker rieši všetky odkazy na globálne symboly.
Napríklad:

```
            .global       BEGIN     ;BEGIN je globálny symbol
BEGIN       .set   $                ;BEGIN má hodnotu poèítadla
                                    ;aktuálnej adresy v sekcii
```

.mmregs

Táto direktíva dovoľuje, aby assembler rozpoznal mená pamäťovo mapovaných registrov. Stačí to raz špecifikovať v zdrojovom súbore.

.option

Táto direktíva povoľuje výstupný výpis assemblera. V tejto príručke budeme najčastejšie používať:

```
.option x
```

,ktorá vytvorí krížovú referenciu symbolov na konci výpisu. Viac informácii nájdete v *Assembly Language Tools User´s Guide.*

.title

Táto direktíva umožňuje mať v hlavičke každej strany titul obsahujúci do 50 znakov. Titul treba uzavrieť do dvojitých úvodzoviek. Napríklad:

```
.title   "Lab 2- Addressing Exercise"
```

.end

Táto direktíva by mala byť poslednou v zdrojovom súbore písanom v JSA. Je nepovinná.

.word

Na definovanie dátovej tabuľky v pamäti ROM použite direktívu .data, ktorá vydá pokyn assembleru, aby umiestnil hodnoty do .data sekcie. Direktíva .data je všeobecne používaná pre inicializovanú pamäť.

Na vloženie 16-bitových celých čísiel do pamäti (nezávisle od sekcie) sa používa direktíva `.word`. Príklad dole umiestni 16-bitové hodnoty 1,2,3 do troch po sebe nasledujúcich pamäťových miest.

```
.word       1,2,3
```

.space

Na rezervovanie *bitov* a inicializáciu tabuľky nulami sa používa direktíva .space. Príklad dole je ekvivalentný s tridsiatimi direktívami `.word 0`.

```
.space      30*16
```

2.3 Softwarové sekcie

Najmenší premiestniteľný blok programu v rámci zdrojového súboru sa nazýva *sekcia*. Všetky programy a dáta v zdrojovom súbore sú skladané do sekcií, ako je to určené direktívami assemblera. Všeobecne sekcie sa môžu objaviť v hocijakom poradí a smú byť podľa potreby opakované.

obrázok 2-4. Sekcie

```
                .text
[návestie:]     MNEMONIKA    operand, operand              ; komentár

                .data
[návestie:]     .word        hodnota, hodnota, hodnota     ; komentár

                .bss         symbol, veľkosť, contig. flag

                .sect        "meno sekcie"

[návestie:]     .label       symbol

[návestie:]     .usect       "meno sekcie", veľkosť, contig. flag
```

Tieto direktívy umožňujú:

- Skladať dáta a programi do špecifikovaných, inicializovaných sekcií
- Rezervovať priestor v pamäti pre neinicializované premenné

Nepomenované sekcie

.text

Toto je štandartná sekcia, ktorá zvyčajne obsahuje vykonateľný zdrojový program. Fyzické umiestnenie v programovej pamäti je vykonané počas linkovania podľa príkazového súboru linkera. Ako je špecifikovaná poloha v pamäti bude vysvetlené neskôr, keď sa bude diskutovať o príkazovom súbore linkera.

.data

Táto sekcia obsahuje inicializované dátové premenné v rámci systémovej pamäti. Direktíva .data povie assembleru, že nasledujúce dátové premenné majú byť umiestnené v pamäti do .data sekcie, ako je to definované príkazovým súborom linkera. Preto .data direktíva je používaná na to, aby oznámila assembleru, že hodnoty dát nasledujúce za direktívou sú predinicializované. Ako je určená poloha začiatku .data sekcie v pamäti bude vysvetlené neskôr.

.bss

Táto direktíva je zvyčajne používaná na rezervovanie priestoru v systémovej pamäti pre neinicializované premenné. Syntax direktívy `.bss` je:

```
.bss        <symbol>,<size>
```

`<symbol>` odpovedá adrese prvého miesta rezervovaného direktívou `.bss`;
t.j. odpovedá to menu premennej, pre ktorú ste rezervovali miesto.
`<size>` je počet rezervovaných, po sebe idúcich slov.

Užívateľom definované pomenované sekcie

.sect

Táto direktíva sa používa na definovanie *pomenovanej sekcie* inicializovanej pamäti. Len čo je definované meno pre sekciu, sekcia môže byť linkovaná tak, aby začínala na špecifickom mieste v pamäti.Syntax je:

```
[label]      .sect         "<meno sekcie>"
```

`<meno sekcie>` je symbol limitovaný do 8 znakov, uzavretý do dvojitých úvodzoviek. Použitím `.sect` v strede hocijakej sekcie .text alebo .data môžete Definovať pomenovanú sekciu. Ak sa objaví návestie, jeho hodnota začiatočnou adresou sekcie.

.usect

Táto direktíva sa používa na definovanie pomenovanej sekcie v neinicializovanej pamäti, zvyčajne aby držala hodnoty dát v RAM. Syntax je:

```
<label>      .usect         "<meno sekcie>",<size>
```

`<meno sekcie>` definuje meno pre sekciu, je limitované ôsmimi znakmi. `<size>` pridelí určitý počet slov. `<label>`, ktoré je pri direktíve `.usect` povinné, je assemblerom spracované ako adresa prvej hodnoty v tabuľke.

Direktíva `.usect` je podobná `.bss`, pretože obe sú použité ku rezervovaniu neinicializovanej pamäti. Ale direktíva `.usect` obsahuje meno sekcie, ktorej umiestnenie je riešené príkazovým súborom linkera inak od prípadu k prípadu.

2.3.1 Príkazový súbor linkera

Súbor link. cmd pripravuje informáciu pre linker. Táto informácia obsahuje nasledujúce položky.

obrázok 2-5. Informácie pripravené súborom LINK.CMD

Mená súborov

- vstupné súbory
  - -objektový
  - -archívne súbory
  - -iné súbory
- mapový súbor
- výstupný súbor

Popisovač pamäti

- popísať každú zložku pamäti
  - -poloha
  - -veľkosť
  - -atribúty
- dovoliť definovať viacnásobné pamäťové mapy

Popisovač sekcií

- usmerňuje softwarové sekcie do pomenovaných pamätí
- umožňuje rozlíšenie per-file
- dovoľuje separovať LOAD/ RUN umiestnenie

Nasledujúce príkazové súbory sú používané na linkovanie väčšiny programov v tejto príručke. Slúžia ako vzory pre iné príkazové súbory linkera, ktoré si môžete vytvoriť.

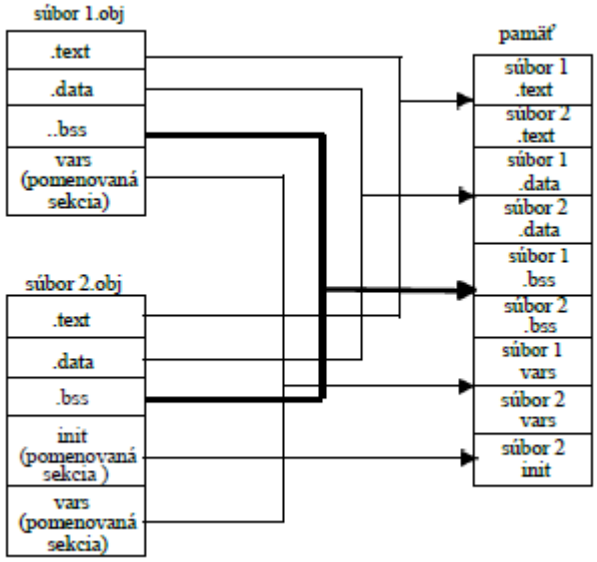
obrázok 2-6. Príkazový súbor linkera-C50

```
        /*      TMS320C50  LINKER COMMAND FILE*/
        /*      9k RAM > Prog. Mem,  BO > Data Mem, uComputer Mode*/

MEMORY    {
  PAGE  0:                                      /*priestor programu */
        VECS          :o=0000H; l=0040H         /* Std. vektory*/
        PROG          :o=0040H; l=07C0H         /* 2k ROM*/
        RAM           :o=0800H; l=2400H         /* 9k RAM blok*/
        EXT           :o=2C00H; l=0D400H        /* externá pam.*/
  PAGE  1:                                      /*priestor dát*/
        REGS          :o=0000H; l=0060H         /* MMR-e*/
        BLKB2         :o=0060H; l=0020H         /* BLK B2*/
        I_RAM         :o=0100H; l=0400H         /* BLK B0 & B1*/
        EXT           :o=0800H; l=0F800H        /* externá pam.*/
        }
SECTIONS  {
        .text        :{} > PROG PAGE 0          /* Kód...*/
        .data        :{} > PROG PAGE 0          /* Tabul'ky...*/
        .bss         :{} > I_RAM PAGE 1         /* Premenné...*/
        vectors      :{} > VECS PAGE 0          /* Vektorová tabul'ka*/
        }
```

obrázok 2-7. Príkazový súbor linkera-C51

```
        /*       TMS320C51    LINKER COMMAND FILE*/
        /*       1k  RAM >  Prog,   BO .  Data Mem,  uComputer  Mode*/

MEMORY    {
  PAGE 0:                                       /*priestor programu*/
        VECS          :o=0000H, l=0040H         /* Std. vektory*/
        PROG          :o=0040H, l=1FC0H         /* 8k ROM*/
        RAM           :o=2000H, l= 400H         /* 1k RAM blok*/
        EXT           :o=2400H, l=0DC00H        /* externá pam.*/
  PAGE 1:                                       /*priestor dát*/
        REGS          :o=0000H, l=0060H         /* MMR-e*/
        BLKB2         :o=0060H, l=0020H         /* BLK B2*/
        I_RAM         :o=0100H, l=0400H         /* BLK B0 & B1*/
        EXT           :o=0800H, l=0F800H        /* externá pam.*/
        }
SECTIONS  {
        .text        :{} > PROG PAGE 0          /* Kód...*/
        .data        :{} > PROG PAGE 0          /* Tabul'ky...*/
        .bss         :{} > I RAM PAGE 1         /* Premenné...*/
        vectors      :{} > VECS PAGE 0          /* Vektorová tabul'ka*/
        }
```

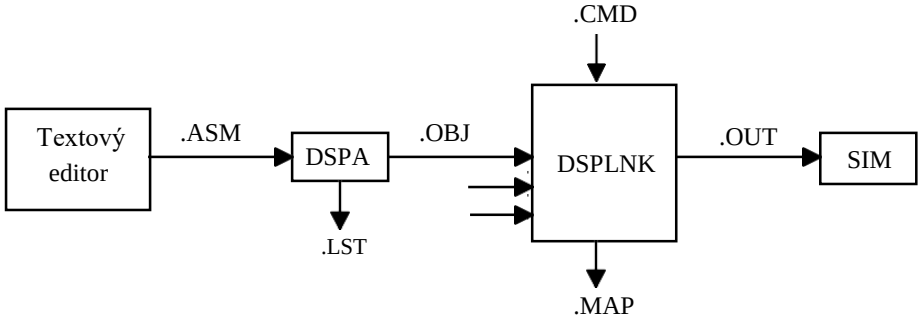
obrázok 2-8. Logické bloky linkovania



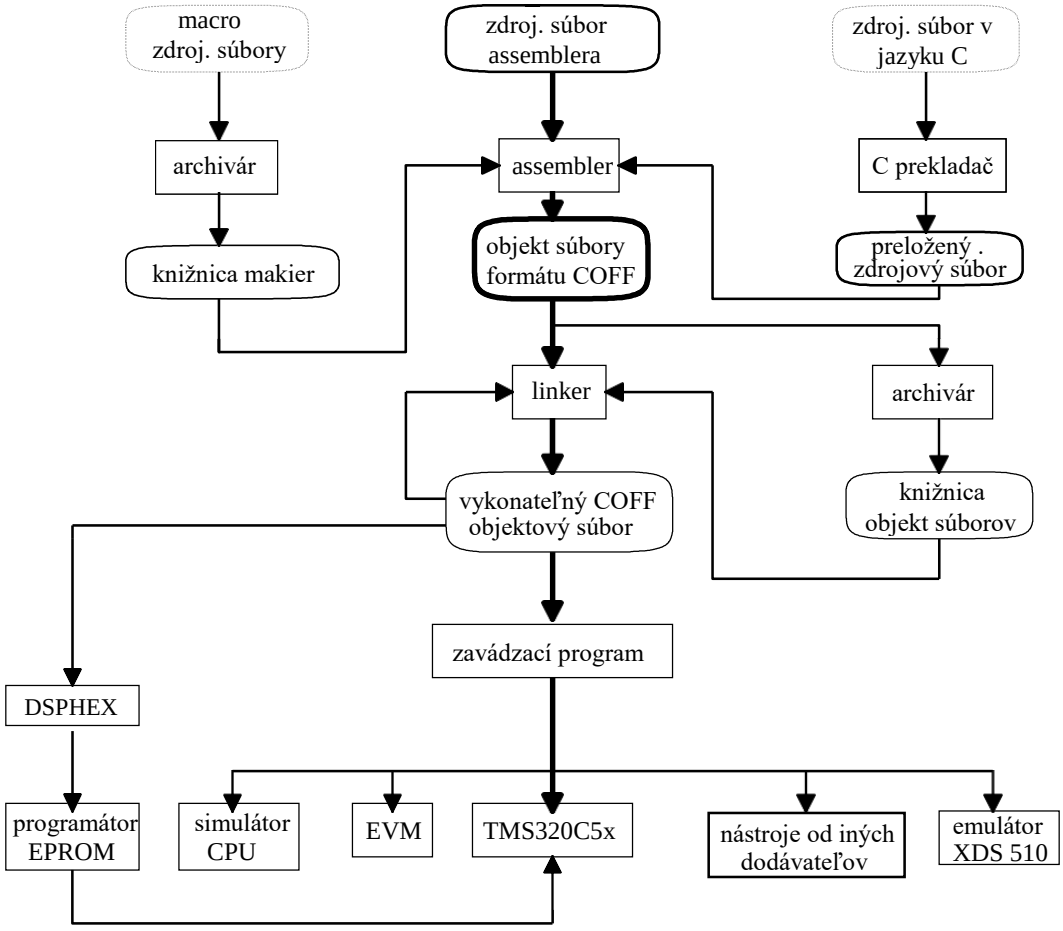
2.4 COFF Tools

COFF vývojový proces softwaru pozostáva z niekoľkých úloh, ako je to zakreslené v nasledujúcom obrázku. Každá z týchto úloh je rozobratá v nasledujúcich častiach.

obrázok 2-9. Vývojový proces softwaru COFF



obrázok 2-10. Vývojové prostredie TMS320C5x



2.4.1 MS-DOS editor

Používať budeme textový editor MS DOS ® Editor MS-DOS6.x. Je to ľahko použiteľný, systémom menu ovládaný program. Pre navigáciu a výber môžete používať klávesnicu alebo myš.

Do príkazového riadku DOS-u napíšte `EDIT` a stlačte `ENTER`.

Ak ste MS-DOS editor predtým nepoužívali, môžete stlačiť `ENTER` a prečítať si informácie o tom, ako ho používať.

Môžete používať hocijaký textový editor, ktorého výstupom je jednoduchý ASCII súbor.

2.4.2 Assembler: DSPA

Asembler je vyvolaný príkazom:

```
DSPA <input file>  [<object file>[<listing file>]]  [-<options>]
```

Štandartná prípona pre `<input file>` je .asm, `<object file>` a `<listing file>` majú prípony .obj a .lst.

Niektoré z prístupných `<options>` sú uvedené v nasledujúcej tabuľke:

voľba	akcia
-v10	cieľový 32010
-v25	cieľový 320C25
-v50	**cieľový 320C5x**
-L	**výpis**
-x	súbor symbolov v tvare tabuľky spolu s uvedením miesta ich použitia
-c	veľkosť písma bezvýznamná
-w	varovania v súvislosti s "pipeline"
-q	kľudný chod

Štandartné privolanie assemblera môže byť:

```
DSPA  TEST  -V50  -L  -C
```

ktoré preloží `TEST.ASM` a vytvorí súbory `TEST.OBJ` a `TEST.LST`.
`-L` `-C` určujú, že bude vytvorený listing súbor, a že nebude záležať na veľkosti písma. Poznámka: bez voľby `C` symboly "Value", "VALUE value" neznamenajú to isté - detail, ktorý môže byť zdrojom problémov, ak sa nezoberie v úvahu.

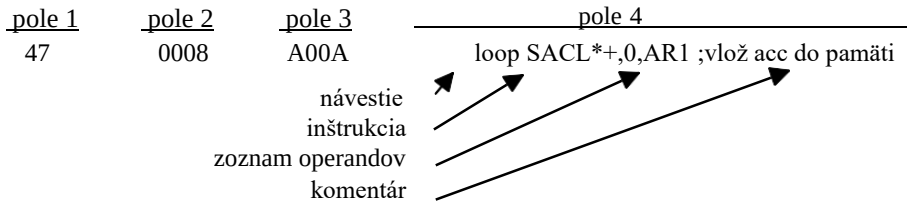
Assembler bude normálne vyhľadávať označené súbory len v aktuálnom adresári. *TMS320 Fixed-Point DSP Assembly Language Tools User´s Guide (SPRU018C)* popisuje, ako používať v príkazovom riadku voľbu `-i`, schopnosť použiť parametre A_DIR v prostredí DOS-u tak, aby sa počas prekladania prehľadali iné adresáre.

Ďalšie informácie o assembleri sú v *SPRU018C*, časti 3 a 4.

Keď je zvolená voľba `-L`, vytvára sa listing súbor, ktorý dokumentuje výsledný proces prekladania. Je to často užitočný nástroj počas ladenia, pretože chyby a výstrahy nasledujú za riadkom programu, s ktorým súvisia. Polia, ktoré sa zobrazujú v listing súbore sú popísané na nasledujúcom obrázku.

obrázok 2-11. Polia listing súboru

pole 1 obsahuje počítadlo riadkov zdrojového programu
pole 2 obsahuje počítadlo sekcie programu
pole 3 obsahuje objektový program (kód)
pole 4 obsahuje pôvodnú zdrojovú inštrukciu



2.4.3 Linker: DSPLNK

Linker pričleňuje software k hardwaru a je vyvolaný príkazom:

```
DSPLNK <options><meno súboru1..... meno súboruN>
```

kde:

`meno súboru1..... meno súboruN` je zoznam všetkých vstupných súborov linkera.

Niektoré z volieb sú popísané v nasledujúcej tabuľke (viac informácií nájdete v *Assembly Language Tools User´s Guide).*

voľba	akcia
-f<hodnota>	vyplní nepoužívané polia s <hodnota>
- m<meno súboru>	**map<meno súboru>**
-o	**output<meno súboru>**
-q	kľudný chod

Nasledujúce prípony súborov sú doporučené:

prípona	typ
.cmd	príkazový súbor linkera
.obj	preložený objektový súbor
.map	mapový súbor po linkovaní
.out	výstupný súbor po linkovaní

Štandartné privolanie linkera môže byť:

```
DSPLNK  TEST.CMD
```

Poznámka:
Je možné nepoužívať príkazový súbor pri špecifikácii všetkých vstupných a výstupných súborov pri vyvolaní linkera, ale tento súbor má niekoľko výhod. Umožňuje väčšie riadenie spôsobu linkovania, šetrí čas pri vývoji programu, a tiež zabezpečuje lepšiu dokumentáciu a opakovateľnosť. Pred linkovaním je samozrejme treba vytvoriť .cmd súbor, ale to je jednoduchá vec (viď vzory príkazových súborov obrázok 2-6. a obrázok2-7.).

2.5 Lab 2: COFF Tools Lab

Cieľom tohto cvičenia je precvičiť si zručnosti v používaní syntaxe COFF a vo vývoji programu. Vykonajte nasledujúce operácie:

1. Vytvorte program v JSA s menom `LAB2.ASM`, ktorý bude vykonávať ďalej uvedené.
   a. Vymedzí priestor pre 4-slovné dátové pole, 4-slovné pole koeficientov a 1-slovné pole pre výsledok.
   b. Vytvorí tabuľku pre hodnoty 1,2,3,4,8,6,4,2,0 s návestím na jej začiatku.
   c. Včlení segment programu so štyrmi inštrukciami NOP v slučke B a s globálnym návestím na jeho začiatku.
   d. Nastaví reset vektor pre segment programu.
2. Aby bol program použiteľný pre 'C5x, preložte ho. Požadovaný je aj listing súbor.
3. Príkazový program pre linker `C50.cmd` skopírujte do `LAB2.CMD`.
   a. Zmeňte `LAB2.CMD` tak, aby modeloval pamäťovú mapu 'C50-ky. Vložiť
      • všetky interné RAM do pamäti dát.
      • Definovať 8K pole externej pamäti programov na miesto 0x0000.
      • Definovať `LAB2` ako vstupný súbor.
      • Vytvoriť výstupný a mapový súbor.
   b. Sekcie vášho súboru v JSA vymedzte takto:
      • Alokácie pamäti RAM sú nasmerované do najmenšej internej RAM-ky.
      • Program je smerovaný do externej pamäti programov.
      • Tabul'ka je poslaná do veľkého interného bloku RAM.
   c. Ako by mal byť spracovávaný reset vektor?
4. Zlinkujte váš objektový súbor s novo modifikovaným príkazovým súborom linkera. Prezretím mapového súboru určite, či boli získané požadované výsledky.
5. Zbehnite váš program na simulátore. Napísaním `SIM5x LAB2.OUT` do príkazového riadku DOS-u sa začne ladenie.

V okne Disassembly window si môžete všimnúť, že reset vektor ukazuje skok na .text namiesto globálneho návestia prvej vykonávanej inštrukcie programu. Vaše návestie je stále platné, ale na obrazovke bolo nahradené návestím vytvoreným automaticky linkerom. Tieto návestia označujú začiatok a koniec sekcií linkera. Aby ste sa vyhli náhodnej zhode týchto návestí linkera s vašimi návestiami, je možné pred začatím písania regulárneho programu začať váš text inštrukciou NOP. Taký istý prístup by sa použil pri prvej adrese v .data. Pre vlastnú činnosť vášho programu sú inštrukcie NOP nepotrebné, ale robia ladenie prehľadnejším. Ak vám čas dovolí, môžete si zmeniť váš súbor tak, aby testoval túto procedúru. Pred opustením simulátora si prekrokujte váš program a všimnite si nekonečnú slučku.

6. Prehliadnite si skupinu súborov `A.BAT`, `L.BAT` a `ALS.BAT` Uvážte ich použitie ako spôsob na zvýšenie rýchlosti vývoja vášho programu.
7. Ak vám čas dovolí, experimentujte s ASM a CMD súbormi uvedenými skôr - vytvoriť väčšie segmenty, linkovať do rôznych pamätí, alebo predefinovať popisovač pamäti. Prezrite si výsledné mapové súbory, aby ste videli výsledky vašich experimentov. Použite rôzne mená súborov tak, aby ste neprepísali vašu pôvodnú prácu z bodu 4.

2.6 Opakovanie

text 2-12. Opakovanie

1. Z akých častí sa skladá riadok programu v JSA?
2. Čo je to direktíva?
3. Aké editory sa môžu používať?
4. Čo sú to sekcie COFF?
5. Aká je úloha príkazového súboru linkera?

Kapitola 3

3

Adresovacie režimy

Odsek 3-1.Učebné ciele :

Cieľom kapitoly je možnosť získať údaje vstupných operandov rôznymi spôsobmi.

Po skončení tejto kapitoly by ste mali vedieť :

-Používať veľké a malé konštanty.
-Nastaviť a používať priamy a nepriamy adresovací režim.
-Identifikovať najlepší postup na využitie v danom procese.
-Načítať ,ukladať ,registrovať ,čítať a odčítať hodnoty medzi pamäťou a akumulátorom.
-Účinne obsluhovať pamäťovo mapové registre.

3.1 Organizácia pamäte

Ako bolo uvedené v predchádzajúcom, mapa pamäte C5x je zložená z troch 64k rozsahov pre program , údaje a I/O pamäť. 64 k pamäťový rozsah vyžaduje 16-bitové adresy na konečnú identifikáciu každého miesta vo vnútri rozsahu. Obsahom každého miesta sa pracuje s 16-bitovým slovom.

Program pamäte je všeobecne adresovaný cez 16-bitové okamžité hodnoty, tak ako bolo vidieť v inštrukciách vetvenia /napr. B 1234h /. Adresovanie I/O pamäti bude prezentované v Module 7. Po prehľade pamäťových štruktúr prezentovaných na C5x, tento modul sa bude sústreďovať na rýchle údaje hodnôt, alebo operandy ako číselné konštanty, alebo ako komponenty mapy dátovej pamäti.

3.1.1 Čipové pamäťové obvody

Zariadenie C5x ponúka rôznorodé zmiešané pamäte, ktoré zahŕňajú programovú ROM, Dátovú RAM, a dvoj účelovú RAM.

Všetky C5x zariadenia obsahujú nejakú ROM, ktorá je aktivovaná keď MP/MC vývod má úroveň nízku. ROM je programovaný v TI a rôznych veľkostiach od 2k-16k bite závislých na zvolených zariadeniach. ROM je prezentovaná na najnižších adresách programového priestoru.

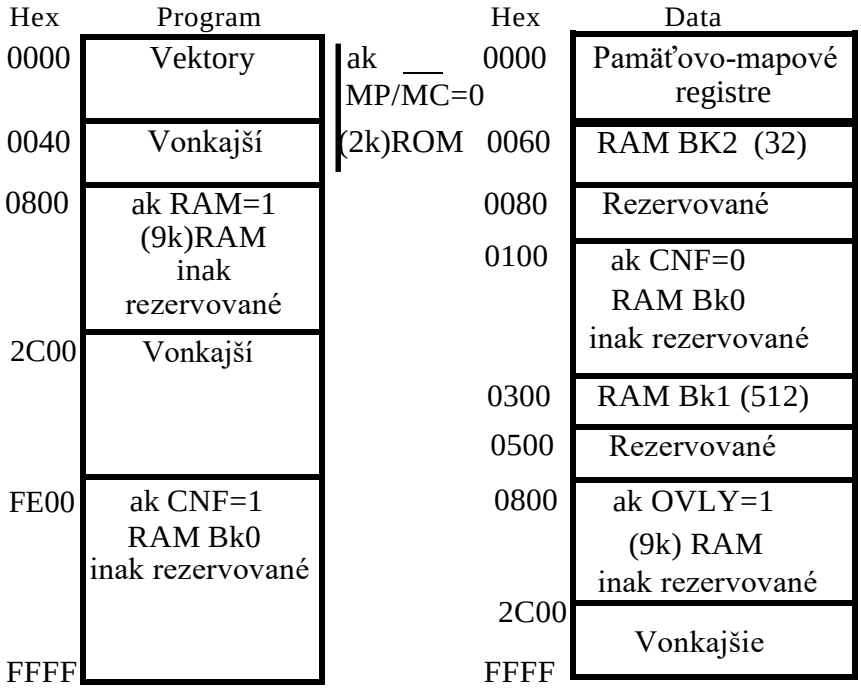
Data RAM sú zložemé z dvoch blokov pamäti umiestnených v 0060-007Fh a 0300-04FFh v údajových pamäťových mapách. Všetky C5x zariadenia majú tieto pamäte spoločné. Dodatočná vlastnosť týchto RAM-iek je to, že sú to dvoj -výberové pamäte, ktoré majú uplatnenie v oneskorovacích linkách (budú predvedené v module 7).

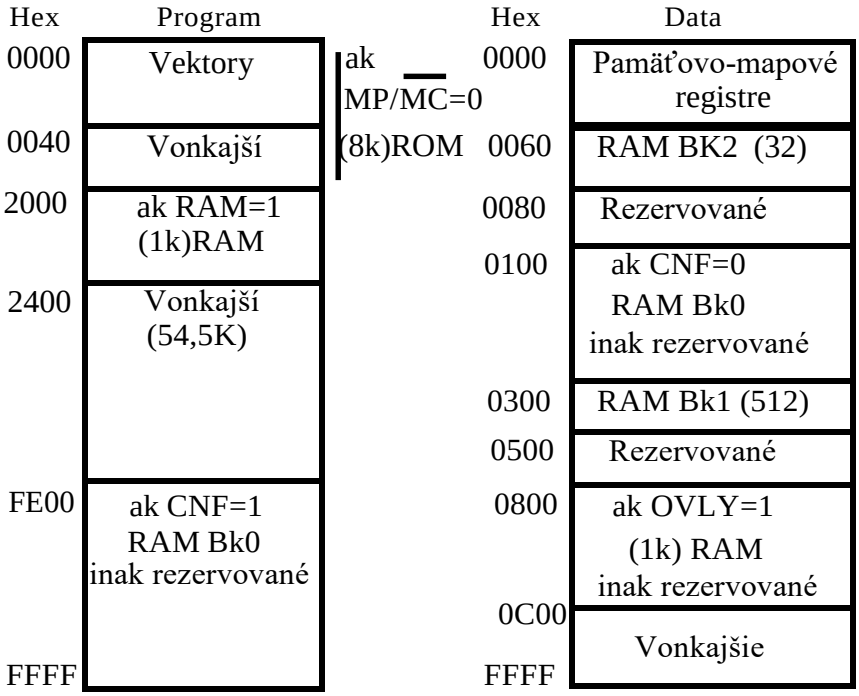
Dvoj účelovú RAM poznáme v dvoch typoch: malý blok dvoj-výberový RAM a veľký blok štandardného jedno-výberového RAM. Umiestnenie týchto RAM-iek sú nastavitelné užívatelom ako Dátové, alebo Programové pamäti, dovolujúcich väčšiu flexibilitu pri použití systémových prostriedkov. Dvoj-prístupový blok, ktorý sa nazýva aj "RAM Block O" je umiestnený v 0100-02FFh v Data pamäti pri nulovaní a možu byť premiestnené do FE00-FFFFh v Programovej pamäti cez CNF/configuračný/ bit pod riadením programu. Druhý RAM blok, je menený vo veľkostiach C5x vybratých zariadení, smie byť usporiadaný ako program, údaje, SHARED prostriedok, cez OVLY a RAM bity.

3.1.2 C5x pamäťové mapy

Nasledovné diagramy ukazujú umiestnenie a veľkosť vnútorných a vonkajších pamäťových komponentov využiteľných v rôznych C5x zariadeniach.

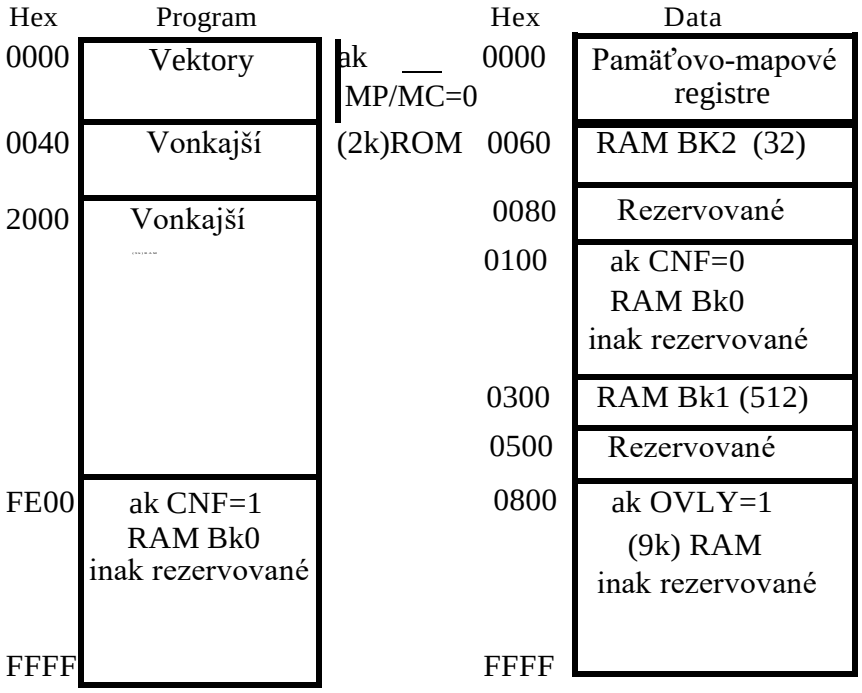
Obrázok 3-2. TMS320C52 Pamäťový plán



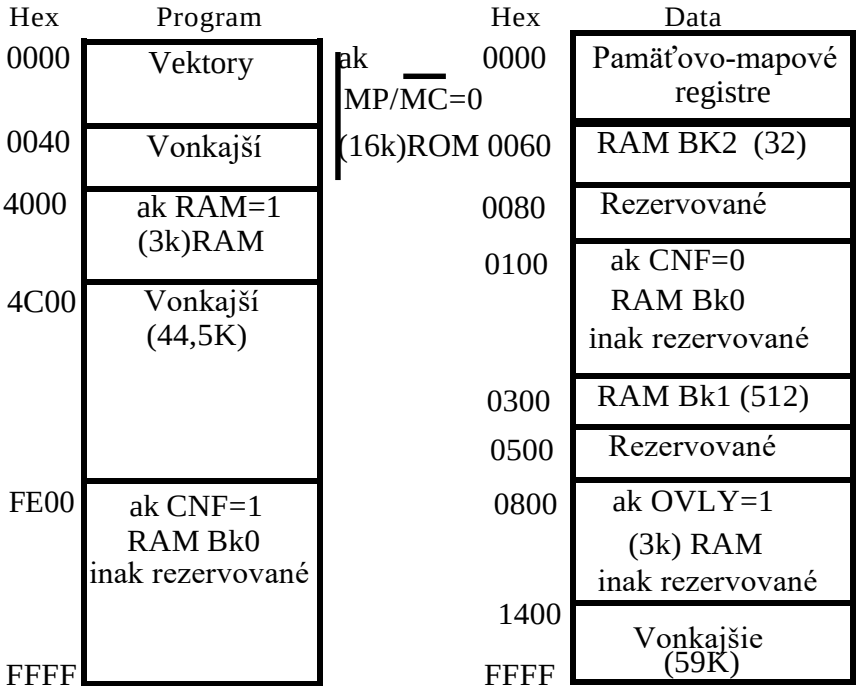


Obrazok 3-3. TMS320C51 Pamäťový plán

Obrazok 3-4 TMS320C52 Mapa pamäte



Obrazok 3-5 TMS320C53 Mapa pamäte



3.2. Režimy adresovania dát

C5x inštrukčný súbor režimov pre adresovanie dátovej pamäti a vyjadrovaní konštánt. V tomto module sa sústredíme na najzákladnejšie a najdôležitejšie z nich, vrátane nasledujúcich:

Odsek 3-6. Režimy adresovania dát

-Okamžitý/konštanta/	Inicializuje registre, obsluha s konštantami.
-Priame stránkovanie	Prístupné údaje na každej stránke v akomkoľvek poradí.
-Nepriame (smerníkové)	Prístupné údaje polí kdekoľvek v dátovej pamäti usporiadaným spôsobom.
-MMR (Špecifikovaný pamäťovo mapovými registrami)	Špeciálne funkčné registre srýchlym prístupom zabudované do pamäťovej mapy pamäťovo

Príklady použitia režimov adresovania nasledujúcimi inštrukciami, ktoré budú popísané viac v Module 4.

```
LACC x      , Load ACCumulator. Accumulator = "x"
SACL y      , Store ACcumulator Low half. Botton 16-bits of
            , acc. stored to memory location "y"
ADD a       , ADD to accumulator. ACC = ACC + "a"
SUB b       ,SUBtract from accumulator .ACC=ACC - "b"
```

3.2.1 Okamžité adresovanie

Pri okamžitom adresovaní, je operand časťou vlastných inštrukčných slov a je identifikovaný podľa / # /symbolu.

Malá hodnota smie byť vyjadrená do jednotlivého slova programu, zatiaľ čo veľká hodnota vyžaduje druhé slovno programu (a potom druhý cyklus k realizácii). Malé okamžité hodnoty sú obyčajne limitované 8-imi bitmi, ale v závislosti na inštrukciách v ktorej sa používajú ich rozsah je od 1-bit do 13 bitov. TMS320C5x užívatelský sprievodca na str.4-9 uverejňuje tabuľku šírok okamžitých inštrukčných slov.

Hodnoty, ktoré prekročia limit krátkych konštánt sa stávajú 2-slovné, 2-cyklové operácie na C5x, ale zhodné pohľadom z programátorskej perspektívy, ako vidieť na nasledujúcom programe:

```
ADD  # 12h        ; 0012 is added to the Acc, 1 cycle ADD  #
3456h             ; 3456 is added to the Acc, 2 cycles
```

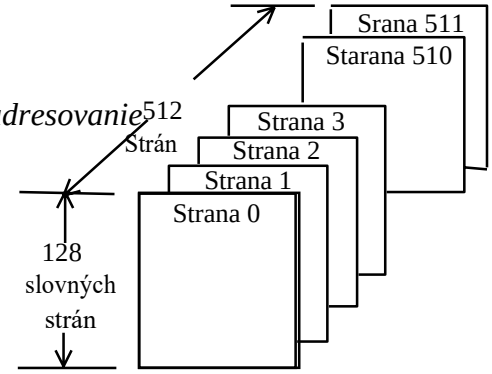
Dlhé okamžité inštrukcie dovoľujú špecifikovať druhý operand: posunutá hodnota ,ktorá môže byť použitá na umiestnenie 16-bit konštanty v 32-bitovom registri. Operácie posuvu budú popísané a využité v neskorších moduloch.

Zoznam všetkých C5x inštrukcií, ktoré dovoľujú okamžité adresovanie je daný v TMS 320 C5x-užívateľský sprievodca, str.4-9.

3.2.2. Priame adresovanie

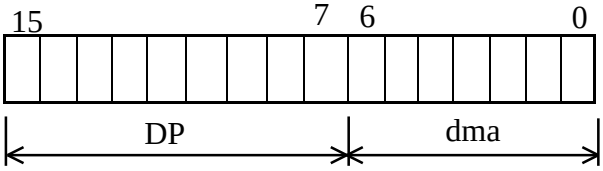
C5x má veľkosť inštrukčných slov 16 bitov a šírka data adries je 16 bitov , to by znamenalo ,že pokus o priame využitie hodnôt data pamäte si bude vyžadovať dva cykly :jeden na špecifikované operácie a druhý k vyjadreniu data pamäťových adries (trochu sa podobá metóde používanej v dlhých okamžitých adresovaniach) .To by však mohlo priniesť pomalý prístup (dvocyklový) priameho adresovania ,ktorý je nežiadúci v mnohých systémoch. C5x dovoľuje špecifikovať priestor pamäte a priamo adresovatelné inštrukcie pracujú s týmto zmenšeným rozsahom pamäti. Táto stránková pamäť je bežná pre vela procesorov ako kompromis medzi pamäťovým rozsahom a rýchlosťou.

V C5x je 9-bitový Data Page (DP) zoznam použiť na špecifikáciu aktívnej oblasti pamäti. Teda v každom čase, 128 umiestnení, na každej jednej z 512-tich vybraných strán, je aktívnych pre priame adresovanie. Týchto 128 miest smie byť adresované iba so 7-mimi bitmi, pracujúce v jednom -cykle. Keďže DP je programovateľný register, celá pamäťová mapa je prístupná na 128 miestach naraz.



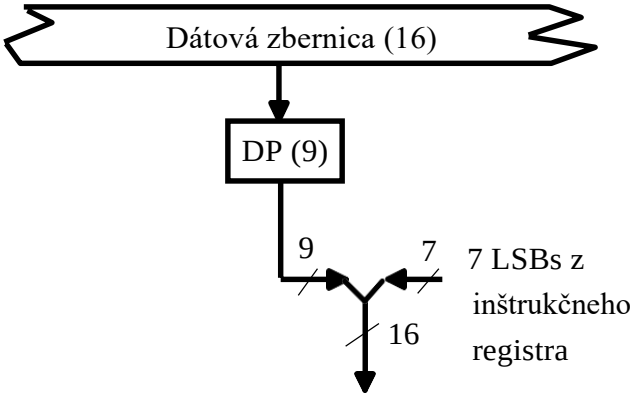
Obrázok 3-7. Priame adresovanie

Vytvorenie 16-bitovej adresy pomocou priameho adresovania :



3

Obrázok 3-8. Priame adresovanie - bloková schéma



15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Kód inštrukcie	**0**	**dma**

Bit 7 = 0 definuje priamy adresovací mód

Priame adresovanie, preto začína so spustením DP , najčastejšie pomocou LDP inštrukcie. Akýkoľvek počet operácií smie nasledovať za vybranou stranou.

```
LDP     # 1   ;make page 1 active (absolute locations 0080-00FFh)
LACC    3     ;load ACC from location 3 on p .1(abs.loc'n n. 0083h)
ADD     22h   ;add contents of loc'n 22, p.1 to ACC(abs. loc'n. 00A2h)
SUB     7     ;subtract contents of loc'n. 7 , p.1 from ACC (abs. loc'n. 0087h)
```


Prístup k hodnotám na rozdielnej strane vyžadujú iba jeden extra cyklus k opätovnému načítaniu DP.

```
LDP     # 2   ;make page 2 active (abs. loc'ns: 0100-017Fh)
SACL    9     ;store ACC to loc'n 9 of p.2(abs loc'n. 0109h)
```

3.2.2.1 Priame adresovacie úvahy.

Programátorovi sa doporučuje, zoskupiť čo najviac dát pre daný proces na jednej strane, teda minimalizuje čas strávený modifikovaním DP, a maximalizuje čas strávený v skutočnom spracovaní. Toto ukazuje ďalšie skutočnosti. Prvé, ako je popísané v module 2, COFF nástroje používané v TI sú najužitočnejšie keď sú symbolické,na rozdiel od pevných adries. Ako je potom nastavené DP, a ako programátor vie, kedy je okraj stránky prekročený?

Z toho vyplýva, že keď metóda pre prevádzanie priameho adresovania na symbolickej adrese je celkom jednoduchá, ako uvádza nasledovný príklad:

```
.bss    x,1     ;allocate 1 (16 bit) location for the variable"x"
LDP     #x      ;load the DP vith the page containing "x"
ADD     x       ;add to the acc. the contents of location "x"
```

V uvedených príkladoch je dôležité všimnúť si symbol # v LDP operandoch. Bez librovej značky by bola dátová stránka načítaná z nižších 9 bitov miesta x na aktuálnej strane a keďže strana ešte nebola inicializovaná úplne náhodná hodnota bude načítaná do DP .

Za odpoveď na druhú otázku (ako programátor vie kedy bol prekročený okraj stránky) považujte následovný príklad:

```
.bss    x,1     ;allocate a location for x
.bss    y,1     ;allocate a location for y
LDP     #x      ;be on the page that contains x
LACC    x       ;load ACC from loc'n x
ADD     y       ;add from loc'n y to ACC - but ,is the page correct?
```

Z tejto časti programu nie je známe že x a y sú na tej istej dátovej strane. Nasledovné prístupy existujú pre výstavbu spoľahlivých priamych adresovacích kódov :

1. Umiestňovať LDP pred každou priamou operáciou. Efektívny a bezpečný pri poruche ale premrhaný procesorový čas a programový priestor.
2. Uložiť .bss pridelenia (cez spojovací príkazový súbor) do pamäťovej štruktúry na jednotlivej data strane. Je to vynikajúce pre malé .bss systémy nedostatočné pre systémy ktoré prekračujú 128 .bss požiadaviek
3. Používanie pomenovaných sekcií /.usect/ na rozširovanie horeuvedeného spôsobu. Efektívny ale vyžaduje riadenie všetkých volaných sekcií v spojovacom príkazovom súbore ,ktoré sa možu stať obtiažné.
4. Používať FILE LIST voľbu v deklarácií sekcie vo vnútri spojovacého príkazového súboru. Štandardný súbor prideľuje .bss týmto spôsobom :

```
.bss   {}                    : >RAM PAGE 1
```

Zátvorky "{}"možu obsahovať zoznam súborov s ktorými sa má pracovať. Teda ak systém obsahuje súbory f1,f2,f3 a prvé dva sú na jednej strane ale posledný potrebuje byť nasmerovaný na novú stranu , príkazový súbor by mohol byť modifikovaný a vyzerať nasledovne :

```
.bss   {f1,f2}               : > RAM1 PAGE 1
.bss   {f3}                  : > RAM2 PAGE  1
```

V tomto prípade .asm súbory môžu používať jednoduché .bss pridelenia a spojovací príkazový súbor by zariadil ich správnu stránkovú kontinuitu.

5. Používanie .bss CPS /Contiguous Page Switch/,e.g., .bss x,5,1.Toto vnúti všetkým 5-tim miestam pola x byť na jednej strane. Spojovací program preskočí na ďalšiu dátovú stranu ak je nedostatočné miesto, poskytované na aktuálnej strane pre celé pridelenie. Preskočené miesta môžu byť vrátené späť .bss prideleniami ktoré sa môžu hodiť bez členenia. Keď používate túto metódu, vedzte ,že susedný vypínač patrí iba do jednotlivých oddelení takto:

```
.bss                 x, 5, 1
.bss                 y, 4, 1
```

môže byť na dvoch oddelených stranách. Na zabezpečenie umiestnenia x a y na tej istej strane považujte tento prístup:

```
      .bss           x, 9, 1
y     .set           x + 5
```

Nastavením x ako jedno veľké pole a deklarovaním y ako bodu v tomto poĺi, spojovací program zaobchádza s oboma ako s jednotlivými subjektami na jednotlivej strane. Vo väčšine prípadov je toto optimálna metóda pre ručné priame adresovanie.

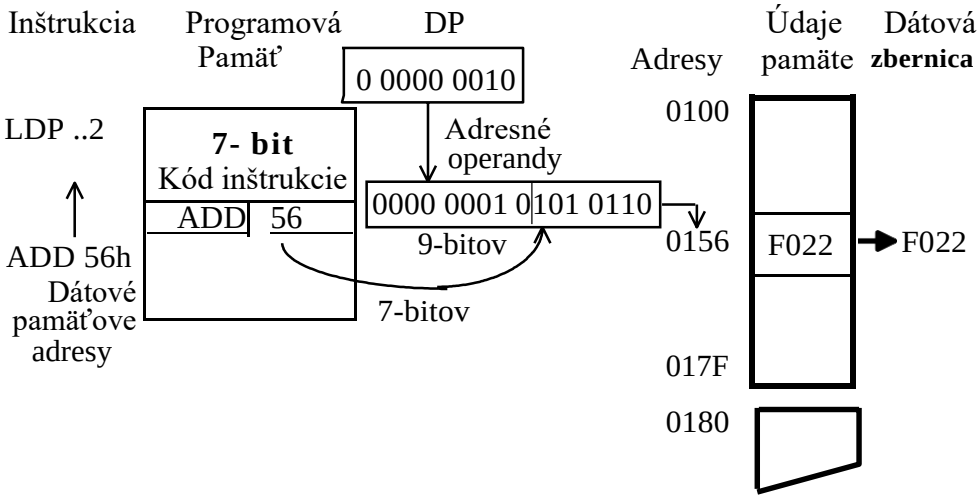
Metódy 3-5 si vyžadujú od programátora nastavenie dátovej strany keď zadáva

nový program. Potom všetky .bss pre individuálny program budú na jednotlivej strane.
Žiadny z týchto konceptov neukazuje najlepšie riešenie. Programátor si vyberie metódu, ktorá poskytuje žiadanú skúsenosť pre daný systém s najnižším množstvom námahy. Jednoduché systémy smú byť obslúžené skoršímy pokynmi, pokynmi, zatiaľ čo náročnejšie systémy vyvolávajú spôsoby z neskorších variant, alebo kombinácie podobných.

3.2.2.2 Prehľad priameho adresovania

Na nasledovnom obrázku /3-9/ je prezentovaný príklad priameho adresovania na C5x.

Obrázok 3-9.Priamy adresovací prehľad



3.2.3 Nepriame adresovanie

Nepriame adresovanie je účinná a výkonná cesta prístupu k údajom uložených na zoznamoch, poliach, alebo iných usporiadaných skupinách v pamäti. Na rozdiel od priameho adresovania adresa nie je vyjadrená v inštrukčnom slove, ale namiesto toho je umiestnená v Pomocnom adresári AR (Auxiliary Register). Používanie registra AR zabezpečí úžitok. V prvom rade AR-ky sú 16-bitové a teda môžu ukazovať na ktorékoľvek miesto v celom data pamäťovej mape bez pomoci data stránkového registra. Dodatočné AR smú byť automaticky zväčšené a zmenšené potom, ako je

operand čítaný, takže na nový údaj sa ukazuje pre použitie v neskoršej operácií. Využitie tejto vlastnosti robí výkon interačných postupov rýchlejší a lahší.
Osem AR -zoznamov je použitelných v C5x zariadeniach. Pre použitie v nepriamom adresovaní, musí byť AR najskor vybratý a spustený. AR-ky sú spustené cez LAR (Load Aux Register) pokyny. Počiatočný výber aktývných AR je cez MAR/ Modify Aux Register/ pokyny. Pre väčší výkon môže byť nasledujúci AR výber špecifický s každým pokynom používaným v priamom adresovaní.

3

3.2.3.1 Príklad nepriameho adresovania

Nasledujúci ukážkový príklad použitia nepriameho adresovania nám pomože objasniť tento proces. Vymedzíme dve polia v pamati, x a y, každé obsahuje 4 hodnoty. Jeden AR môže byť iniciovaný ako smerník na x a druhý na y .Na sčítanie všetkých hodnôt spolu, v prvom rade by bolo potrebné aktivovať jeden AR pre sumáciu x hodnôt, a potom mať aktívny druhý AR pre sumáciu y hodnôt. Nasledujúci kód zavádza tento proces.

Obrázok 3-10. Príklad nepriameho adresovania 1

```
.bss    x,4             ;allocate 4 locations for the "x" array
.bss    y,4             ;allocate 4 locations for the "x" aray
LAR    AR2,/.../x       ;load AR2 with the first "x" address
LAR    AR5,/.../y       ;load AR5 with the first "y" address
MAR    *,AR5            ;make AR2 active
LACC   *+               ;load acc. using AR2, then increment AR2
ADD    *+               ;add to acc. using AR2, then increment AR2
ADD    *+               ;add to acc. using AR2, then increment AR2
ADD    *,0,AR5          ;add to acc. using AR2, then make AR5 active
ADD    *+               ; add to acc. value pointed to by AR5 (y) , inc AR5
ADD    *+               ;add and increment
ADD    *+               ;add and increment
ADD    *+               ;add and increment
```

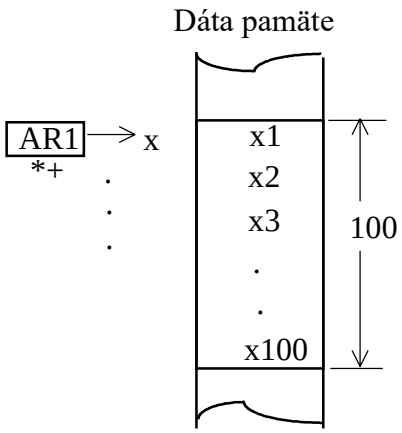
Hvezdička # je nepriamy operátor, označujúci použitie aktuálneho AR ukazujúci na hodnotu, ktorá bude použitá. Ak je tam pridané /+/ to značí funkciu samopričítania , kde aktuálnemu AR bude pričítaná jednotka po načítaní operandu. Podobne /-/znamená automatické odčítanie. Jednoúčelové zariadenia C5x poskytujú hardver pre zavádzanie autopričítacích (odčítacích) operacií, pre ktoré sa nepožaduje zvláštný cyklus.

Iný príklad použitia neprameho adresovania, kde je smerník použitý na predvedenie súčtovania hodnôt v poli.

Obrazok 3-11. Príklad nepriameho adresovania 2

$$y = \sum_{n=1}^{100} X_n$$

```
.text
.bass   x,100
LAR AR1,/.../x
LARP AR1
RPTZ /.../99
ADD  *+
SACL y
```



Note: LARP ARn = MAR*,ARn

3.2.3.2 Operandy nepriameho adresovania

Všimnite si, že keď AR5 je aktivovaný, 0 bol špecifikovaný ako druhý operand. To je preto, lebo inštrukcie , ktoré obsluhujú akumulátor ponúkajú posúvne varianty v druhom operandovom poli. Keďže žiadny posuv nebol potrebný v tomto prípade, a 0 (implicidná hodnota) bola vložená do tretieho operandového poľa - NARP / nový ARP /. Z tohoto je zrejmé, že jednotlivá inštrukcia je schopná hľadania, posúvania a používania operandu, potom práce aktuálnym AR a potom vyberania nového ARP.

Obrázok 3 - 12. Nepriame adresovanie - logické sekvencie

Príklad: ADD *+, 1,ARO

1. * ARP vyberá AR k adres. - operandu
2. 1 posúva operand v ľavo
3. ADD Obsluha v posunutom operande
4. + Modifikacie AR hodnody
5. ARO nová ARP hodnota

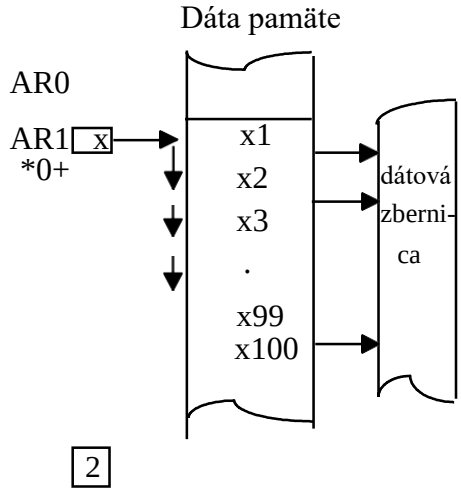
3.2.3.3 Indexovanie s nepriamym adresovaním

Niekedy je potrebné pričítať/odčítať iné hodnoty než jednu . V tomto prípade je indexový register požadovaný na špecifikáciu veľkosti kroku, ktorý bude robený. V C5x , ARO može byť použitý buď ako jeden smerník alebo ako hodnota indexu na iný smerník. Použitie indexu počas automatickeho pričítania/odčítania je špecifikované pridaním 0 /skratka pre ARO / pred modifikátor / *0+ alebo *0-.

Obrázok 3-13 Príklad indexovaného adresovania

$$y = \sum_{n=0}^{49} X_{2n}$$

```
.text
.bss    x,100
LAR    AR1,..x
LAR    AR0,..2
LARP AR1
RPTZ ..49
ADD *0+
SACL  y
```



Niektoré programy môžu potrebovať prístup ku všetkým ôsmim smerníkom /ARs / a indexovej hodnote. V tomto prípade je možné efektívne rozdeliť ARO do troch oddelených registrov :

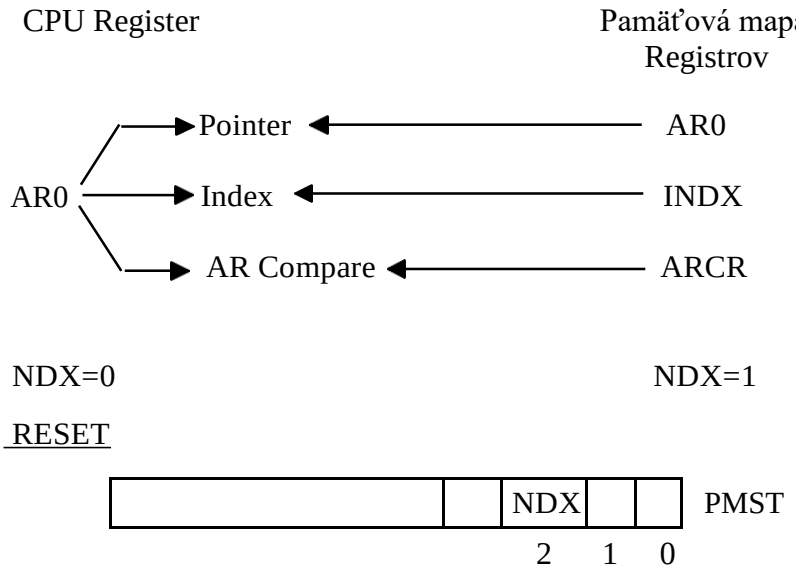
-SMERNÍKOVÝ REGISTER /A POINTER/ nazývaný ARO

-INDEXOVÝ REGISTER /An INDEX / nazývaný INDX

-/An AUX. REG. COMPARE /register nazývaný ARCR

To je robené nastavením NDX bitu v PMST registri.. Po resete ,NDX=0, a ARO je jeden register s tromi funkciami a INDX a ARCR obraz ARO . Ak NDX=1 , tri registre sa stanú nezávislé , dovoľujúce všetkým trom funkciám byť aktívne v rovnakom čase . V ďalšej časti program potrebný na predvedenie tejto činnosti bude ukázaný .

Obrazok 3-14. ARO



Ak je daný výber z dvoch možných módov činnosti pre ARO uvažujeme nasledovné možnosti.

So spojeným ARO to je:

-Prospech ,ktorým LAR inštrukcie môzu byť použité k iniciovaniu každej z týchto operácií

-Zodpovednosť z možných potrieb deliť jeden AR na 2 , alebo 3 funkcie .

S oddeleným ARO ,INDX a ARCR registrom to je:

-Prospech z udržiavania všetkých troch funkcií naraz .

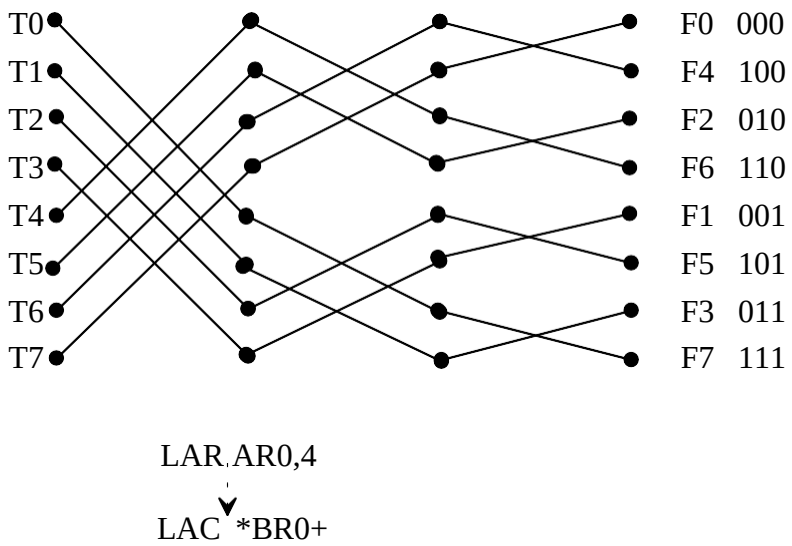
-Zodpovednosť z iniciovania zoznamov oddelene . INDX a ARCR registre nie sú CPU registre, teda musia byť prístupné cez pamäťové mapované miesta .

Ktorýkoľvek mód z NDX bitu je použitý , indexné adresovanie je vždy špecifikované *0+ alebo *0- , nie *I+ alebo iným označením .

3.2.3.4 Bitové reverzné adresovanie

Rýchla fourierova transformácia pracuje na poliach údajov smerujúcich k premene medzi časovými a frekvenčnými oblastnými vzorkami . Zoskupenia sú zahrnuté pomocné registre sú prirodzene vybraté pre pracovanie na údajoch .

Obrázok 3-15 Bit-prevrátené adresovanie



Aktuálny kód k zavádzaniu FFT možno nájsť na TI bulletine, a nebude tuto popisovaný . Jeden ďalší detail potrebuje byť zavedený pre bitove reverzné adresovanie k správnej funkcií ,t. j. FFT pole musí byť zavedené v "0"štartovacej adrese , alebo adresa s dostatočným počtom LSB rovných 0 . V tomto príklade (8-bitový FFT), tri LS bity sú použité v bitovo reverznom adresovaní a 13 zvyšných MSB zostáva nezmenených. Teda akákoľvek adresa dátovej pamäti s 3-omi LS bitmi v nule by bola prípustná pre štartovací bod. Potreba pre špecifický typ , alebo RAM pre FFT volá po použití *.usect* v súbore *.asm*. Pri špecifikovaní počtu LS nulových bitov si treba dávať pozor na spojovací príkazový súbor s *align* direktívou , ako to ukazuje nasledovný obrázok.

Obrázok 3-16. ASM a LINKER CMD Files

ASM FILE (excerpt) :

```
BASE  .usect        "fft array" , 8 ;allocate 8 loc'ns starting at BASE
      .mmregs                       ;allow use of MMRegister names
      .text
      LACC          #0100b          ;bit-rev increment value
      SAMM          INDX            ;store to INDX register
      LAR           AR1,,#BASE ;AR1 points to top of fft array
      MAR           *,AR1           ;AR1 is the active pointer

      RPT           #7              ;8 interations
      IN            *BRO+,PA2 ;read a data sample into array (bit rev'd)
```

LINKER CMD FILE (excerpt)

```
SECTIONS  {
 vectors :                  {} > VECS,    PAGE0
 .text :                    {} > ROM,     PAGE0
 fft array  align (8) :     {}  > RAM,     PAGE1
 .bbs:                      {}  >RAM,      PAGE1
                    }
```

3.2.3.5 Prehľad nepriameho adresovania

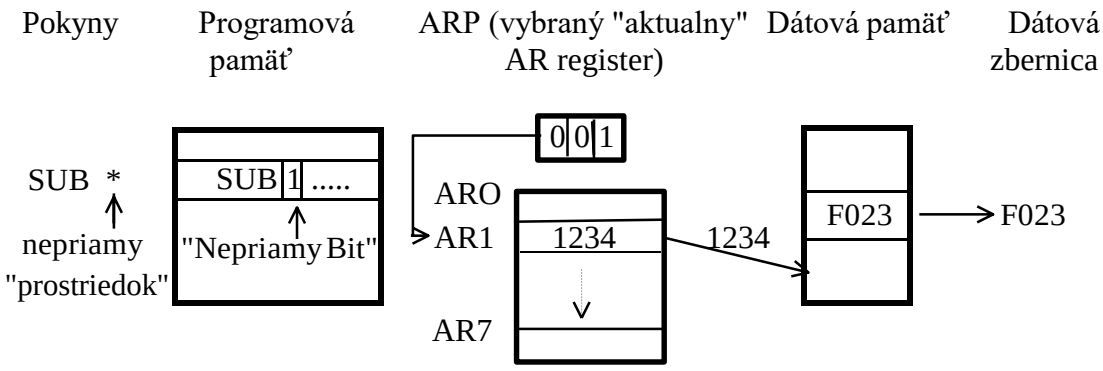
Nepriame adresovanie dovoľuje 16-bitové registre používať ako smerníky k dátovej pamäte. Často sa to používa na prácu v dátových poliach a vrátane vstavaného hardveru k zavádzaniu niekoľkých foriem samopričítavacích a samoodčítavacích funkcií v rámci operácie ako ukazuje nasledovný obrázok .

Obrázok 3-17 .TMS 320C5x nepriame adresovanie

OP * [, nalbližší ARP]	Žiadna zmena v aktuálnom AR
OP *+ [, najbližší ARP]	Aktuálny AR je pričítaný
OP * -[, najbližší ARP]	Aktuálny AR je odčítaný
OP *0+[, najbližší ARP]	INDX je pridaný k aktuálnemu AR
OP *0- [, najbližší AR/]	INDX je odčítaný z aktuálneho AR
OP *BRO+ [, najbližší ARPú]	INDX je pridaný k aktuálnemu AR s opačným vykonávaním šírenia/
OP *BRO-[, najbližší ARP]	INDX je odčítaný z aktuálneho AR /s opačným vykonávaním šírenia/

Nakoniec, ak zvažujeme príklad o nepriamom adresovaní na nasledovnom obrázku, ako je písané, objavuje sa v C5x a je zavádzaný C5x - om.

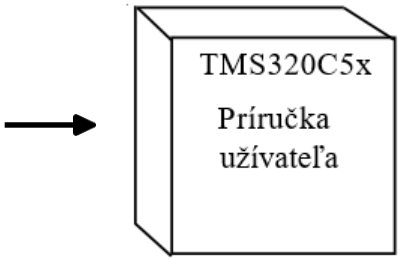
Obrázok 3-18. TMS320C5x príklad nepriameho adresovania



3.2.4 MMR ADRESOVANIE

C5 x má veľký počet pamäťovo - mapových registrov (MMR s), ktoré majú schopnosť prevádzať rôzne funkcie, ktoré budú preskúmané v priebehu tohoto kurzu. Tieto registre môžu byť prístupné ako dátová pamäť v pamäťovom mieste OOOOH - OO6OH. . Nasledovný ukážkový materiál uvádza celý zoznam MMRs, ich mená, pamät. miesta a popis ich funkcií.

Mapa adries dátovej stránky 0
tabuľka 6-8 ,strany 6-14-6-15



Pretože MMR sú súčasťou dátovej pamäťovej mapy, smú byť písané a čítané rovnakým spôsobom, ako ktorékoľvek iné dátové pamäťové miesto cez priame a nepriame adresovanie.

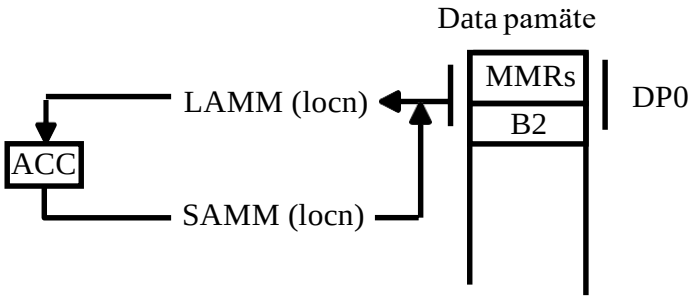
Ešte jedna metóda, nazývaná pamäťovo - mapové adresovanie existuje pre vstup do MMR. Pretože všetky MM Rregistry sú umiestnené na strane O, MMR adresovacie inštrukcie pracujú iba na strane O, preto ignorujú obsah z DP, ak bolo použité priame adresovanie a používajú iba LS 7 bitov z aktuálneho AR ak bolo použité nepriame adresovanie.

Dve MMR operácie sú LAMM / načítaj akumulátor z pamäťovo mapového registra a SAMM ulož akumulátor do pamäťovo mapového registra. Tieto inštrukcie sú užitočné v mnohých programovaniach C5x. Napr. ako uvažujeme prípad v predošlej časti, kde bolo žiadané umiestniť bit do PMST, prístupné ako MMR. Program pre nastavenie NDX bitu je jednoduchý :

```
.mmregs          ; allovws mmr names to be used as addres labes.
lamm PMST        ; load acc from proc.mode stat . reg.
or # 0100B ; turn on the NDX bit separating AR0, INDX, and
                 ; ARCR regs.
sam PMST         ; store the modified value bact to the proc. mode
                 ; stat.reg.
```

Tieto inštrukcie boli vyvinuté hlavne pre prístup do MMR registrov, ale môžu sa tiež vysvetľovať ako metóda pre prístup do pamäti v RAM Block 2, ktorá je umiestnená na data strane 0.Teda RAM block 2, hoci iba 32 umiestnení dlhá, je niekedy cenná časť systému C5x, pretože môže byť prístupná cez MMR adresovacie techniky.

Obrazok 3-19. Adresovanie panäťovo mapovaných regisrov

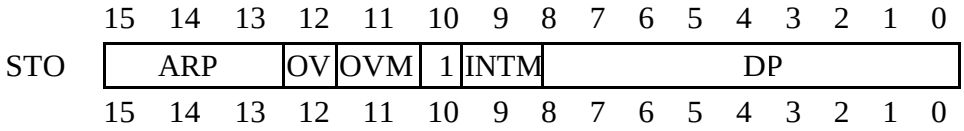


Príklady:

```
LAMM   07    ;load ACC from PMST
OR #0100B ;set bit in NDX position
SAMM 07      ;store back to PMST
LACC #0      ;clear ACC
```

```
SAMM 60H ;store 0 to lst location in B2

.mmregs        ; equates all MMR names with locations
LAMM PMST
OR #0100B
SAMM PMST
LACC #0
SAMM 60H
```

Existujú dve iné inštrukcie, ktoré pracujú špeciálne na strane 0. Tieto budú zvažované v module na prerušeniach.



3.3. Stavové registre a "Watch" operácie

Pri ladení programu na C5 x je často požadované pozorovať hodnoty DP a ARP pre overenie si správnej funkcie adresných operácií. Pretože DP a ARP sú komponenty stavového registra STO, nie je možné vidieť ich priamo ak sa so stavovým registrom nepracuje v rámci tzv "Watch" operácie, ako ukazuje nasledovný obrázok :

Obrazok 3-20 Organizácia stavového registra

Simulator command to ectract Dp and ARP:

```
wa        ST0 >> 13, ARP

wa        ST0  << 7 , Base, x
                |          |     |
          item to watch    label  format
```

(1151319G)

Kedykoľvek je ladený program v asembleri doporučuje sa, aby nejaká verzia z týchto "Watch" operácií, buda špecifikovaná tak, že adresovacie registre zoznamy môžu byť ľahko obslúžené. Okrem toho "Watch" príkazy uvedenévyššie mohli byť písané do ASCII súboru a vyvolané ako DOS súbor, použtím príkazu simulátora TAKE a mena súboru.

"Watch" operácie sú často používaným komponentom ladenia. "Watch" príkaz v debugeri sa skladajá z 3-och polí. Prvé pole, ako je vidno hore, definuje

položku ku sledovaniu. To smie byť jednoduchý výraz alebo zložený na postupnosti "Booleových", alebo aritmetických operácií. Ak je požadovaný k sledovaniu obsah umiestnenia v pamäti, použi konvenciu C jazyka a pred adresu predraďte operátor smerníka ra (*). Druhé pole je voliteľné - dovoľuje užívateľovi zadať text, ktorý sa objaví na ľavo sledovanej položky. Bez tohto meno je implicitne text prvého poľa. Tretie pole dovoľuje užívateľovi zostaviť alebo vybrať formát čísla zobrzenia. Najoblúbenejšie voľby sú X pre šestnástkove a *d* pre desiatkové. Tieto voľby dovoľujú užívateľovi vytvoriť popis a vhodné zobrazenie tzv. "Watch".

Ďaľšia užitočná vlastnosť debugera je schopnosť mať všetko čo napíšete do príkazového riadku kopírované do DOSu. Tento proces dovoľuje užívateľovi zaznamenať klávesové údery, ktoré možu byť opakované v neskoršej relácii podobne ako dávkový "Watch" súbor v DOSe. Táto funkcia je zlepšenie skupiny alebo súborov, hoci v tom užívateľ intenzívne pracuje so simulátorom na vytvorení dávkového "Watch" procesu namiesto písania textu v súbore a po tomto neskôr testuje, aby videl , či to pracuje správne. "Dialógová" sekcia je spustená v DLOG príkazom, kde súbor je meno záznamového súboru, ktorý bude vytvorený. Užívateľ potom zadá činnosti, ktoré budú zaznamenané a úkon či sekciu s DLOG CLOSE príkazom. Užívateľ je uistený o schopnosti opakovať činnosti sekcie s TAKE < file > príkazom.

Program	Accum	DP	ARP	AR0	AR1	AR2
LDP #0 MAR *,AR1 LAR ARO,#2						
LAR AR1,#200h LAR AR2,#300h LACC 61h						

ADD *+
SUB 60h,1
ADD *+,AR2						

LDP #6 ADD 1 ADD *+,4						

SUB *+,0,AR1
SUB #32
ADD *0-,0,AR2						

SUB *0- SACL 62h						

3.4. Precvičenie adresovania

Na začiatku budú detaily adresovania skúšané so segmentom programu, uvedeným nižšie. Máte prečítať program na ľavej strane, používať pamäť tak, ako je uvedené vyššie a vykonať program, tak ako by ho podľa vás vykonal procesor C5x. Ako je to často úloha vo vývoji programu, dávaj pozor na chyby.

Obrazok 3-21. Precvičenie adresovania

Adresy/Data (hex)	Blok B2		Blok B0		Blok B1	
DP=0	60	10	DP=4 200	100	DP=6 300	10
	61	120	201	60	301	30
	62		202	40	302	60

3.5. Cvičenie 3 : adresovanie

Cieľom tohto cvičenia je overenie mechanizmu adresovania. V tomto procese zväčšíme ASM súbor z predchádzajúceho cvičenia k zahrňovaniu nových funkcií. Naviac budeme načítavať program do simulátora učiť sa ako spustiť a pozorovať operáciu programu kódu v HLL debugeri.

V tomto cvičení budeme inicializovať ".bss" polia pridelené v predchádzajúcom cvičení s obsahom ".data" tabuľky. Ako je to najlepšie dosiahnuteľné? Zvažujeme postup načítania prvej ".data" hodnoty do akumulátora a potom ukladanie tejto hodnoty do prvého ".bss" pamäťového miesta a opakovanie tohto procesu pre každé nasledujúce hodnoty.

Ktoré formy adresovania možu byť použité na tento účel ?

Ktorý adresovací režim by bol najlepší v tomto prípade ? Prečo ?

Aké problémy možu vzniknúť s použitím iného režimu ?

Berúc v úvahu tieto otázky, vykonávame nasledujúci postup :

3.5.1 Postup :

1. Kopírovať LAB2. ASM do LAB3. ASM, zmeniť LAB3. ASM na :
A. Iniciovať pridelené RAM pole z ROM inicializačnej tabuľky :
A1 - Otvoriť súbor LAB3. ASM.
A2 - Zmazať NOP a B operácie z "text" časti.
A3 - Iniciovať smerníky na začiatok "data" a "bss" polí.
A4 - Premiestniť prvú hodnotu z "data" do "bss" poľa.
A5 - Opakovať postup pre všetky hodnoty, ktoré majú byť inicializované.
B. Čítaj, zmeň a ulož PMST (MMR ktoré riadi ARO štýl) :
B1 - Prideľ dve umiestnenia v RAM, nazvané x a y.
B2 - Uložiť PMST do pamätového miesta x.
B3 - Zmeniť PMST na rozdelené ARO funkcie.
B4 - Skopírovať novú PMST hodnotu do pamäťového miesta y.
B5 - Skončiť postup so skokom späť na začiatok programu.

2. Preložiť program asemblerom pre použitie na C5x. Požiadanie a skontrolovanie výpisu programu.

3 . Skopírovať LAB2. CMD do LAB3. CMD. Spojiť Váš asemblerom preložený tzv "object" súbor s náležite zmenenou verziou LAB3. CMD. Skontrolovať mapový súbor a rozhodnúť, či bol získaný očakávaný výsledok.

4. Odladiť program použitím simulátora. Pozorovať registre vyskytujúce sa v tomto cvičení pomocou " Watch" operácie popísanej v časti 3.3. Tiež uvažujte vytvorenie LOG súboru z týchto pozorovaní .

3.6 Prehľad adresovania

Obrázok 3 - 22. "C5x inštrukcie :

LACC	B
ADD	SUB
SACL	LAR
LAMM	SAMM
MAR	LDP

Obrázok 3 - 23. Prehľad :

1. Aký je rozdiel medzi dátovou pämaťou a programovou pamäťou ?
2. Čo je priame adresovanie ? Uveď príklad.
3. Čo je okamžité adresovanie ? Uveď príklad.
4. Čo je nepriame adresovanie ? Uveď dva príklady.
5. Koľko pomocných regitstrov existuje ?
6. Aký je rozdiel, keď používam LAMM a SAMM ?

Kapitola 4

Základné programovacie techniky

Odsek 4-1. Učebné ciele :

Predmetom tohto modulu je naučiť sa písať kód C5x na vykonávanie základných aritmetických rovníc.

Základné predmety zahŕňajú schopnosť :

- Vykonávať základné vetvy, spätnú kontrolu a rutinné operácie.
- Identifikovať komponenty CALU a ich vnútorné spojenie.
- Použiť akumulátor na nabitie, skladovanie, sčítanie a odčítanie 16-bit hodnôt z datovej a programovej pamäte.
- Pracovať s 32-bit dátovými hodnotami.
- Použiť akumulátorový buffer pre dočasné uskladnenie alebo min/max operácií.
- Použiť deličku na zostrojenie čiastkových súčtových rovníc.

4.1 Programová kontrola

C5x poskytuje niekoľko metód kontroly priebehu vykonávania programu. Normálny priebeh programu je sekvenčný; to znamená, že procesor načíta a vykoná ďalšiu inštrukciu v programovej pamäti. Keď je potrebné prerušiť sekvenčné vykonávanie, môžeš použiť BRANCH, CALL alebo TRAP a softverové prerušenie. BRANCH-y prenášajú kontrolu na hociaké miesto v programovej pamäti. CALL prenášajú tiež kontrolu na ľubovoľné programovo-pamäťové miesto, ale umožnia ľahký návrat do pôvodného programu s použitím RETURN(RET) inštrukcie. TRAP a softvérové prerušenia sú špecifické formy volaní.

BRANCH a CALL môžu byť nepodmienkové, t.j. vždy berieme keď sme na ne narazili; alebo podmienkové, kde BRANCH alebo CALL je vykonaná v závislosti od stavu procesora.

Matica inštrukcií štandartnej programovej kontroly a podmienkových kódov sú dole. V ďalšom bude všetko detailne vysvetlené.

Odsek 4-2. Programová kontrola

BRANCH	CALL	RET
B ďalší BACC BCND ďalší, podm, podm, ...	CALL podprog CALA CC podprog, podm, ...	RET - RETC podm, podm, ...

Vysvetlivky :
podm - podmienkové
podprog - podprogram
nepod - nepodmienkové

Podmienkové kódy

EQ LT LEQ	ACC=0 ACC<0 ACC≤0	NEQ GT GEQ	ACC<>0 ACC>0 ACC≥0
C OV TC	C=1 OV=1 TC=1	NC NOV NTC	C=0 OV=0 TC=0
BIO	BIO=0	UNC	nepod

4.1.1 BRANCH-y

BRANCH inštrukcie prenášajú kontrolu na ľubovoľné miesto v programovej pamäti. Väčšina BRANCH inštrukcií sú dvojslovové a vykonávajú sa v štyroch cykloch.

4.1.1.1 Nepodmienková BRANCH

Nepodmienkové BRANCH sú vždy brané. Ako aj upravenie programového počítadla, nepodmienkové BRANCH tiež poskytujú programátorovi príležitosť použiť ARAU na úpravu aktuálneho pomocného registra (AR) a smerníka pomocného registra (ARP). V programe napísanom v asemblery, nepodmienkové BRANCH inštrukcie sú písané následovne :

[label] B pma [, { ind } [, ďalší ARP]]

kde :

0000h ≤ pma ≤ 0FFFFh
0 ≤ ďalší ARP ≤ 7

Prvé slovo inštrukcie je BRANCH operačný kód. Druhé slovo je adresa programovej pamäte (< pma >). Keď je normálna BRANCH inštrukcia dekódovaná, ľubovoľná ARAU operácia špecifikovaná nepriamym adresovaním je vykonaná. Keď je inštrukcia vykonaná, PC je nahratý s < pma > a vykonávanie programu začína na novej adrese špecifikovanej v < pma >.

4.1.1.2 Podmienková BRANCH

BRANCH-y môžu byť tiež podmienené na stave procesora. Podmienkové BRANCH-y sú kódované následovne :

[label] BCND pma , [podmienka 1] [podmienka 2] [, ...]

Keď všetky podmienky sú splnené, PC je nahratý s < pma >. Keď ľubovoľná podmienka nie je splnená, PC prečíta adresu nasledujúcu po BRANCH adrese a vykonávanie bude pokračovať.

Podmienkové BRANCH-y, keď sú vzaté, vyžadujú štyri cykly na vykonanie. Keď podmienkový BRANCH nie je vzatý, vykonanie nastane v dvoch cykloch.

Testovanie $\overline{\text{BIO}}$ a TC je vzájomne vylúčené. Všetky iné kombinácie podmienok môžu byť špecifikované, hoci nie všetky kombinácie majú význam.

4.1.1.3 Dynamická BRANCH

BACC je jednoslovová inštrukcia, ktorú číta PC z nižších 16 bitovakumulátora. BACC inštrukcia poskytuje pre BRANCH priebehovo programovateľnú adresu. Táto inštrukcia je použiteľná, keď program môže vziať jeden z mnohých BRANCH-ov založených na nejakej podmienke.

Zlomkový kód programu v nasledujúcom diagrame číta hodnotu medzi 0 a 16 z I/O portu a zoberie BRANCH založený na načítanej hodnote.

Odsek 4-3. Príklad dynamického BRANCH-u

```
begin :  LAMM     PA0
         ADD      #table
         BACC

         .data            ; ukladá .data do dátovej mapy
table:   .word            rutina 1
         .word            rutina 2
           •
           •
           •
         .word            rutina 14
         .word            rutina 15
```

4.1.1.4 Dekrementácia a BRANCH

BANZ je špeciálny prípad podmienkovej BRANCH inštrukcie. Môže byť použitá na implementáciu slučiek, ktoré sa vykonajú určený počet krát. BANZ inštrukcia je kódovaná následovne :

[label] BANZ pma [, { ind } [, ďalší ARP]]

BANZ inštrukcia je podmienená na obsahu aktuálneho AR predtým, ako bola inštrukcia BANZ dekódovaná. Keď obsah aktuálneho AR nie je nula, BRANCH je vzatý. Keď aktuálny AR=0, kontrola prechádza na ďalšiu inštrukciu v programovej pamäti. Aktuálny AR a ARP sú modifikovateľné ako poznámkované. Keď nebola špecifikovaná žiadna modifikácia, modifikácia AR sanastaví na základnú, t.j. dekrementácia o 1. Keď BANZ inštrukcia je použitá na konci slučky, N+1 iterácia slučky bude uskutočnená, keď pomocný register je inicializovaný na hodnotu N mimo slučky. Sú dve námietky požadujúce použitie BANZ inštrukcie v C5x :

1. Špecifikovanie a modifikovanie iných ako dekrementovanie o 1 môže mať za následok slučku, ktorá nikdy neskončí.

2. Inštrukcie, ktoré modifikujú pomocný register s pamäťovo mapovaným odkazom; napr. : SAMM, LMMR alebo SACL, uskutočňujú modifikáciu pri vykonaní radšej ako dekódovanie fázy. Preto, keď BANZ inštrukcia je uskutočnená, ako dva slová nasledujúce jednu z týchto inštrukcií, efekt týchto inštrukcií sa neodzrkadlí, keď je BRANCH vykonaná.

Odsek 4-4. Príklad na BANZ slučkovú kontrolu

$$Y = \sum_{n=1}^{5} X_n$$

```
        .bss    x,5
        LAR     AR1,#x
        LAR     AR2,#4
        MAR     *,AR1
loop :  ADD     *+,0,AR2
        BANZ    loop,*-,AR1
        SACL    y
```

4.1.2 Podprogramy

Volanie podprogramov sa líši od BRANCH v tom, že od volania sa vyžaduje poskytnutie dočasného odchodu zo sekvenčného vykonávania programu. Preto, keď volanie je uskutočnené, návratová adresa je zapísaná, od ktorej môže znovu začať vykonávanie programu po dokončení podprogramu. Keď ľubovoľná CALL inštrukcia je vykonaná, adresa ďalšej inštrukcie nasledujúcej po inštrukcii CALL je vložená do hardvérového zásobníka C5x. Podobne ako BRANCH, CALL môže byť nepodmienková alebo podmienková a môžu byť okamžité alebo oneskorené.

4.1.2.1 Volanie a podprogram

CALL inštrukcia je nepodmienkové volanie. Podobne ako nepodmienková BRANCH inštrukcia, ako aj zmena obsahu PC, CALL môže tiež modifikovať AR a ARP cez ARAU. Inštrukcia CALL je kódovaná následovne :

[label] CALL pma [, { ind } [, ďalší ARP]]

kde :

0000h ≤ pma ≤ 0FFFFh
0 ≤ ďalší ARP ≤ 7

Adresa inštrukcie na vykonanie nasledujúca po vykonaní podprogramu je vložená v zásobníku. Zásobník C5x je osemúrovňový hardvérový zásobník, ktorý sa používa len na nahrávanie návratových adries. Je zdielaný oboma volaniami a prerušeniami. Systémy, ktoré môžu prekročiť osemúrovňovú CALL / INTERRUPT hĺbku, musia nahrať hardvérový zásobník do dátovej pamäti ako časť ich obsahu SAVE / RESTORE rutinou. Tento proces je popísaný do detailov vnútri modulu PRERUŠENIA v tejto príručke.

4.1.2.2 Podmienkové CALL

Podobne ako BRANCH, CALL môže byť tiež podmienková. Podmienková CALL inštrukcia je kódovaná následovne :

[label] CC pma , [podmienka 1] [podmienka 2] [, ...]

Podmienkové CALL fungujú na nejakých podmienkových kódoch ako BCND a taktiež budú vykonané, keď všetky vybrané podmienky sú splnené. Keď ľubovoľná z vybraných podmienok nie je splnená, riadenie prechádza na inštrukciu po CC inštrukcii. Inštrukcia CC je podrobená nejakým obmedzeniam ako BCND inštrukcia.

4.1.2.3 Dynamická CALL

Inštrukcia CALA umožňuje volania adresy programovej pamäte obsiahnutej v akumulátore. Keď je inštrukcia CALAvykonaná, návratová adresa je uložená v zásobníku a nižších 16 bitov akumulátora sú nahraté v PC.

4.1.2.4 Návrat z podprogramu

Ako bolo spomenuté skoršie, inštrukcia volania nahraje návratovú adresu od ktorej bude znova pokračovať vykonávanie programu po skončení podprogramu. Rutina, do ktorej sa vstúpilo cez ľubovoľnú inštrukciu volania, môže byť normálne opustená cez inštrukciu RETURN. C5x poskytuje oba, t. j. podmienkové a nepodmienkové, návraty z podprogramov.

RET inštrukcia uskutočňuje nepodmienkový návrat z podprogramu. RET inštrukcia je kódovaná následovne :

[label] RET

Keď je RET inštrukcia vykonaná, obsah vrcholu zásobníka je vybraný do programového počítadla.

4.1.2.5 Podmienkový návrat z podprogramu

Inštrukcia RETC (return conditionally) funguje ako RET inštrukcia, okrem toho RETURN je vzatý len vtedy, keď špecifikované podmienky sú splnené. RETC je kódovaná následovne :

[label] RETC [podmienka 1] [, podmienka 2] [, ...]

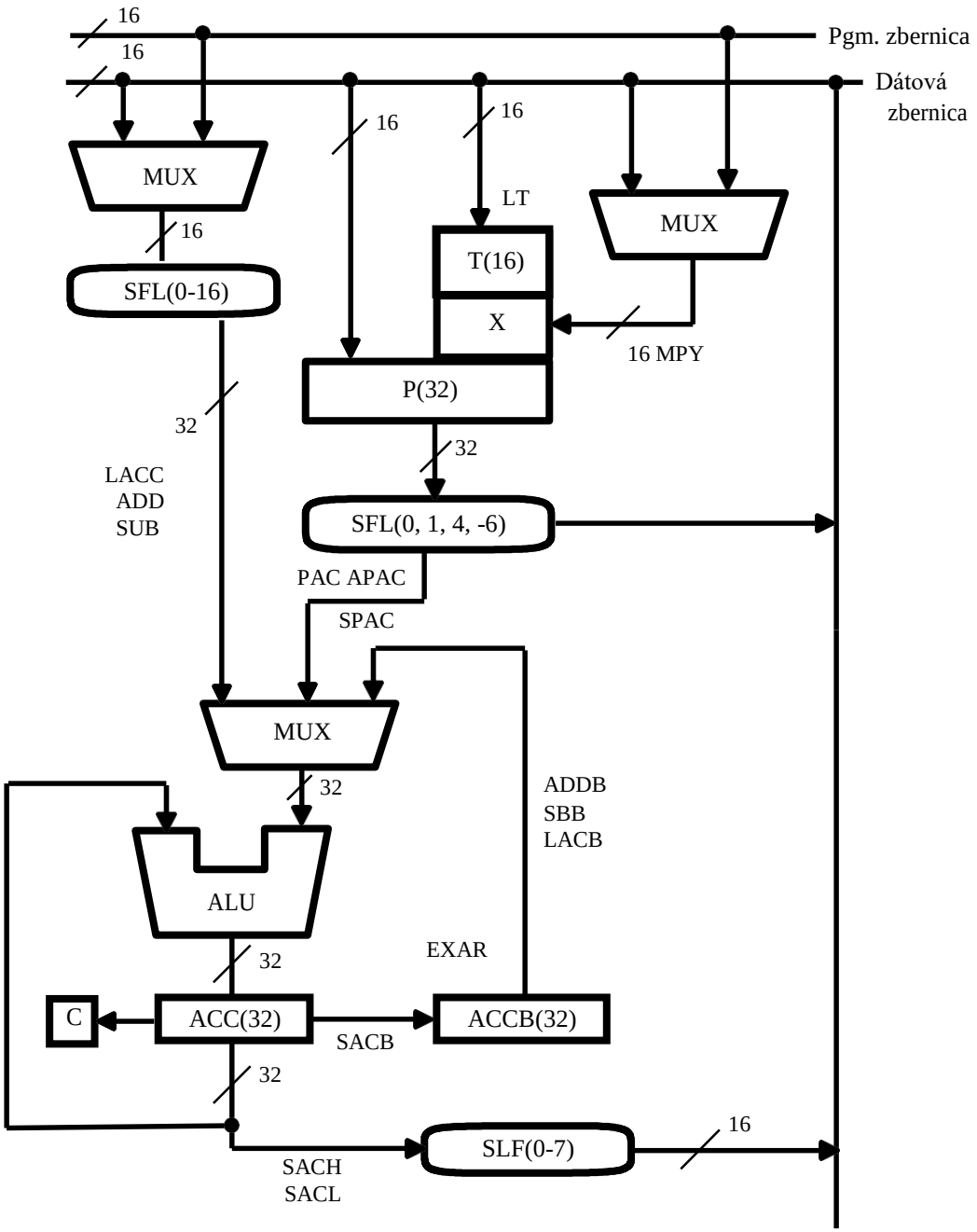
Podmienkové kódy a operácie sú identické ako pri BCND.

4.2 CALU

Centrálna aritmetická logická jednotka (CALU) je, kde hodnoty z dát a programovej pamäte môžu byť aritmeticky operované (sčítanie, odčítanie a násobenie). Registre a funkčné komponenty CALU pracujú spolu, čím umožňujú rýchlu implementáciu základných matematických procesov.

V tomto module bude niekoľko komponentov CALU vyšetrených, s cieľom byť schopný rozumieť prácu týchto komponentov a napísať kód, ktorý efektívne implementuje aritmetické rovnice.

Odsek 4-5. Centrálna aritmetická logická jednotka



4.2.1 Aritmetická logická jednotka

Aritmetická logická jednotka (ALU) je funkčný blok, ktorý vie uskutočniť sčítanie, odčítanie a Booleovské funkcie. Výsledky ALU sa objavia v registri akumulátor (ACC). Vstupy do ALU sú z ACC a buď pamäte (dát alebo programu), alebo register (Produkt alebo akumulátorový buffer). 32 bitov ALU a ACC pracujú so zápornými a kladnými hodnnotami do 32 bitov v jednom cykle. Nasledujúce sú príklady inštrukcií často používaných operovanie s akumulátorom


```
LACC    #123        ; ACC = 123 po inštrukcii.
                    ; Stará hodnota ACC je prepísaná
ADD     x           ; ACC = 123 + obsah < dma > "x"
SUB     *       ; ACC = 123 + @x - obsah < dma >
                    ; ukazujúci na aktuálny AR
```

Všetky tri adresné módy, ktoré tu boli ukázané, môžu byť použité v ktorejkoľvek z týchto inštrukcií. Navyše, keďže ACC je 32 bitový register a operandy z pamäte sú 16 bitové a druhý operand je sprístupnený potom, čo dátový operand, ktorý indikuje počet bitov zo spodnej časti akumulátora, s ktorým sa označí LSB operandu. V tomto module žiadny posun nebude ukázaný, takže základná nulová hodnota umiestni LSB operandu na LSB akumulátora. Pole posunov a jeho výhody budú preberané neskôr v 6. module a v TMS320C5x užívateľskom manuály.

Keď je ukončený výpočet, je často žiadúce uložiť výsledky späť do pamäte dát. Keďže pamäť má len polovičnú šírku slova oproti akumulátoru, je vhodné mať možnosť uložiť ACC do pamäte po poloviciach použijúc nasledujúce dve inštrukcie :

```
SACL    y       ; dolných 16 bitov akumulátora zapísané do < dma > "y"
SACH    z       ; horných 16 bitov akumulátora zapísané do < dma > "z"
```

Tieto inštrukcie môžu používať aj nepriamy adresný mód. Navyše je možnosť extrahovať ľubovoľných 16 za sebou nasledujúcich bitov z ACC cez výstupný posuvný register, ktorý je riadený druhým poľom operandov v SACH a SACL inštrukciách. Ako bolo spomenuté skôr, žiadne posuny nebudeme uvádzať v tomto module, uvažujúc základnú hodnotu 0. Odlišné príklady sú ukázané v module 6 a v TMS320C5x užívateľskom manuály.

Na rozdiel od väčšiny inštrukcií používajúcich priame adresovanie, LACC #const vždy používa dve slová (preto dva cykly) bez ohľadu na dlžku konštanty. Napríklad :

```
LACC    #2
LACC    #427Ah
```

Obe používajú dva slová programového miesta. Keď nahrávaš 8 bitovú (alebo menšiu) bezznamienkovú hodnotu, môžeš použiť namiesto nich LACL :

```
LACL    #2      ; jedno slovo, jeden cyklus
```

LACL nahrá do spodnej polovice ACC danú hodnotu a vynuluje hornú polovicu v jednom cykle. Toto je perfektný šetrič času, ale užívateľ musí pamätať na to, že LACL nepodporuje pre záporné čísla normálnu znamienkovú príponu cez vrch akumulátora.

4.2.2 32-bitová aritmetika

SAC a SACL operácie umožňujú ukladanie 32 bitových hodnôt (po poloviciach) do pamäte dát. Tieto dlhé operandy môžu byť použité na prácu s ACC ako nasledujúce vstupné hodnoty pre prenos, sčítanie a odčítanie použitím následovných inštrukcií :

Odsek 4-6. Inštrukcie pre 32 bitovú aritmetiku

```
LACL    < dma >     ; nahrá bezznamienkové < dma > do ACC
ADDS    < dma >     ; pripočíta bezznamienkové < dma > do ACC
SUBS    < dma >     ; odčíta bezznamienkové < dma > z ACC

LACC    < dma >, 16 ; nahraje hornú časť ACC s < dma >
ADD     < dma >, 16 ; pripočíta < dma > do hornej časti akumulátora
SUB     < dma >, 16 ; odčíta < dma > z hornej časti ACC
```

Ich použitie je demonštrované v následovnom príklade :
D32 = A32 + B32 - C32

```
.bss    A,2     ; Prideľuje dva 16 bitové miesta
.bss    B,2     ; pre každý údaj. V tomto príklade
.bss    C,2     ; prvé miesto je brané ako najviac
.bss    D,2     ; významové slovo.

LACC    A,16    ; nahrá hornú polovicu A do ACC
ADDS    A+1     ; súčet je bezznamienková dolná polovica A
ADDS    B+1     ; pričíta bezznamienkovú dolnú polovicu B
ADD     B,16    ; súčet v hornej polovici B
SUB     C,16    ; odčíta hornú polovicu C
SUBS    C+1     ; odčíta bezznamienkovú dolnú polovicu C
SACH    D       ; odloží horný výsledok
SACL    D+1     ; odloží dolný výsledok
```

Pamätajte, že poradie operácií so spodnou polovicou a vrchnou polovicou dlhého slova nie je podstatné. Toto je osvetlené príkladom pri pričítavaní B-termu v opačnom poradí ako ostatných termov.

4.2.3 Buffer akumulátora

Tridsaťdva bitový Buffer akumulátora (ACCB) je doplnok k akumulátoru. Môže byť použitý na presun 32-bitových hodnôt z a do akumulátora. Takýmto spôsobom je možné uloženie alebo načítanie 32-bitovej hodnoty v jednom cykle. Inštrukcie vkonávajúce túto funkciu sú :

```
SACB        ; uloží ACC do ACCB
LACB        ; načíta ACC z ACCB
EXAR        ; vymení obsahy ACC a ACCB
```


Ďalšie využitie ACCB je pre 32-bitovú aritmetiku. Hodnota z ACCB môže byť pričítaná alebo odčítaná od hodnoty v akumulátore použitím nasledujúcich inštrukcií :

```
ADDB        ; ACC = ACC + ACCB
SBB         ; ACC = ACC - ACCB
```

Konečne, ACCB možno využiť na vykonanie operácií minima/maxima použitím inštrukcií Compare Greater Than (CRGT) a Compare Less Than (CRLT). Po porovnaní s väčšou (menšou) hodnotou je väčšia (menšia) hodnota skopírovaná do ACC a tiež do ACCB. Taktiež, ak hodnota v ACC bola väčšia (menšia) alebo rovná hodnote v ACCB, je nastavený Carry bit (C) na hodnotu 1, v opačnom prípade je nastavený na 0. V nasledujúcich príkladoch sú uvedené inštrukcie a ich funkcia :

Príklad : Použitie ACCB a CRLT

```
                ; ACC = 01234567h
                ; ACCB = 07654321h
CRLT            ; ACC = ACCB = 01234567h,        C= 1
```

Príklad : Použitie ACCB a CRGT

```
                ; ACC = 01234567h
                ; ACCB = 07654321h
CRGT            ; ACC = ACCB = 07654321h,        C = 0
```

Odsek 4-7. Funkcia ACCB

Načítanie/uloženie	+/-	min/max
LACB SACB EXAR	ADDB SBB	CRGT CRLT

```
CRGT    ; porovnaj, či je väčší  ako
        1.  Ak ACC ≥ ACCB nastav  Carry
        2. ACC = ACCB = max ( ACC, ACCB )
```

4.2.4 Násobička

Násobenie už tradične vyžaduje mnoho inštrukčných cyklov, čo značne spomaľuje riešenie algoritmov obsahujúcich mnoho násobenia. Toto platí najmä pre DSP systémy, ktoré sa skladajú z množstva súčtov súčinov. Základnou črtou zariadení C5x je prítomnosť špeciálneho hardwaru, určeného na vykonanie násobenia v jednom cykle.

Násobič dostane dve 16-bitové hodnoty a uloží 32-bitový výsledok do 32-bitového registra súčinu. Výsledok v registri súčinu je väčšinou presunutý do akumulátora cez 32-bitovú zbernicu pred vytvorením ďalšieho súčinu. Počas tohto procesu môže byť výsledok rotovaný. V tomto module je rotácia zrušená (takáto situácia nastane po resete). Rotácia a jej vlastnosti budú prebraté v ďalšom module. Násobenie prebieha v dvojkovom doplnku, pokiaľ nie je použitá inštrukcia pre násobenie bez znamienka.

4.2.4.1 Základné použitie násobiča

Ako už bolo spomenuté, násobič potrebuje dve hodnoty. V najjednoduchšom prípade je prvý operand uložený do dočasného registra (T) (tiež TREG, TREGO) použitím inštrukcie LT. Operand inštrukcie LT môže byť určený priamou alebo nepriamou (ale nie immediate) adresáciou.

Druhý operand je špecifikovaný inštrukciou násobenia MPY pomocou priameho alebo nepriameho, alebo immediate adresovania. Po vykonaní inštrukcie MPY je vypočítaný 32-bitový súčin uložený do registra súčinu (P) (tiež PR alebo PREG).

Všimnite si, že TREG a vstup násobenia sú 16-bitové hodnoty, a preto nie je možné využiť rotovanie spolu s inštrukciami LT alebo MPY.

Následovné násobenie prepíše predchádzajúce výsledky, preto je potrebné presunúť výsledok z registra súčinu pred jeho prepísaním. Najčastejšie je súčin časťou súčtu súčinov, kde sú jednotlivé súčiny sčitované, preto je žiadúce presunúť súčin do akumulátora. Toto možno uskutočniť následovnými spôsobmi:

```
PAC             ; ACC = P
APAC            ; ACC = ACC + P
SPAC            ; ACC = ACC - P
```

Každá z uvedených inštrukcií pracuje s 32-bitovou hodnotou v jednom cykle
používajúc špeciálnu 32-bitovú cestu medzi P a ACC. Podľa predchádzajúcich inštrukcií možno súčet súčinov realizovať tak, ako ukazuje následovný príklad.

Odsek 4-8. Príklad súčtu súčinov

```
                    ; y = A.x1 + B.x2 + C.x3 + D.x4
LT      X1          ; T register = x1
MPY     A           ; P register = Ax1
PAC                 ; akumulátor = Ax1
LT      X2          ; T register = x2
MPY     B           ; P register = Bx2
APAC                ; akumulátor = Ax1 + Bx2
LT      X3          ; T register = x3
MPY     C           ; P register = Cx3
APAC                ; akumulátor = Ax1 + Bx2 + Cx3
LT      X4          ; T register = x4
MPY     D           ; P register = Dx4
APAC                ; akumulátor = y
```

4.2.4.2 Rozšírené využitie násobiča

Ako už bolo poznamenané, súčet súčinov je často jadrom procesu systémov DSP, ktorých výkonnosť je indikovaná rýchlosťou operácií. V uvedenom príklade každý "tap" (násobenie-uloženie) vyžaduje tri inštrukcie, teda možno očakávať rýchlosť, alebo benchmark tri cykly/tap. Hoci DSP je oveľa rýchlejší ako normálny mikroprocesor, často potrbujeme ešte rýchlejšiu funkciu. Toto je možné spojením LT a APAC funkcií, ako je uvedené v príklade na inštrukciu LTA (Load Temporary and Accumulate). Existujú tri verzie inštrukcie LT s uložením výsledku :

```
LTP     X           ; ACC = P, T = obsah pozície x alebo "@x"
LTA     *           ; ACC = ACC + P,  T = hodnota, na ktorú
                    ; ukazuje súčasný AR, alebo "*AR(ARP)"
LTS     *-,AR3      ; ACC = ACC - P, T = *AR(ARP), zníž AR(ARP),
                    ; ARP = 3
```

Je potrebné si uvedomiť, že načítanie n-tého údaju je vykonané súčasne s uložením predchádzajúceho (n-1)-ého údaja. Potom bude využitie inštrukcie LTA z predchádzajúceho príkladu vyzerať následovne :

Odsek 4-9. Príklad na rozšírené využitie súčtu súčinov

```
                        ; y = Ax1 + Bx2 + Cx3 + Dx4
LT      X1              ; T = x1
MPY     A               ; P = Ax1
LTP     X2              ; ACC = Ax1, T = x2
MPY     B               ; P = Bx2
LTA     X3              ; ACC = Ax1 + Bx2, T = x3
MPY     C               ; P = Cx3
LTA     X4              ; ACC = Ax1 + Bx2 + Cx3, T = x4
MPY     D               ; P = Dx4
APAC                    ; ACC = y
```

Použitím LTA namiesto LT a APAC nemá nijaké nevýhody (okrem prípadného zmätku alebo chyby spôsobenej programátorom), preto je vo väčšine prípadov uprednostňované kôli kratšiemu času vykonávania a zmenšenú dlžku kódu.

4.3 Laboratórne cvičenie 4 : Základné programovanie

Cieľom tohto cvičenia je precvičovanie funkcie aritmetiky TMS320. V tomto procese rozšírime .AMS súbor z predchádzajúceho cvičenia o nové funkcie.

Cieľom je pridanie kódu potrbného na získanie súčtu súčinov n hodnôt z každého poľa, alebo :

$$y = \sum_{n=1}^{4} (data_n * coeff_n)$$

Ako v predchádzajúcich cvičeniach, uvažujte o vhodnom spôsobe adresácie pre danú úlohu.

Ako doplnkovú úlohu skúste nájsť najväčší súčin a uložte ho na pozíciu x. Nezačnite pracovať na tejto časti pokiaľ nemáte úspešne splnenú prvú časť.

Môžete cvičenie absolvovať len na základe tejto informácie alebo využiť nasledujúci postup. Riešenie, ak bude potrebné, je uvedené na konci cvičenia.

4.3.1 Postup

1. Skopírujte LAB3.ASM do LAB4.ASM. Zmeňte LAB4.ASM tak, aby násobil údajové pole poľom koeficientov a uložil výsledok do pamäte.

a) Otvorte súbor LAB4.ASM.
b) Zmažte operácie modifikujúce PMST zo sekcie .text.
c) Použite cyklus BANZ v inicializačnom procese.
d) Nastavte ukazovatele na začiatok datového a koeficientového poľa.
e) Vynásobte prvú dvojicu hodnôt a presuňte výsledok do akumulátora.
f) Opakujte pre ostatné páry (Nepoužite BANZ v rutine MPY).
g) Uložte výsledok do pamäte.

2. Preložte a zlinkujte program.
3. Odlaďte váš kód na simulátore.
4. Doplnková úloha : Získajte maximum zo súčinov.

a) Nastavte ukazovatele na začiatky polí.
b) Vynásobte prvý pár a umiestnite výsledok do Buffra akumulátora.
c) Porovnajte nasledujúci súčin s ACCB a uchovajte väčšiu hodnotu.
d) Uložte výsledok do pamäte.

4.4 Prehľad

Odsek 4-10. Prehľad

Nové inštrukcie - modul 4

LACB	ADDB	SBB	SACB
EXAR	CRGT	CRLT	
PAC	APAC	SPAC	MPY
LTP	LTA	LTS	LT
B	CALL	RET	BANZ

Kapitola 5

ZDOKONALENÉ PROGRAMOVACIE TECHNIKY

Odsek 5-1 učebné ciele:

Na konci tejto kapitoly by ste mali byť schopní kontrolovať vykonávanie programu rôznymi cestami a efektívne prenášať obsah jednej časti pamäťovej mapy (máp) do inej.

Ďalšie ciele kapitoly zahrňujú schopnosť písania programov v jazyku symbolických adries (JSA) používajúc:

- Zdokonalené použitie násobiacich operácii pre dlhé sumy súčinov
- Opakovacie funkcie, ktoré optimalizujú cyklus
- Oneskorené operácie, ktoré šetria čas procesora
- Podmienené operácie, ktoré určia či sa má alebo nemá vykonať krátky kódový segment
- Operácie na presun blokov, ktoré kopírujú hodnoty z:

  -Programovej do dátovej pamäte
  -Dátovej do programovej pamäte
  -Dátovej do dátovej pamäte

5.1 Zdokonalené použitie násobičky

Pretože algoritmus sumy súčinov sa často vyskytuje v numerických procesoch, je žiaduce aby sme boli schopní vykonať funkciu násobenia a uchovania v čo najkratšom čase. V predchádzajúcich častiach sme videli, že funkcia násobenia a ukladania môže byť vykonaná na ´C5x v dvoch alebo troch cykloch. Ešte lepší výkon by sa mohol dosiahnuť, ak by sa táto funkcia vykonala v jedinom cykle.

5

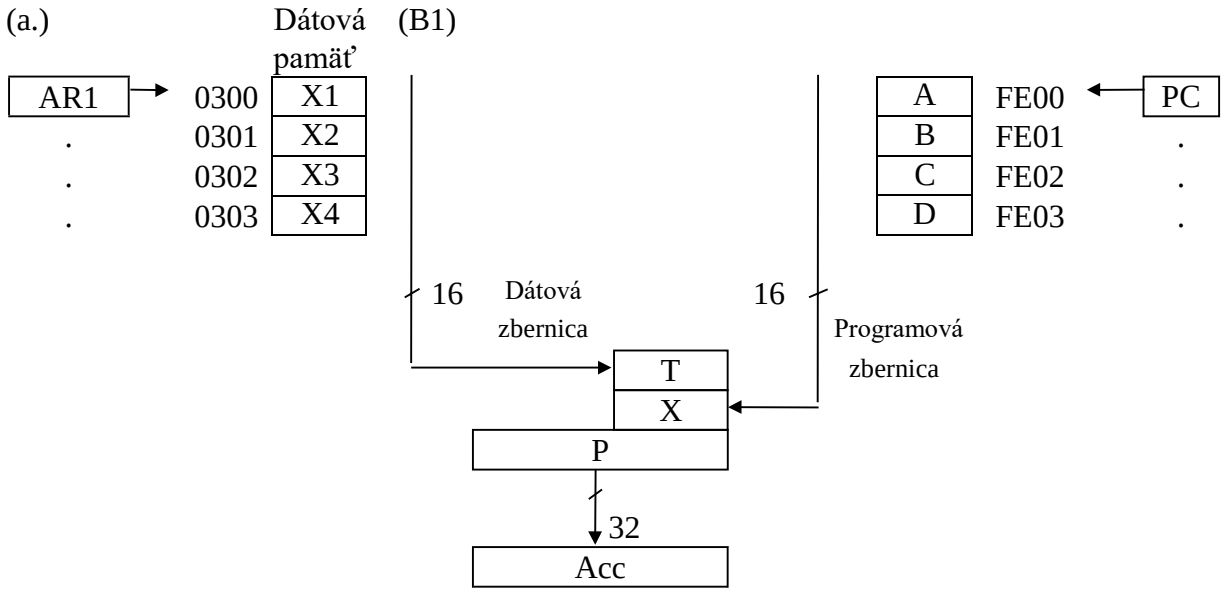
Predpokladajme tieto požiadavky funkcie násobenia a uloženia: v jedinom cykle musia byť načítané dva operandy, poslané do násobičky a výsledok uložený. Inštrukcia LTA vykonáva mnoho z týchto funkcií. Výsledok ukladá do a operand číta z dátovej pamäti. Takže všetko čo ostáva ešte spraviť je načítať druhý operand a postať do násobičky. Nanešťastie, čítanie prvého operandu blokuje použitie dátovej zbernice. Ničmenej, keďže ´C5x je stroj Harvardskej architektúry, má ešte jednu zbernicu. Táto zbernica (programová) môže byť použitá aby bol ňou poslaný operand, ale potom musí tento operand spočívať v programovej pamäti aby programová zbernica mala k nemu prístup. Takže jediná cesta ako vybrať miesto z programovej pamäti je cez programový čítač (PC). Normálne operácie programového priestoru - vyvolávajúce inštrukcie - musia byť dočasne suspendované. Táto operácia sa vykoná inštrukciou MAC a vyžaduje tri cykly na vykonanie kôli vyprázdneniu programového čítača:

```
                                ; ekvivalent :
MAC   <pma>, <dma>              ; APAC, LT <dma>, MPY <pma>
```

Rýchlosť vykonania inštrukcie MAC sa môže priblížiť k rýchlosti jedného cyklu ak by sa pracovalo s opakovacou (RPT) inštrukciou. V opakovacej slučke, zdielanie PC medzi operandom a vyvolávajúcou inštrukciou je znemožnené. Pretože opakovaná inštrukcia je vyvolaná len raz, uložená v „inštrukčnom registre“, a opakovaná. Takže prvá inštrukcia MAC vyžaduje tri cykly, ale všetky následujúce MAC inštrukcie vnútri RPT slučky vystačia s jedným cyklom. Toto môže byť zobrazené schématicky a vykonané v ´C5x kóde, ako je to znázornené na obrazku na ďalšej strane.

Tým, že inštrukcia MAC môže byť vykonaná v jednom cykle, sa občas predpokladalo, že použitie MAC je vždy rýchlejšie než LTA a MPY. Ničmenej, ak zoberiemie do úvahy len niekoľko opakovaní slučky, administratívnosť (a zložitosť) v nastavení inštrukcie MAC spraví z LTA/MPY preferovanú možnosť. Také hlavné pravidlo je, že ak máme viac ako desať opakovaní slučky, použijeme inštrukciu MAC. Ináč použijeme inštrukcie LTA/MPY. Nakoniec, koncepcia použitia dekrementu a vetvenia s LTA/MPY sa niekedy aj predpokladá. Pretože vetvenia vyžadujú násobné cykly na vykonanie a neprispievajú číselnému riešeniu, všeobecne sa im vyhýba v systémoch, kde sa kladie dôraz hlavne na rýchlosť vykonávania kódu.

Obrázok 5-2. Príklad zdokonalenej sumy súčinov



(b.)

```
                LAR   AR1, #300h
                MAR * ,AR1
APAC            RPTZ  #3
LT    *+        MAC 0FE00h, *+
MPY   0FE00h    APAC
```

Poznámka: Tento program používa fixné adresy čo je v praxi neodporúčané s procesom COFF.

5.1.1 Zdokonalené násobenie s dynamickým zdrojom

Inštrukcia MAC má tú výhodu, že v jednom cykle môže vynásobyť a uložiť, ale občas sa vyskytujú niektoré obmedzenia: <pma> je špecifikovaná ako pevná hodnota a nemôže byť (jednoducho) modifikovaná cez programové riadenie. Schopnosť „dynamicky“ zmeniť <pma> je možné pomocou inštrukcie MADS, ktorá odkazuje obsah BMAR (Block Move Address Register) ako adresu do poľa v programovej pamäti. S ohľadom na ostatné, je táto inštrukca takmer identická s inštrukciou MAC. Následujúci obrázok ukazuje použitie inštrukcie MADS s miestom „x“ za predpokladu, že bolo nahraté (nejakou inou rutinou) so štartovacou adresou požadovaného poľa do programovej pamäti.

Obrázok 5-3. Príklad sumy súčinou použitím MADS

```
LACC    X           ; Do ACC je uložená adresa pamäťového
                      miesta X.
SAMM    BMAR        ; Ulož ACC do Memory Map register
                      „BMAR“
MAR     *, AR1      ; Použi AR1
LAR     AR1, #300h  ; nahraj AR1
RPTZ    #31         ; vymaž ACC a P, opakuj 32 krát
MADS    *+          ; vynásob/ulož , inkrementuj adresu
APAC                ; pridaj posledný výsledok
```

5.1.2 Ďalšie operácie násobičky

Niekoľko ďalších inštrukcí využíva funkciu násobenia. Jedna skupina poskytuje schopnosť vykonania funkcie „umocnenia a uloženia“. V tejto funkcii je špecifikovaný jeden operand a predaný do oboch vstupov násobičky. Výsledná hodnota je umiestnená v sučinovom registre a vrátená hodnota P je pripočítaná k akumulátoru a doň uložená. Táto funkcia, ktorá je užitočná pre mocninové funcie a iné operácie, sa často objavuje s vnútornými opakovacími slučkami. Iná skupina, funkcie „násobenia a sčítania“, sú kombináciou inštrukcií APAC a MPY. Tieto boli úspešné pri vykonávaní súčtu v adaptívnych filtroch LMS pre aktualizáciu LMS zatiaľ čo sa simultánne vykonávalo násobenie v odbočke filtra. Tu je niekoľko príkladov použitia týchto inštrukcií.

```
MPYA    x     ; ACC = ACC+P , P = x*T
MPYS    *     ; ACC = ACC–P , P = T*(*AR(ARP))
SQRA    y     ; ACC = ACC+P , P = y*y
SQRS    *+    ; ACC = ACC–P, P=square*AR(ARP), AR(ARP) = AR(ARP)+1
MPYU    n     ; bezznamienkové mpy: P = ABS (T*N)
```

5.2 Operácie opakovania

Okrem inštrukcií vetvenia a volania, ´C5x môže vykonať „hardwerové slučky“, čo znamená schopnosť opakovane vykonať kód bez použitia vetvenia vnútri kódovej slučky. V ´C5x, tieto slučky môžu pozostávať z jednoriadkóvého kódu alebo z bloku niekoľkých kódových riadkov.

5.2.1 RPT - Opakovanie nasledujúcej inštrukcie

Inštrukcia RPT je použitá na usmernenie ´C5x aby opakoval inštrukciu ktorá nasleduje, určitý počet krát. RPT má nasledujúce charakteristiky:

- 2-cyklová inštrukcia
- Nahrá RPTC (MMR) s 8 alebo 16 bitovou hodnotou
- Nasledujúca inštrukcia je opakovaná RPTC+1 krát
- Vytvorí „Nultú doplnkovú slučku“ v jednej ištrukcii
- Hodnota opakovania je obyčajne špecifikovaná krátkym alebo dlhým priamym operandom
- Hodnota opakovania môže byť tiež špecifikovaná priamym alebo nepriamym adresovaním
- Operácie MAC a block (BLxx) redukuje na jednocyklové po prvom opakovaní.
- Opakovacia sľučka je neprerušiteľná

V nasledujúcom príklade, inštrukcia RPT je použitá s inštrukciou MAC. Všimnite si, že:

- Vykoná sa päť opakovaní ak je argument inštrukcie RPT nastavaný na hodnotu štyri.
- V prvom opakovaní má inštrukcia MAC tri cykly a jeden cyklus v ďalších opakovaniach.

5 *Obrázok 5-4 Príklad použitia RPT s MAC*

$$\sum_{n=1}^{5} X_n * coeff_n$$

```
        .text
loop:   LAR     AR1, #x
        MAR     *,AR1
        ZAP
        RPT     #4
        MAC     coeff, *+
        APAC
        B       loop
        .data
coeff:  .word   1,2,3,4,5
        .bss    x, 5
```

Inštrukcia RPT nemôže byť použitá v žiadnom vetvení alebo inštrukcii priameho operandu. Pretože používanie týchto typov inštrukcií s RPT funkciou neprináša úžitok, nestráca sa rozsah výkonu, ale treba poznamenať, že takéto počínanie môže nechať procesor v nedefinovanom stave. Všetky inštrukcie ktoré sa nemôžu použiť spolu s RPT sú vypísane v *Uživatelskej príručke TMS320C5* tabuľka 3-11, strana 3-45.

5.2.2 RPTZ - Opakuj nasledujúcu inštrukciu a vynuluj ACC a P-register

Operácia RPTZ je podobná RPT s nasledujúcimi výnimkami:

- Používa len dlhé priame operandy na špecifikovanie RPTC hodnoty
- Vynuluje súčinový register a akumulátor počas nahratia hodnoty RPTC
- Šetrí cykly v niektorých matematiských funkciách

5.2.3 RPTB - Opakuj blok

Schopnosť opakovať skupinu troch alebo viacerých inštukcií je zabezpečená inštrukciou RPTB (Repeat Block).

Na vykonanie opakovania bloku musíme poznať 4 parametre:

- Počiatočnú pozíciu slučky
- Koniec slučky
- Počet opakovaní slučky
- Či blok opakovania je zapnutý alebo vypnutý

Na uchovanie tejto informácie sú použité štyri MMR

- PASR: Program Address Start Register (register štartovacej adresy programu)
- PAER: Program Address End Register (register koncovej adresy programu)
- BRCR: Block Repeat Count Register (register počítania opakovaní bloku)
- BRAF: Block Repeat Active Flag in the PMST (Program Mode Status register) (Aktívny príznak bloku opakovania v PMST (registri stavového módu procesora))

Inštrukcia RPTB automaticky nastaví BRAF, a jeho operand bude špecifikovať hodnotu na nahratie do PAER. Do PASR je automaticky uložená adresa nasledujúcej inštrukcie RPTB. Všetko čo ešte ostáva špecifikovať je hodnota, ktorá sa nahrá do BRCR. Toto sa spraví (niekde) pred inštrukciou RPTB, obyčajne cez inštrukciu SAMM alebo SPLK, ako ukazuje nasledujúci príklad.

Obrázok 5-5. Príklad použitia RPTB

$$y = \sum_{n=1}^{5} X_n * coeff_n$$

```
        .text
loop2:  LAR     AR1, #coeff
        LAR     AR2, #x
        LACC    #4
        SAMM    BRCR
        ZAP
        MAR     *, AR2
        RPTB    next-1
        LT      *+, AR1
        MPY     *+, AR2
        APAC
next:   SACL
        B       loop2
        .bss    x, 5
        .bss    y, 1
        .data
coeff:  .word   a0, a1, a2...
```

Všimnite si ako je PAER nahratý. Predpokladajme možnosť, že posledný riadok kódu bol dvoj-slovová inštrukcia. Ak by sme ho označili návestím a použili ho ako hodnotu PAER, druhá časť slova inštrukcie by **nebola** vyvolaná a výsledok by bol neprípustné (neplatné) vykonanie. Aby sme sa vyhli tomuto problému, programátorom sa odporúča označiť prvý riadok kódu za smyčkou NÁVESTIE a použiť hodnotu NÁVESTIE-1 ako operand inštrukcie RPTB.

Tu sú niektoré predpoklady týkajúce sa bloku opakovania:

5

- Na rozdiel od RPT, RPTB **je** prerušitelná
- Pretože používajú rôzne registre, jednotlivé RPT inštrukcie sa môžu včleniť do vnútra slučiek RPTB
- Vkladanie ďalších RPTB vo vnútri RPTB slučky sa nedoporučuje, pretože ukladanie/výber kontextu asociovaných registrov by zabralo viac času než samotné vetvenie.
- RPTB sa všeobecne používa v najvnútornejších slučkách v členovacom systéme kvôli najväčšiemu užítku z výkonu.
- Volanie podrutín a prerušenia su povolené vo vnútri bloku opakovania.
- **Veľkosť bloku MUSÍ byť najmenej tri slová.**

5.3 Zdokonalené programové riadenie

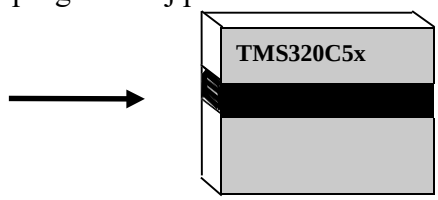
Pre systémy, ktoré vyžadujú najvyšší možný výkon, existuje mnoho inštrukcií ktoré ho môžu zlepšiť. Na príklad štandartné 4-cyklové inštrukcie vetvenia môžu byť nahradené 2-cyklovými **oneskorenými** vetveniami. Ďalšie príklady zahrňujú podmienené vykonanie a inštrukcie opakovania.

5.3.1 Vyšetrovanie problému zreťazenia

Pred tým než začneme diskutovať o inštrukciách volania a vetvenia, bolo by vhodné v krátkosti si povedať niečo o povahe zreťazenia ´C5x.

´C5x operuje so štvor-slovovým zreťazením. Ak je vykonávanie programu sekvenčné, zreťazenie je v podstate neviditeľné. V hociktorom danom procesorovom cykle môžu byť len 4 inštrukcie aktívne. Tento problém je ukázaný na obrázku referovanom nižšie. V prípade, že je zreťazenie plne využívané, počas vykonávania aktuálnej (N) inštrukcie, načíta sa ďalší operand pre ďalšiu inštrukciu (N+1) aby sa vykonala; ďalšia inštrukcia (N+2) sa dekóduje, zatial čo ďalšia (N+3) sa vyvoláva z programovej pamäti.

Štvor-úrovňová operácia zreťazenia
obr. 3-14, strana 3-35



Vo fáze dekódovania, okrem dekódovania inštrukcie, ARAU vykoná akékoľvek špecifikované modifikácie pomocných registrov.

5.3.1.1 Nespojitosti zreťazenia

Ako bolo povedané pred tým, operácia zreťazenia je v podstate neviditelná ak je program vykonávaný sekvenčne. Štvor-slovové zreťazenie zabezpečuje efektívny inštrukčný vykonávací pomer jednej inštukcie za cyklus. Keď sekvenčná operácia je prerušená, je prerušený tok zreťazenia, a zreťazenie musí byť znova aktualizované ešte pred tým, než vykonávanie bude pokračovať. Oneskorené vetvenia spôsobujú nespojitosti programového čítača bez porušenia zreťazenia.

5.3.1.2 Sledovanie zreťazenia v simulátore

Simulátor ´C5x priamo neumožňuje sledovací režim zreťazenia. Hoci umožňuje použitie špeciálnych premenných na sledovanie zreťazenia. Otvorte si okno WATCH pre nasledujúce premenné:

```
fins    /* fetch instruction opcode */ (operačný kód vyvolávacej inštrukcie)
dins    /* decode instuction opcode */ (op. kód dekódovacej inštrukcie)
rins    /* read operand opcode */ (op. kód inštrukcie čítania)
xins    /* execute instruction opcode */ (op. kód vykonávanej inštrukcie)
faddr   /* fetch address */ (vyvolávacia adresa)
daddr   /* decode address */ (dekódovacia adresa)
raddr   /* read address */ (adresa čítania)
xaddr   /* execute address */ (adresa vykonávania)
```

5.3.2 Oneskorené vetvenia

Na realizáciu oneskorených vetvení sa doporučuje nasledobné:

1. Pri písaní kódu používať štandartné vetvenia.

2. Preveriť vlastné vykonanie kódu.

3. Pridať „D“ vo vetviacej mnemotechnickej skratke (napr.: CALL bude CALLD)

5

4. Pridať dve NOP inštrukcie po oneskorenej operácii s komentárom „ ; tu nastáva vetvenie“.

5. Nahradiť jeden alebo obidva NOP-y s jedným alebo dvoma slovami z predchádzajúceho kódu.

Je veľmi **dôležité** aby programátor vedel, koľko slov zaberajú použité inštrukcie v riadku oneskoreného vetvenia. Všimnite si, že parameter ktorý počítame sú **slová a nie cykly**.

Použitie oneskorených vetvení spôsobí programátorovi ďalšiu starosť s riadením zreťazenia, zaťial čo štandartné vetvenia ošetrujú zreťazenie automaticky.

Tu sú ďalšie predpoklady týkajúce sa inštrukcií oneskoreného vetvenia:

- Žiadna inštrukcia vetvenia nemôže byť použitá v oneskorovacom riadku.

- Pre podmienené operácie, inštrukcie oneskorovacieho riadku ovplyvnia status po vykonaní rozhodnutia vetvenia. (Takto by sa v slučke vyskytlo oneskorené opakovanie predpokladjúc, že staus nebol nasledovne modifikovaný.)

- Ak dve slová kódu hneď pred vetvením nie sú vhodné na použitie v oneskorovacom riadku, predpokladajme možnosť znovu upravenia kódu alebo použitie NOP-u ak môže byť použité len jedno slovo kódu.

5.3.3 Podmienené vykonanie (eXecute Conditional)

Ešte rýchlejšie podmienené vetvenie môže byť implementované do kódového segmentu dĺžky jedného alebo dvoch slov. Inštrukcia podmieneho vykonania (XC) je umiestnená priamo pred jeden alebo dva riadky kódu, ktorý má byť podmienene vykonaný. XC má dva operandy: počet slov v podmienenom segmente a zoznam podmienok (z tej istej skupiny ako podmenené vetvenie.)

Pri XC inštukcii ak všetky podmienky sú pravdivé, povoluje vykonanie jedného alebo dvoch slov, ktoré nasledujú za XC inštrukciou. Ak je nesplnená hociktorá podmienka, potom špecifikovaný počet inštreukcií po XC sú (efektívne) nahradené NOP-mi.

Ďalšie detaili o XC:

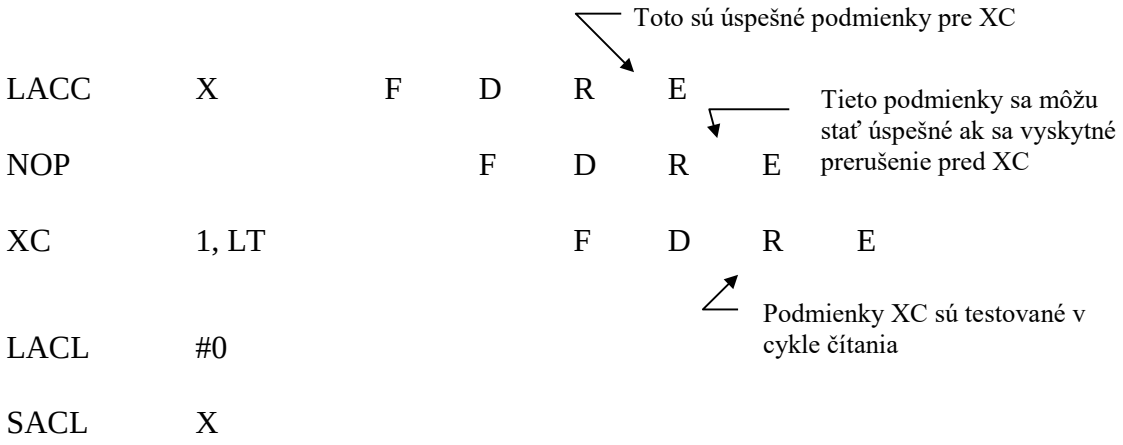
XC je jedno-slovová, jedno-cyklová inštrukcia.

XC používa rovnaký počet cyklov pre testovanie pravdivosti a nepravdivosti podmienky.

XC testuje podmienky jeden cyklus **pred** ich vykonaním; takže inštrukcia ktorá nasleduje hneď po XC **neovplyvní výkon XC**.

Obrázok 5-6 Inštrukcia podmieneného vykonania

[návestie] XC k, [podm1], [podm2], ... ;k=1 alebo 2



POZOR: Cyklus nasledujúci hneď za XC nemôže nastaviť podmienky pre vykonanie kôli zreťazeniu.

5.3.4 Výsledky zreťazenia

Ako bolo povedané, ´C5x využíva zreťazenie (pipeline) na dosiahnutie vysokej rýchlosti. Návrhári ktorí predtým pracovali s procesormi zreťazenia, sa mohli stretnúť s problémami pri zabezpečovaní aby sa inštrukcie nevykonávali v nesprávnom poradí, alebo nejakým iným neočakávaným spôsobom. Ak sa ´C5x programuje svojimi štandartnými inštrukciami, pracuje bez očividných efektov zreťazenia, čím odbremeňuje programátora pred testovaním problémov tohto typu. Ak sa vykonali niektoré so špecialnych inštrukcií ´C5x ako napr. BD a XC, užívateľ je priamo konfrontovaný s výsledkami zreťazenia a preto musí programovať s týmto v mysli, aby dosiahol požadovaný výsledok. V tomto bode by bolo vhodné ukázať tie procesy, ktoré vytvárajú výsledky zreťazenia. Je potešujúce poznamenať, že okrem týchto procesov ´C5x operuje ako keby tam žiadne zreťazenie nebolo.

Tie operácie, ktoré ovplyvňujú zreťazenie sú väčšinou tie, ktoré majú efekt vo vykonávacej fáze, ale majú tendenciu ovplyvňovať predošlé fáze. Napríklad oneskorené vetvenia (BD) majú efekt počas vykonávania, ale operujú v prvej fáze vetvenia. Takže tri udalosti sa prihodia ešte pred tým, než sa vetvenie vyskytne. Ako sme videli, jednoduché pravidlá nám umožňujú BD používať bezpečne a menej nároční programátori môžu používať štandatrné vetvenie (B) aby sa vyhli problémom s zreťazením úplne (za cenu niekoľkých cyklov navyše).

Podmienené vykonanie (XC) je ďalší taký príklad, XC musí kontrolovať podmienky počas svojej čítacej fáze (jedna fáza pred vykonaním), spôsobujúc efekt vynechania hneď nasledujúcej inštrukcie, ako bolo uvedené hore.

Ďalší možný výsledok zreťazenia môžeme pozorovať pri zapisovaní do adresných registrov (AR) cez interface pamäťovo mapovaného registra (MMR). Toto je kôli zápisu do adr. registrov na báze MMR, ktorý prebieha počas vykonávacej fáze, ale v AR sú väčšinou používané v dekódovacej fáze - čiže o dva cykli skôr. Preto používanie AR v tomto cykle priamo nasledujúc zápis MMR do AR bude odkazovať predošlú a nie očakávanú hodnotu AR. Jednoduchá cesta ako sa vyhnúť tomuto problému, je použiť jednoduchú a vhodnú inštrukciu LAR, ktorá sa plánovala vykonať skôr a zabrať jeden cyklus navyše, čím by sa eliminoval problém zreťazenia pri nahrávaní do AR za predpokladu, že sa použila inštrukcia LAR namiesto nahrávaní do AR metódou MMR.

Normalizačná inštrukcia (NORM) tiež využíva adr. registre modifikované počas vykonávania, a mala by byť umiestnená v kóde tak, aby sa žiadne modifikovanie AR alebo ARP nevyskytlo dva cykli po operácii NORM.

Konečne, ak programátor zmení konfiguráciu pamäte cez PMST bity, toto môže vytvoriť výsledky zreťazenia ak sa kód pokúša vyvolať z týchto pamätí počas troch cyklov pamäťového prepínača, pretože režim PMST sa vyskytuje vo vykonávacej fáze a vykonanie by nastalo o tri cykli skôr.

5.4 Operácie na presun blokov

Pretože ´C5x operuje v dvoch pamäťových priestoroch je niekedy dôležité presunúť pole z jedného pamäťového miesta na iné. Taký najbežnejší príklad je inicializácie dátovej pamäti RAM z programovej ROM. ´C5x môže kopírovať do dátovej RAM z ROM umiestnenej v dátach alebo programovom priestore.

Aj keď inicializáčné hodnoty sú funkčne asociované s dátami, fakt, že potrebujú byť umiestnené v pamäti typu ROM by bolo výhodné, ak by mohli ostať v tej istej ROM, ktorá by mohla byť použitá na uchovanie programového kódu. Existujú tri „blokové“ inštrukcie na kopírovanie polí:

- BLPD: Program memory to data memory (prog. pamäť do dátovej pamäte)
- BLDP: Data memory to program memory (dátova pamäť do programovej pamäte)
- BLDD: Data memory to data memory (dátová pamäť do dátovej pamäte)

Príkazy na presun blokov sú akosi zle pomenované, pretože vykonávajú kopírovacie a nie presunové operácie a vykonávajú jednoduchú (nie blokovú) pamäťovú operáciu. Ak sa znásobia opakovacou operáciou, potom blokové inštrukcie môžu efektívne kopírovať celé polia z jedného miesta na iné.

Blokové inštrukcie sa vykonávajú v dvoch alebo troch cykoloch. Ničmenej, v opakovacej slučke len prvá inštrukcia vyžaduje viac cyklov, všetky ďalšie opakovania sa vykonajú po jednom cykle.

Proces presunu bloku vyžaduje špecifikovanie troch parametrov.

- Umiestnenie poľa z ktorého sa bude kopírovať
- Umiestnenie poľa do ktorého sa bude kopírovať
- Počet sekvenčných slov v danom poli

Bloková inštrukcia špecifikuje zdrojovú a cieľovú adresu rôznymi cestami. Veľkosť bloku môže byť špecifikovaná ako argument inštrukcie RPT. Detaili o každej blokovej inštrukcii a ich použití nasleduje ďalej.

5.4.1 BLPD - Kopírovanie z programovej do dátovej pamäte

BLPD je akosi podobná inštrukcii MAC v tom, že sa tu špecifikujú dva operandy:

1. Adresa do poľa v programovom priestore

2. Adresa do poľa v dátovom priestore

5

<pma> môže byť explicitná 16 bitová adresa alebo môže použiť „Block Move Address Register (BMAR)“ ako ukazovateľ <pma>. <dma> môže byť špecifikovaná cez priame alebo nepriame adresovanie, aj keď nepriame adresovanie sa takmer vždy používa, pretože umožňuje špecifikovať kopírovanie do inkrementujúcich adries.

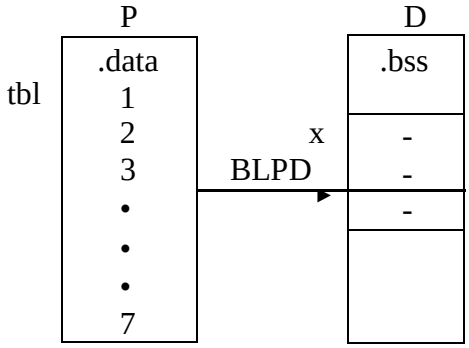
Podobne ako s MAC, PC je jediný ukazovateľ do programového priestoru ak je použitý v opakovacej sľučke, <pma> sa automaticky zväčší po každom opakovaní; pôvodný obsah PC sa obnoví po blokovej operácii. Pre MAC alebo blokové operácie hardwareový zásobník nie je vyžadovaný.

Čas vykonania sú tri cykli ak je <pma> špecifikovaná priamo a dva cykli ak je špecifikovaná pomocou BMAR. Ako bolo uvedené, v opakovacej sľučke len prvá bloková operácia bude vyžadovať viac cyklov, ostatné sa vykonajú po jednom cykle pre každé opakovanie.

V príklade na nasledujúcom obrázku je BLPD použité na inicializáciu bloku RAM z tabuľky ROM. Všimnite si potrebu položenia znaku (#) pred priamym operandom *tbl* v inštrukcii BLPD

Obrázok 5-7 Príklad použitia BLPD

```
        .bss    x,7
        .data
tbl:    .word   1,2,3,4,5,6,7
lenght  .set    $-tbl

        .text
        MAR     *,AR1
        LAR     AR1, #x
        RPT     #lenght-1
        BLPD    #tbl, *+
```



alebo pri použití BMAR, použite tento text namiesto predošlého:

```
.text
LACC    #tbl
SAMM    BMAR
MAR     *, AR1
LAR     AR1, #x
RPT     #lenght-1
BLPD    BMAR, *+
```

5.4.2 BLDP - Kopírovanie z dátovej do programovej pamäte

Inštrukcia BLDP je v celku podobná BLPD s nasledujúcimi rozdielmi:

- Tok je obrátený, kopírovanie prebieha z dátovej do programovej pamäte
- <pma> môže byť špecifikovaná len pomocou BMAR
- Pretože <pma> je definované, BLDP vyžaduje len jeden operand: <dma>

Čas vykonania je taký istý ako u BLPD. Pretože musí byť použitý BMAR, jedna inštrukcia BLPD sa vykoná v dvoch cykloch a po jednom cykle pri opakovaní s RPT.

5.4.3 BLDD - Kopírovanie z dátovej do dátovej pamäte

Niekedy je výhodné kopírovať jednu časť dátovej pamäte do inej. Ako príklad by bol presun informácie medzi vnútornou a vonkajšou pamäťou. Inštrukcia BLDD vykoná túto funkciu takou istou cestou ako BLPD s nasledujúcimi rozdielmi:

- Každý operand špecifikuje <dma>
- Prvý operand je zdrojová adresa a druhý cieľová adreesa.
- Jeden operand je špecifikovaný cez priamu adresu a obsah BMAR.
- Druhý operand je špecifikovaný cez priame alebo nepriame adresovanie.
- Vykonanie trvá dva cykly pri použití BMAR a tri cykly pri priamom adresovaní.

Podobne ako s BLPD, čas vykonania závisí od toho, či je použitý BMAR alebo dlhé priame adresovanie, a je jeden cyklus počas opakovania slučky RPT.

Poznámka

Prístup do oblasti MMR dátovej pamäti je nedostupný inštrukcii BLDD cez priamy alebo BMAR režimy, aj keď MMR môžu byť prístupné cez priame alebo nepriame adresné režimy.

5.5 LAB 5: Zdokonalené programovacie laboratórium

Účelom tohto cvičenia je praktikovať a overovať vylepšené programovacie procedúry. V tomto procese budeme modifikovať súbor .ASM s predošlého laboratória v troch fázach, každá navrhnutá tak, aby poskytla rovnaký výsledok ale s väčším efektom.

5

Doporučuje sa ukončiť každú fázu ešte skôr než prejdeme na ďalšiu, aby sa mohol čo najlepšie pozorovať efekt každej modifikácie a (čo je ešte dôležitejšie) zjednodušiť proces debbugingu riešením len jedného problému v danom čase. Na koniec, skopírujte LAB4.ASM do LAB5.ASM a spravte nasledujúce úpravy:

1. Nahradíte inicializačnú rutinu, ktorá používa LACC/SACL s procesom používajúc RPT/BLPD

2. Po overení správnej operácie blokovej inicializácie nahradíte sumu súčinov, ktorá používa LTA a MPY s implementáciou na báze MAC. Pretože operácia MAC vyžaduje jedno pole na to aby bola v programovej pamäti, môžete modifikovať vašu inicializačnú rutinu aby sa preskočil presun jedného z polí a tým zredukovať množstvo dát RAM a cyklov vyžadovaných na inicializáciu.

3. Po ukončení horeuvedených procesov, uvažujme nasledujúci scenár. Ak je programová ROM v pomalej vonkajšej pamäti, výkon MAC-u bude znížený, pretože bude mať prístup v pomalej pamäti. Funkcia MAC môže byť ešte stále v takýchto systémoch použitá bez straty výkonu ak sa najprv spraví niekoľko doplňujúcich krokov:

   a. Pridelte RAM pre pole koeficientu použijúc *.usect.*
   b. Nasmerujte túto RAM na nahratie do premiestniteľnej vnútornej RAM (napr. Block0).
   c. Nasmerujte túto RAM na spustenie na svojej adrese v programovej pamäti.
   d. Skopírujte pole ROM do RAM použijúc ďalší RPT/BLPD.
   e. Nakonfigurujte RAM do programového priestoru.
   f. Vykonajte MAC používajúc koeficienty, ktoré sú už umiestnené vo vnútornej pamäti.

Tento krok je skôr otestovaním zručnosti COFF než zdokonalených programových operácii, je dostatočné skontrolovať už vyvinutý kód a ak sa vyžaduje, tak ho aj odkrokovať. Na preskúšanie najdete na disku súbory LAB5C.ASM a LAB5C.CMD. Nájdite inštrukcie, ktoré vykonávajú operácie uvedené hore a uvažujte ako súbory .ASM a .CMD pracujú ako team na vykonaní týchto zložitých pamäť používajúcich operácií.

5.6 Prehľad

Obrázok 5-8. Prehľad novo uvedených inštrukcií

Novo uvedené inštrukcie:

MAC	MADS	MPYU	
MPYA	MPZS	SQRA	SQRS
BD	CALLD	XC	
RPT	RPTZ	RPTB	
BLPD	BLDD	BLDP	

Ktoré podmienky môžu byť testované?
Ako veľa môžeme špecifikovať v jednej inštrukcii?
Čo určuje pravdivosť viac-podmienkovej inštrukcie?

Poznámka: A1. Pozrite si výpis z kapitoly 4
A2. Tak veľa ako je potrebné (nie je limit)
A3. Všetky podmienky musia byť pravdivé
(logický AND všekých podmienok)

Obrázok 5-9. Prehľad variant riadenia cyklu

$$y = \sum_{n=1}^{100} x_n a_n$$

	BANZ		RPTB		RPT
	LAR	AR1, #x	LAR	AR1, #x	LAR AR1,#x
	LAR	AR2, #a	LAR	AR2, #a	LARP AR1
	LAR	**AR0, #99**	**LACC**	**#99**	
	LARP	AR1	**SAMM**	**BRCR**	**RPTZ #100**
loop:	APAC		ZAP		MAC a, *+
	LT	*+, AR2	**RPTB**	**next-1**	SACL y
	MPY	*+, AR0	APAC		
	BANZ	**loop, *- , AR1**	LT	*+, AR2	
	APAC		MPY	*+, AR1	
	SACL	y	next: APAC		
			SACL	y	
cyklov	9+7N		12+3N		8+N
pgm.ctl	**4N+1**		**5**		**2**

Kapitola 6

Číslicová aritmetika

Odsek 6-1. Učebné ciele

Dôkladným preštudovaním tejto kapitoly získate vedomosti, ktoré vám pomôžu pri riešení aritmetických otázok spojených s binárnym systémom s pevnou čiarkou. Budete schopní využívať techniky kódovania tak, aby ste ich negatívne vplyvy na `C5x minimalizovali.

Ciele tejto kapitoly sú:

- Konvertovať dvojkové a dvojkové doplnkové čísla na dekadické.
  Vysvetliť rozdiel medzi dvojkovými a dvojkovými doplnkovými číslami.
  Vybrať vhodný režim operácií pre ´C5x.
- Opísať všetky problémy spojené s násobením celých čísel.
  Zistiť riešenia týchto problémov.
- Konvertovať dvojkové zlomky na dekadické.
  Zistiť otázky týkajúce sa násobenia dvojkových zlomkov.
  Ukázať, ako môžeme použiť tzv."Q-zápis" (Q-notation) pri sledovaní binárnej čiarky.
  Ukázať metódy ukladania výsledkov násobenia zlomkov.
- Zistiť otázky týkajúce sa sčítavania dvojkových zlomkov.
  Opísať možnosti vhodné pre zachytenie pretečenia.
  Vybrať vhodný režim pretečenia pre daný systém.
  Porovnať klady a zápory, ktoré vyplývajú z používania celých čísel oproti zlomkovým.
- Opísať metódu, ktorá umožňuje prechod zlomkov cez prekladač (assembler).
- Opísať ako sa môže uskutočniť v ´C5x delenie.
  Ukázať hlavnú inštrukciu delenia.
  Opísať ako pracuje kód (zápis) pre postup delenia.
  Zhodnotiť vykonanie delenia v ´C5x v porovnaní s inými možnými realizáciami.
- Napísať jednoduchý kód na demonštráciu schopnosti inicializovať ´C5x k vykonaniu rozsahu aritmetických operácií so spoľahlivými výsledkami.

6.1 Základy číselnej sústavy

Na získanie schopností vykonávať sčítania a násobenia na ´C5x, je dôležité porozumieť nižšie uvedenej číslicovej aritmeticke, s ktorou sa budeme zaoberať. Preto chceme vysvetliť číslicové pojmy, ktoré súvisia s ´C5x a mnohými procesormi.

6.1.1 Dvojkové čísla

Dvojková číslicová sústava je najjednoduchším číslicovým systémom, ktorý sa využíva v počítačoch a je základom pre iné systémi. Niekoľko detailov s ňou súvisiacich :

- Používajú sa v nej iba dve hodnoty : 1 a 0
- Každá dvojková číslica, všeobecne nazývaná bit, je jedno miesto v dvojkovom čísle a reprezentuje narastajúcu mocninu dvojky.
- Významovo najnižší bit (LSB-least significiant bit) je úplne napravo a má váhu 1.
- Hodnoty sú reprezentované príslušnými jednotkami v dvojkovom čísle.
- Počet použitých bitov určuje aké najväčšie číslo môže byť reprezentované.

Príklady :

6

$0110_2 = (0*8) + (1*4) + (1*2) + (0*1) = 6_{10}$

$11110_2 = (1*16) + (1*8) + (1*4) + (1*2) + (0*1) = 30_{10}$

6.1.2 Dvojkové doplnky

Povšimnite si, že dvojkové čísla môžu reprezentovať len **kladné** čísla. Často je potrebné, aby sme dokázali reprezentovať aj záporné čísla. Číselný systém dvojkových doplnkov modifikuje dvojkový systém, aby zahŕňal aj záporné čísla tak, že významovo najvyšší bit (MSB-most significiant bit) urobí **záporným**.Teda dvojkové doplnky :

- Sledujú operácie jednoduchého dvojkového systému s uvážením toho, že MSB je záporný - okrem jeho veľkosti.
- Môžu mať ľubovolný počet bitov - viac bitov umožňuje reprezentovať väčšie číslo.

Príklady :

$0110_2 = (0*8) + (1*4) + (1*2) + (0*1) = 6_{10}$

$11110_2 = (1* -16) + (1*8) + (1*4) + (1*2) + (0*1) = -2_{10}$

Pre oba príklady (dvojkové doplnoky a jednoduché dvojkové čísla) boli použité rovnaké dvojkové hodnoty. Povšimnite si, že dekadická hodnota je rovnaká ak MSB je 0, ale úplne odlišná ak MSB je 1.
Pre prácu s dvojkovými doplnkami sú užitočné dve operácie :

- Schopnosť vytvoriť súčtový opak, alebo doplnok (komlement) hodnoty.

- Schopnosť uložiť malé čísla do väčších registrov (pomocou znamienkového rozšírenia).

Vytvorenie dvojkového doplnku :

1. Invertujte každý z bitov, t.j. nahradte všetky **1** nulami a všetky **0** jednotkami.
2. Pričítajte 1.

Príklady :

Pôvodné číslo		0110_2	$= 6_{10}$		11110_2	$= -16+8+4+2 = -2_{10}$
1.Invertujte		1001			00001	
2.Pripočítajte 1		1010	= -8+2 = -6		00010	= 2

Nahranie malých dvojkových doplnkov do väčších registrov :

MSB pôvodného čísla musíme preniesť do MSB čísla vo väčšom registri.

1. Nahrajte malé číslo zarovnané sprava do väčšieho regisra.
2. Zkopírujte znamienkový bit (MSB) pôvodného čísla do všetkých nevyplnených bitov v registri.(znamienkové rozšírenie)

Popremýšľajte nad predchádzajúcimi dvoma hodnotami , zkopírovanými do 8-bitového registra :

Príklady :

Pôvodné číslo		0110_2	$= 6_{10}$		11110_2	$= -2_{10}$
1.Nahrajte menšie		0110			11110	
2.Rozšírte o znamienko		00000110	= 4+2 = 6		11111110	= -128+64+...+2 = -2

6.1.3 Režim znamienkového rozšírenia

´C5x môže pracovať s bezznamienkovými alebo doplnkovými operandmi. Bit "režimu znamienkového rozšírenia" (SXM), reprezentovaný v rámci stavového registra ´C5x ,určuje či pri vkladaní hodnoty do akumulátora je operand uvažovaný so znamienkom alebo bez znamienka. Pre prácu s SXM bitom (nastavenie ,vymazanie) môžu byť použité uvažovaný so znamienkom podmienkové inštrukcie :

```
SETC    SXM     ;nastaví SXM bit -akumulátor pracuje v režime
                ;dvojkových doplnkov (vyvolá znamienkové rozšírenie)
CLRC    SXM     ;vymaže SXM bit -akumulátor pracuje v bezznamienkovom
                ;binárnom režime
```

Resetom sa SXM nastaví automaticky. Avšak je dobrým programátorskym zvykom, vždy na začiatku nastaviť požadovaný SXM k zabezpečeniu správneho režimu.Ako bolo ukázané, nesprávne nastavenie môže priniesť značne rozlišné výsledky! Iba v prípade funkčnosti celého systému môže programátor dozrieť a odstrániť možné prebytočné stavové podmienkové nastavovania.

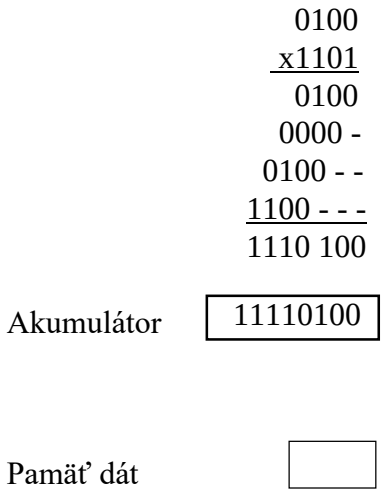
6.2 Dvojkové násobenie

Teraz, keď rozumiete dvojkovým doplnkom, pouvažujte nad procesom násobenia dvoch doplnkových hodnôt. Použitím "zdĺhavého" dekadického násobenia, môžeme uskutočniť aj dvojkové násobenie a nakoniec spočítať všetky výsledky dohromady, aby sme získali výsledný produkt.

Pamätajte si
Toto nie je metóda, ktorú používa ´C5x na násobenie čísel - je to len spôsob nahliadnutia ako aritmetické operácie spracovávajú dvojkové čísla.

´C5x používa 16 - bitové operandy a 32 - bitový akumulátor. Pre objasnenie ,pouvažujte nad nižšie uvedeným príkladom, na ktorom môžeme vyšetrovať použitie 4 - bitových operandov a 8 - bitového akumulátora :

Obrázok 6-2. 4-bitové násobenie



V tomto príklade zvážte nasledovné :

- Aké sú vstupné hodnoty a aký bude očakávaný výsledok?
- Prečo sú "čiastkové výsledky" posunuté počas výpočtov doľava?
- Prečo je konečný čiastočný výsledok "odlišný" ako ostatné?
- Aký výsledok získame, keď sčítame čiastkové výsledky?
- Ako môžeme výsledok uložiť do akumulátora?
- Ako môžeme zaplniť zvyšujúci bit? Je jeho hodnota stále nevyhnutná?

- Ako sa dá výsledok uložiť späť do pamäte? Aké problémy z toho vyplývajú?

Na všetky vyššie uvedené otázky okrem poslednej nájdete odpoveď v tejto kapitole. Na poslednú otázku je niekoľko odpovedí :

- Uložte spodný akumulátor do pamäte použitím inštrukcie SACL. Aký problém je v tomto prípade zrejmý, použitím tejto metódy?
- Uložte horný akumulátor do pamäte použitím inštrukcie SACH. Nevytvorí to nepresnosť a problémy ako neskôr interpretovať výsledky?
- Uložte **oba** aj horný aj dolný akumulátor použitím SACH a SACL. Toto vyrieši vyššie uvedené problémy, ale vytvára niekoľko nových :

  - je použitý extra priestor na kódovanie, pamäťový priestor a čas cyklu
  - Ako sa dá použiť výsledok ako vstup pre nasledujúci výpočet? Je pravdepodobná nejaká podmienka (uvažovať nejaký "spätnoväzobný" systém)?

Z tejto analýzy je jasné, že celé čísla sa pri násobení dobre nesprávajú. Môže sa iný typ čísla chovať lepšie?

6.3 Dvojkové zlomky

Nastolením problémov spojených s celými číslami a násobením, pouvažujte nad možnosťami použitia **zlomkových** hodnôt. Zlomky pri násobení nenarastajú, preto zostávajú reprezentovateľné bez udania veľkosti slova a riešia problém. Nastolené sú výhody zlomkového násobenia. Pouvažujte nad otázkami, ktoré súvisia s používaním zlomkov :

- Ako reprezentujeme zlomky ako dvojkový doplnok?
- Aké otázky zahŕňa násobenie dvoch zlomkov?

6.3.1 Reprezentácia zlomkov v dvojkovom systéme

Mimo reprezentácie kladných a záporných hodnôt, môže byť použitý proces dvojkového doplnku. Predsa len, v prípade zlomkov, nemôžeme nastaviť LSB na 1 (ako to bolo v prípade celých čísel). Uvaažujúc o tom, že rozsah zlomkov je od -1 do +1, a že MSB je jediný bit vyjadrujúci zápornú informáciu, vyzerá to, že MSB musí byť "negatívna pozícia". Kým dvojková reprezentácia je založená na mocnine dvoch, podobne bit bude "polovičná" pozícia každý ďalší bit bude mať opäť polovičnú hodnotu. Uvažujúc, tak ako pred tým 4-bitový model, v nasledujúcom príklade je ukázaná reprezentácia zlomkov.

Príklad : Dvojkové zlomky

1	0	1	1

= -1 + 1/4 + 1/8 = -5/8

6.3.2 Násobenie dvojkových zlomkov

Keď ´C5x uskutočňuje násobenie, proces je rovnaký pre všetky operandy, celé čísla aj zlomky. Preto užívateľ musí určiť ako sa výsledok bude interpretovať. Tak ako predtým, pouvažujme nad príkladom 4-bitového násobenia :

```
   0100
  x1101
  0100
  0000 -
 0100 - -
1100 - - -
1110 10 0
```

akumulátor | 1 1110 10 0 |

pamäť dát | |

Pouvažujte nad nasledovným :

- Aké sú vstupné hodnoty a očakávaný výsledok?
- Tak ako pred tým, "čiastkové výsledky" sú posúvané doľava a výsledok je záporný.
- Ako sa výsledok (získaný sčítaním čiastkových výsledkov) číta?
- Ako sa dá tento výsledok uložiť do akumulátora?
- Ako môžeme vyplniť zostávajúci bit? Je stále táto hodnota nevyhnutná?
- Ako môže byť výsledok uložený späť do pamäte? Aký problém je s tým spojený?

Aby sa dal výsledok násobenia zlomkov "prečítať", je potrebné umiestniť binárnu čiarku (the base 2 equivalent of 10 decimal point). Začnime s určením miesta binárnej čiarky pre vstupné hodnoty. MSB je celé číslo a ďalší bit je 1/2, preto binárna čiarka bude medzi nimi. V našom prípade preto budeme mať 3 bity napravo od binárnej čiarky vo všetkých vstupných hodnotách. Pre ľahšie popísanie možeme to označiť ako čísla "Q3", kde Q označuje počet miest napravo od čiarky.

Pri násobení čísel sa hodnoty Q **sčítavajú**. Takto môžeme (z hlavy) umiestniť binárnu čiarku nad šiesty LSB. A teraz môžeme bez váhania vypočítať výsledok Q6.

Tak ako pri celých číslach, výsledky sú uložené dolu a MSB je znamienkovým rozšírením siedmeho bitu. Ak bola táto hodnota uložená do akumulátora, môžeme výsledok uložiť späť do pamäte rôznymi spsobmi :

- Uložte oba - horný aj dolný akumulátor späť do pamäte. Toto ponúka maximálnu presnosť, ale prináša rovnaké problémy ako pri celočíselnom násobku.
- Uložte iba horný (alebo dolný) akumulátor späť do pamäte. Toto vytvára potenciál pre pamäť zaplnenú rôznymi Q - typmi.
- Uložte horný akumulátor posunutý o jedna doľava. Takto uložíte hodnoty späť do pamäte v rovnakom Q - formáte ako vstupné hodnoty a s rovnakou presnosťou ako vstupné. Ako sa dá uskutočniť posun doľava?

6.3.3 Operácie ukladania a posunutia

Ako sme mohli vidieť v predchádzajucej časti, násobenie dvoch zlomkových operandov (Q15) v TMS 320 znamená prebytočný znamienkový bit v hornom akumulátore. Mimo toho, že sa spomenie najväčšia možná presnosť a možnosť mať výsledky v rovnakom formáte ako zdrojové operandy, je potrebné uložiť späť do pamäte horný akumulátor "posunutý o jedna". Toto sa dá uskutočniť v dvoch cykloch s nasledujúcimi inštrukciami :

```
; A * B = C

LT      A       ; A a B sú zlomky typu Q15
MPY     B       ; P = A*B          : typu Q30
PAC             ; Acc = A*B typu Q30
SFL             ; Acc = A*B typu Q31
SACH    C       ; C = A*B v type Q30 (dolný 16 "Q" sa stratil)
```

Rovnaký výsledok môžeme dostať do C o jeden cyklus skôr, ak použijeme na výstupné posunutia akumulátora možnosti posúvania, ktoré ponúka inštrukcia SACH:

```
; A * B = C

LT      A       ; A a B sú Q15
MPY     B       ; P = A*B          : Q30
PAC             ; Acc = A*B        : Q30
SACH    C,1     ; C = A*B          : Q15 (dolný 16 "Q" sa stratil)
```


Ešte existuje iná metóda pre úpravu prebytočného znamienkového bitu čísla Q30 ;použitím režimu posuvu produktu (Product Mode Shifter).

6.3.3.1 Výsledky posuvov

Režim posuvu produktu, vytvorený ako P - scaler, je umiestnený medzi P registrom a ALU vstupným multiplexorom. Produkt môže prejsť cez posuvník bez posuvu, alebo môže sa uskutočniť posuv o jeden alebo štyri bity doľava, alebo posuv doprava o šesť bitov. Posuvník je riadený obsahom PM poľa stavového registra 1 (ST 1), ktorý môže byť zmenený inštrukciou SPM (nastav režim posuvu produktu). Hodnoty PM a výsledný posuv ukazuje tabuľka 6-1.

Tabuľka 6-1. Režim posuvu produktu

PM	POSUV
0	Bez posuvu
1	Posuv doľava o 1
2	Posuv doľava o 4
3	Posuv doprava o 6

Inicializovaním posuvníka pre posuv o jedna, všetky ukladania z registra produktov budú v tvare Q31. Nasleduje program pre túto operáciu :

```
 SPM 1          ; režim posuvu produktu = 1
; A * B = C
 LT    A        ; A a B sú v tvare :Q15
 MPY   B        ; P = A*B          :Q30
 PAC            ; Acc = A*B :Q31
 SACH  C        ; C = A*B          :Q15 (dolný 16 "Q" sa stratil)
```

PM posuvník je nastavený pri resete na 0. Doporučuje sa, používať PM posuvník na úpravu Q - tvaru. V tomto ohľade, horný akumulátor obsahuje vždy správne Q15 číslo, ktoré sa ľahšie vyhodnocuje počas ladenia programu (debugging-u) a vytvorí sa vyjadrenie, ktoré ľahšie vytvorí sumu nenásobených a násobených čísel. Ako príklad sú uvedené dve časti programu, ktoré tvoria vyjadrenie rovnicu priamky: Y = m * X + b. Prvá časť je v tvare Q30 a druhá v Q31. Všimnite si spôsob, akým je hodnota b pripočítaná do akumulátora.

```
; Q30 príklad
SPM     0       ; režim Q30
LT      X       ;
MPY     m       ; P = m*X         : Q30
PAC             ; Acc = m*X       : Q30
ADD     b,15    ; Acc = m*X+b     : Q30
SACH    Y,1     ; Y = m*X+b       : Q15

; Q31 príklad
SPM     1       ; režim Q31
LT      X       ;
MPY     m       ; P = m*X         : Q30
PAC             ; Acc = m*X       : Q31
ADD     b,16    ; Acc = m*X+b     : Q31
SACH    Y       ; Y = m*X+b       : Q15
```


Existujú už len dve možnosti posunutia pomocou Režimu posunutia produktu : posuv doľava o 4 bity a posuv doprava o 6 bitov.

Posuv o 4 bity doľava (SPM 2) poskytuje akumulátor tvaru Q31 ak je bezprostredne (13-bitová konštanta) použitý MPY (Q12).

Posledný PM posuv hodnoty (SPM 3) poskytuje znamienkovo rozšírený pravý šesť bitový posuv. Toto je spôsob, ako získavať priestor pre sumácie, ktoré môžu (ľahko) prekročiť rozsah zlomkových čísel. V nasledujúcej časti sa uvažuje nad pojmami súčtové pretečenie - zahŕňajúce aj spôsoby jeho zachytenia.

6.3.4 Zlomková reprezentácia v porovnaní s celočíselnou

Naposledy pouvažujte nad porovnaním celých a zlomkových čísel :

- Rozsah
  - Celé čísla majú maximálny rozsah určený počtom použitých bitov.
  - Maximálny rozsah zlomkov je +/- 1.
- Presnosť
  - Maximálna presnosť celých čísel je 1.
  - Presnosť zlomkov je daná počtom použitých bitov.

Teda, akumulátor ´C5x, 32-bitový register, pridáva extra rozsah pre celočíselné výpočty, ale prináša problémy so spätným ukladaním výsledkov do 16-bitovej pamäte.

Naopak, keď používame zlomky, extra bity akumulátora zvyšujú presnosť, ktorá pomáha minimalizovať rastúcu nepresnosť. Pretože každé číslo je presné (nanajvýš) na +/- 1/2 LSB, sčítanie dvoch takýchto hodnôt prinesie v najhoršom prípade nepresnosť 1 LSB. Štyri sčítania vytvoria nepresnosť 2 LSB bitov. Po 256-tich sčítaniach je nepresných 8-bitov. Hoci akumulátornesie 32-bitovú informáciu a zlomkové výsledky sa ukladajú z **horného** akumulátora, extra rozsah akumulátora je hlavnou výhodou pre zníženie nepresnosti pri dlhých výpočtoch sčítania.

6.4 Zachytenie pretečenia

V predchádzajúcich častiach bolo ukázané, že zlomky sú lepšie než celé čísla v tom, že zlomky poskytujú ohraničené výsledky pri násobení, zatiaľ čo celé čísla sú neohraničené - s výsledkami, ktoré rastú a sú ťažšie reprezentovateľné a použiteľné.
Druhá úvaha je pre sčítanie. Kým skúsenosti s násobením zlomkov sú bez "debát", neposkytujú také garancie pri sčítavaní. Hoci veľa DSP algoritmov využíva štruktúry sčítania produktov, je pre užívateľa dôležité, aby vedel uskutočniť sčítavania s o spoľahlivými výsledkami.
´C5x ponúka niekoľko mechanizmov, ktoré pomáhajú rozriešiť túto požiadavku a sú popísané nižšie.

6.4.1 Vytvorenie priestoru (headroom) cez režim posuvu produktu (SPM 3)

Ako bolo spomenuté v predchádzajúcej časti, PM posuvník je schopný uskutočniť posuv doprava o 6-bitov. Keď jeden, vrátane už predstaveného bitu znamienka a "znamienkového rozšírenia" v registre produktov, celkom osem celočíselných bitov môžu byť prezentované v akumulátore ak použijeme variantu SPM 3. Teda môže byť reprezentovaná hodnota väčšia ako +/- 128, ktorá je mimo rozsah "zlomkového" (ohraničeného +/- 1) číselného systému.

6.4.2 Režim možnosti pretečenia

V predchádzajúcej časti sme videli že, je možné vytvoriť ôsme celočíselné bity akumulátora, ktoré umožňujú reprezentáciu veľkých čísel. Avšak, je stále možnosť pre generovanie ešte väčších sčítaní, ktoré však môžu spôsobiť pretečenie akumulátora a poskytovať nesprávne výsledky. Programátori často (právom) uvažujú "v najhoršom prípade" o pričítaní jednotky k najväčšiemu možnému kladnému číslu. Pre náš 4 - bitový príklad, porozmýšľajme o výsledku :

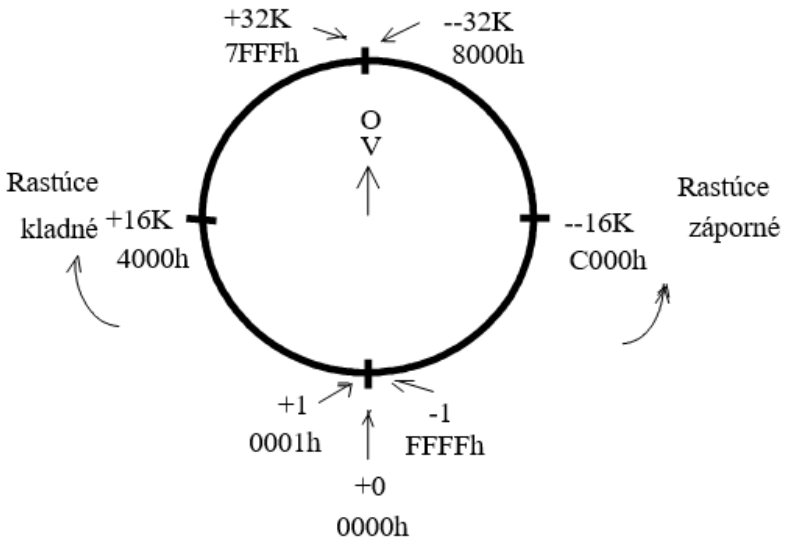
$0111_2 + 0001_2 = 1000_2$ čo znamená $7_{10} + 1_{10} = -8_{10}$

Samozrejme, toto nie je požadovaný výsledok, ale realita pre dve doplnkové čísla. V prípade ´C5x, je počet bitov väčší (16-bitov), ale efekt pretečenia je stále rovnaký.
Je zjavné, že číselná os, ktorá sa používa na zobrazenie rozsahu možných hodnôt, je v našom prípade "číselný kruh", pretože najväčšie kladné a záporné hodnoty sú susedné, nepostavené jedno na druhé. ´C5x môže byť naprogramované ošetriť tento proces dvoma spôsobmi :

- Dovolením možnosti pretečenia pomocou inštrukcie CLRC OVM.
- Zavedením "saturácie" alebo "limitovania", ktoré obmedzia činnosť akumulátora, aby nikdy nedošlo k prechodu cez "hranicu" medzi najväčšou kladnou a zápornou hodnotou použitím inštrukcie SETC OVM.

Uvážte tieto možnosti pri vykonávaní výpočtov zaznamenaných v nižšie uvedenom diagrame 16-bitového celočíselného "číselného kruhu", ktoré môžu byť použité na reprezentáciu rozsahu čísel pre ´C5x:

Obrázok 6-4 . Číselný kruh



- Aký vplyv má stav OVM bitu (režim pretečenia) na výsledky získané výpočtami?
  Ktorý režim je lepší? Za akých podmienok?

Po resete sú nakonfigurované bity pre režim procesora, ale nie režim pretečenia. Ak je potrebné, je dôležité špecifikovať OVM pred začatím výpočtov, aby sme ošetrili pretečenie procesora. Často je dobré pri programovaní špecifikovať celý stav procesora počas každej časti. Takto je možné predísť možnosti spustenia programu so zlou konfiguráciou - čo je oceniteľné hlavne počas zdokonalovania programu. Ak program správne pracuje, môžete nájsť veľa takýchto nepotrebných inicializácií a ak je potrebné, môžete ich z programu vylúčiť, aby sa zlepšila rýchlosť a výkonnosť

6.4.3 Testovanie pretečenia

Pretečenie je pojem, ktorý môže opisovať rozličné situácie. V prípade ´C5x, bit pretečenia (OV) v stavovom registri je nastavený vždy, ak hodnota akumulátora prekračuje hranicu medzi najväčšími kladnými a zápornými číslami. Pretečenie preto indikuje objavenie sa 33. - bitu. Ďalšie podrobnosti o bite pretečenia :

- Pretečenie je blokované - ak sa raz nastaví, zostáva nastavené až do testovania alebo resetu procesora.
- Pretečenie je najlepšie testovať cez podmienkové vetvenie, napr. BCND <pma>, OV

- Aby sme mali správne výsledky, je nutné nulovať OV pred začatím nového výpočtu. V nasledujúcom príklade je použitá jednoduchá metóda využívajúca BCND :

```
. . . starý výpočet . . .
          .
          .
                    BCND NEWCODE, OV
NEWCODE   ZAC
          .
          .
. . . nový výpočet . . .
```

Či bolo alebo nebolo použité podmienkové vetvenie, nasledovne príde riadok NEWCODE a hodnota môže byť testovaná, a tak nulovanie začne pred novým výpočtom.

- Počas hardwarového resetu sa OV bit nastaví na nulu.

6.4.4 Štandardná konfigurácia stavu procesora

Väčšina systémov može byť špecifikovaná tak, aby boli ohraničené a lineárne. Takto môžeme vytvoriť proces, ktorý neprekročí dostupný rozsah čísel. Ak si stanovíme túto požiadavku a chceme dosiahnuť najlepšie výsledky, môžu byť použité nasledovné inicializačné inštrukcie procesora:

```
SETC    SXM         ;povolenie dvojkových doplnkov
CLRC    OVM         ;povolenie medziľahlého pretečenia
SPM     1           ;Q15*Q15 vytvorí Q31 vakumulátore
```

6.4.5 Druhá konfigurácia stavu procesora

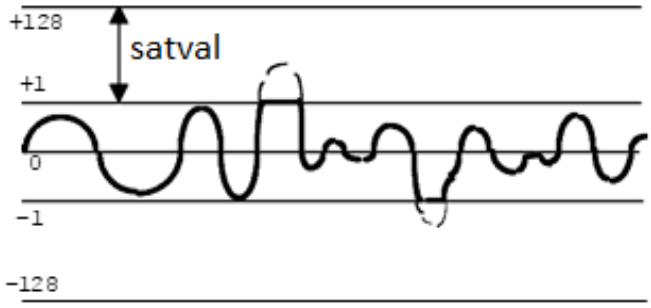
V systémoch, ktoré nemajú podmienky, aby boli ohraničené a (alebo) lineárne, vyššie uvedené štandardné režimy procesora (kap.6.4.4), nemusia dosiahnuť očakávané výsledky. Napríklad, ak je potrebné, aby systém bol schopný "orezania", alebo saturovania, tak ako operačný zosilňovač, systém už nie je lineárny a jeho konfigurácia sa musí adekvátne zmeniť. Na rozdiel od analógových obvodov, predsa len ak digitálny systém orezáva, výstupné hodnoty môžu byť úplne nepresné, tak ako to bolo zobrazené na "číselnom kruhu" v príklade. Predsa len je možné vytvoriť základný systém pre ´C5x, ktorý ľahko a spoľahlivo modeluje proces obmedzovania analógového systému. Ten poskytne pre ´C5x dostatočný "priestor" pre modelovanie lineárneho systému počas výpočtov, ale výstupnú "saturáciu" hodnôt ak prekročia určitý limit. Pouvažujte nad číselnou osou na nasledujúcom obrázku.

Obrázok 6-5. Model saturácie

Použitím režimu posunutia výsledku (SPM 3), môžeme zlomkové hodnoty vyčísliť doprava na celkovo sedem bitov a povoliť 8 bitov pre rozsah celých čísel. Ak si vyberieme lineárny výstupný rozsah +/-1 (zlomkový), priestor - vzdialenosť od saturačných limitov po maximum akumulátora - je celkom podstatný (128 krát väčší ako lineárny rozsah). Teda máme značný priestor pre modelovanie signálov vo veľkom rozsahu bez pretečení v ´C5x.

My však môžeme "orezať" signál pred tým ako ho pošleme na výstupné zariadenie obmedzením výstupu saturačnou hodnotou vždy, keď správna hodnota dosiahne saturačný limit. Tento proces môžeme uskutočniť celkom účinne použitím zápisu na nasledujúcom obrázku.

Obrázok 6-6. Zápis pre saturáciu



- požiadavky na inicializáciu

```
.bss satval
LDP     #satval
SPLK    7E00, satval

SETC SXM
SETC OVM
SPM 3
```

rutina pre saturáciu:

```
ADDH   satval
SUBH   satval
SUBH   satval
ADDH   satval
```

Táto rutina využíva výhody bitovej algebry a fakt, že procesor používa ochranu pred pretečením.

Pričítaním hodnoty 7E00 0000 (rozdiel madzi maximom akumulátora a limitmi zlomkových čísel ak sme zvolili SPM 3) do akumulátora, môžeme vidieť, že hodnoty väčšie ako zlomkový limit (kladný saturačný bod) budú stlačené na 8000 0000.

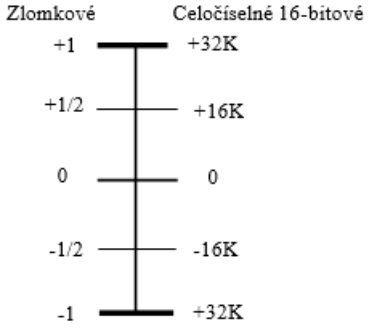
Druhá inštrukcia odčíta 7E00 0000 od akumulátora. Pre hodnoty, ktoré **neprekročili** kladný saturačný limit, sa takto efekt prvej inštrukcie ruší. Ale pre hodnoty, ktoré kladný saturačný limit **prekročili**, akumulátoru sa priradí hodnota kladného saturačného limitu.

Ďalšie dve inštrukcie vykonajú rovnaký postup na testovanie záporného limitu.

Teda hodnoty, ktoré sa nachádzajú medzi saturačnými limitmi, zostávajú nezmenené a hodnoty, ktoré prekročili limity sú nahradené (+/-1) saturačnými hodnotami - efektívna emulácia orezávania, ktorá môže prejsť do výstupného zariadenia - bez straty informácie v ´C5x.

6.5 Binárne zlomky a Assembler

Hoci nástroje COFF **pripúšťajú** celočíselné hodnoty, hexa, binárne i iné formy, **rozumejú** len celým alebo nezlomkovým hodnotám. Pre použitie zlomkov v ´C5x, je potrebné popísať ich tak, ako by boli celými číslami. Na to je veľmi jednoduchý trik. Pouvažujte nad nasledujúcou číselnou osou :



Prenásobením zlomku 32K (32768), vytvoríme normalizovaný zlomok, ktorý môže prejsť cez nástroje COFF ako celé číslo. Pre ´C5x, normalizovaný zlomok vyzerá a aj sa chová presne tak ako zlomok. Teda, ak používame zlomkové konštanty v programe pre ´C5x, program najprv prenásobí zlomok 32768-mimi a použije celočíselný výsledok (zaokrúhlený na najbližšiu celú hodnotu) na reprezentáciu zlomku.

Nasleduje jednoduchá, efektívna metóda prevedenia zlomkov cez assembler :

1. Vyjadrite zlomok ako dekadické číslo (zrušte desatinnú čiarku).
2. Prenásobte 32768-mimi.
3. Predelte vhodným násobkom desiatich na obnovenie desatinného postavenia.

Príklady :

- Reprezentácia 0,62 : 62 * 32768 / 100
- Reprezentácia 0,1405 : 1405 * 32768 / 10000

Táto metóda vytvára správne hodnoty s presnosťou na 16 bitov. Matematiku nemusíte robiť sami a zmena hodnôt vo vašom preloženom súbore (assembly file) prebehne viacmenej jednoducho.

Existuje ešte jedna metóda pre normalizáciu zlomkov. Pre-procesorové nástroje zahrnuté v software prekladača automaticky normalizujú zlomky ak udáme presný Q - formát.

6.6 Delenie v ´C5x

´C5x nemá explicitnú inštrukciu pre delenie. Namiesto toho, je proces rozdelený na série odčítaní a posunutí. Použitím inštrukcie SUBC je možné uskutočniť efetívny a pružný postup delenia. Inštrukcia SUBC zavádza do jedného kroku dlhý postup delenia.

6.6.1 Inštrukcia SUBC

Proces dlhého delenia na štandardných mikroprocesoroch je náročný na čas a vyžaduje značnú manipuláciu pri vykonávaní každého kroku tohto postupu. Inštrukcia SUBC, efektívne "jedno-bitové-delenie", robí tento postup v `C5x účinnejším.

V každom kroku rutiny delenia, musí byť vykonaný test, ktorý zistí, či sa menovateľ "blíži" k čitatelu. Po každom odčítaní, sa vykoná test výsledku na kladné znamienko. Ak je kladný, menovateľ sa "blížil" k čitatelu. Po vykonaní delenia, sa do dolného akumulátora uloží jedna (po posunutí akumulátora doľava o jedna) na zistenie kladného testu. Ak sú testy akumulátora záporné, menovateľ sa "neblížil" k čitatelu. V tomto prípade, je obnovená predošlá hodnota akumulátora s posuvom do ľava a nula sa uloží do LSB na identifikáciu neúspešného testu. Posunutie akumulátora dovoľuje, aby bol testovaný nový bit delenia.

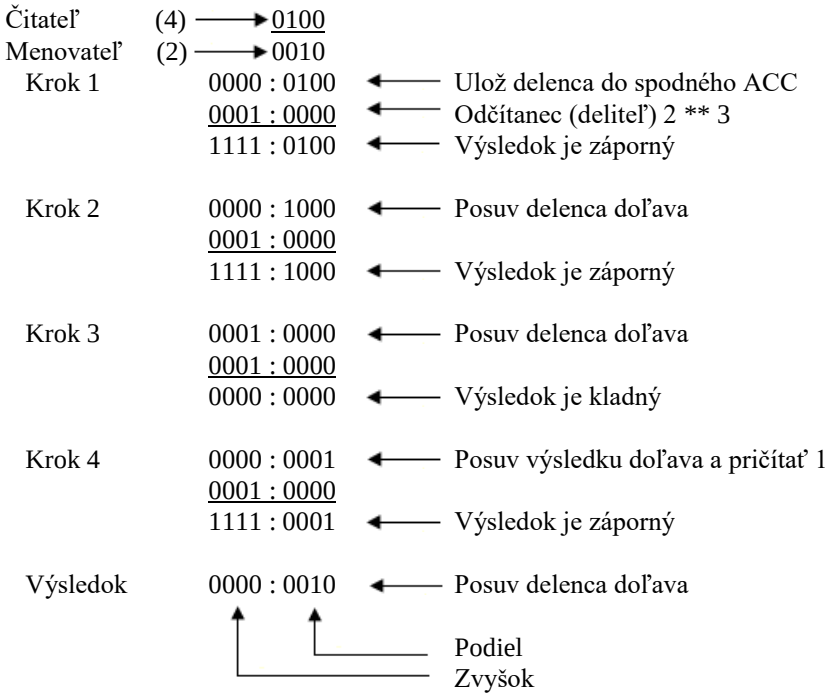
Po N takýchto podmienkových odčítaniach, N-bitový výsledok delenia dvoch operandov je v dolnom akumulátore a zvyšok je uložený v hornom akumulátore.

Hoci nie je rovnako rýchla ako jedno-cyklová inštrukcia delenia, rutina založená na inštrukcii SUBC je o hodnotu rádu rýchlejšia ako vykonanie delenia bez možnosti "1-bitového delenia". Pretože delenie je zriedka používaný proces v DSP, kompromis medzi cenou kremíka a výkonnosťou, je v mnohých prípadoch, celkom primeraný.

6.6.2 Celočíselné delenie

Nasledujúci obrázok ukazuje model celočíselného delenia s použitím 4-bitových operandov. Prechod na 16 - bitov je zrejmý. Uvedomte si, že proces SUBC je zopakovaný 4 - krát v systéme 4 - bitov.

Obrázok 6-7. Celočíselné delenie

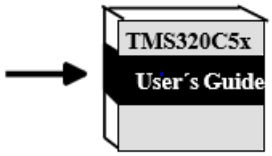


Ukážka programu pre zavedenie rutiny delenia je v nasledujúcom príslušnom materiále.

Pozor
Program v Užívatelskej príručke TMS320C5x je zlý. Použitie LACL musí byť nahradené LACC, aby sa zachovala správna informácia o znamienku.

Celočíselné delenie použitím SUBC

Príklad 7-16, strana 7-29



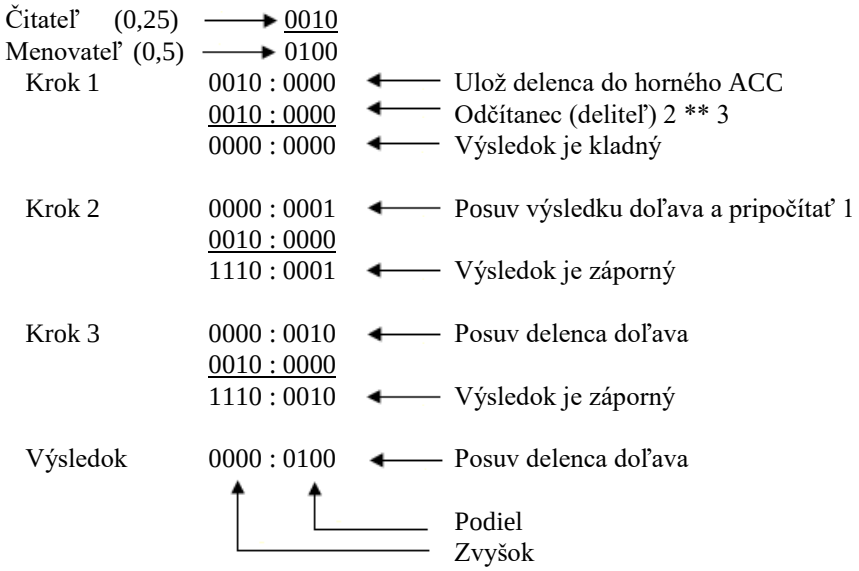
Zapamätajte si zložky, ktoré vykonávajú rutinu delenia. V strede programu delenia je opakovacia slučka a podmienkové odčítanie. Hoci SUBC pracuje len s bezznamienkovými hodnotami, deleniu predchádza rutina, ktorá určí znamienko výsledku a odstráni znamienko vstupným hodnotám pred začatím delenia. Akonáhle delenie končí, k bezznamienkovému výsledku sa pripojí správne znamienko a postup sa tým končí.

6.6.3 Zlomkové delenie

Zlomkové delenie je rovna ké ako celočíselné v dvoch výnimkách :

- Čitateľ je uložený do horného, nie dolného, akumulátora.
- Pre N-bitový zlomok je potrebných len N-1 opakovaní. Podiel je v dolnom akumulátore a zvyšok v hornom, tak ako je to ukázané na nasledujúcom obrázku.

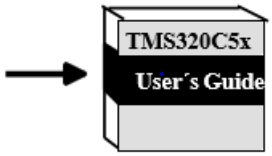
Obrázok 6-8. Zlomkové delenie



Príklad programu pre zavedenie postupu delenia je uvedený v nasledujúcom príslušnom materiále.

Zlomkové delenie použitím SUBC

Príklad 7-17, strana 7-30



6.7 Lab 6: Cvičenie

Cieľom tohto cvičenia je ukázať vašu schopnosť napísať program, ktorý je dôsledný v princípoch teórie čísel, ktoré sme uviedli v tejto kapitole.

1. Vymedzte priestor v RAM pre premenné A,B,C,D a E.
2. Vytvorte nasledujúcu tabuľku dát v ROM pre inicializáciu hodnôt A-E :

A	+0,9
B	+0,8
C	+0,7
D	+0,6
E	-1,0
F	+0,4
X	+0,0

3. Napíšte program, ktorý inicializuje RAM z ROM.
4. Vyriešte rovnicu : X = (A * B) + (C * D) + (E * F) za nasledovných stavových podmienok procesora :

PM	SXM	OVM
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Možte sa rozhodnúť pre ktorúkoľvek možnosť zmeny stavových podmienok procesora:

- Použitím vyvinutého nástroja na zmenu stavových bitov dajte prednosť spätnej slučke a novému vykonaniu rovnice. (najľahšie)
- Zmenou vášho programu pred každým novým testom (náročnejšie).
- Napísanie programu, ktorý prejde v slučke všetky dané kombinácie.

Rada : Pouvažujte nad použitím režimu "Q", a nad tým, ako s ním pracovať.

5. Popíšte účinok stavovej konfigurácie.

Ktoré konfigurácie vytvoria správne hodnoty?
Prečo ostatné zlyhali?
Ktorá kombinácia(-cie) je pre náš prípad "najlepšia"?

ČASŤ 7

ZÁKLADY DSP

Odsek 7.1 učebné ciele

V tejto časti sa naučíte ako sa robia filtre na 320C5x.

Špecifické časti tohto modulu sú:

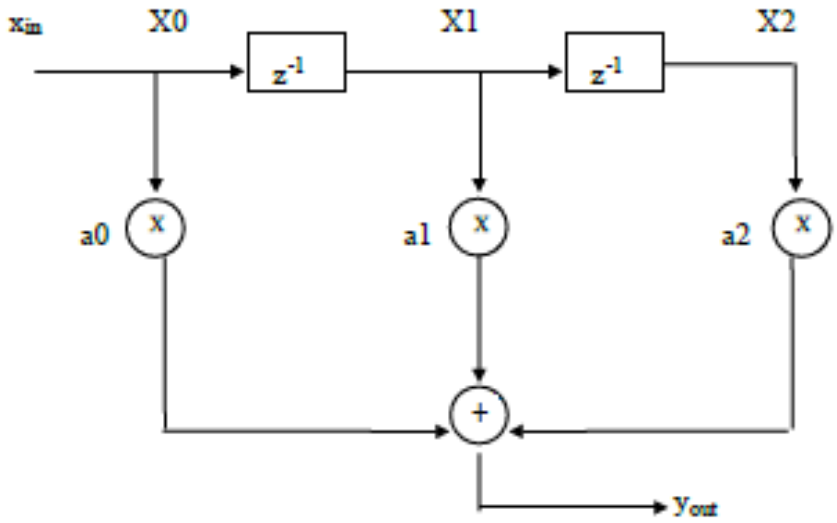
- Opísať potrebu oneskorovacích liniek v digitálnych filtračných systémoch.
- Realizácia oneskorovacích liniek na 320 dvoma spôsobmi.
- Opísať činnosť FIR a IIR fitrov. Vybrať optimálny režim činnosti 320 FIR a IIR filtrov pri daných podmienkach.
- Identifikovať zdroje šumov v diginálnych filtroch. Napísať kód, ktorý minimalizuje šumové efekty v diginálnych filtroch.
- Čítať signálové schémy. Prepísať signálové schémy do kódu 320-ky.

7.1 FIR filtre

V tejto časti budeme skúmať FIR (Finite impulse response) filter a uvažovať o jeho princípe a využití. Ukážeme vlastné metódy použitia FIR filtra. Táto časť sa skončí ukážkou, ako sa FIR filter odvodzuje, ako sa kóduje a aplikuje na vstupný signál.

obr.7.2 Jednoduchá schéma FIR filtra

y(n)=a0*x(n)+a1*x(n-1)+a2*x(n-2)



7.1.1 Operácie FIR filtra

Na hore uvedenej schéme vstup X(0) vychádza z miesta X0. V tom istom čase je X(0) nasobený a0 a privedený na výstup Yout. O jednu časovú jednotku neskôr má vstup Xin novú hodnotu X(1). Hodnota X(0) sa presunie do oneskorovacej linky a je teraz v X1, t.j. robí miesto pre X(1) v X0. Teraz sa Yout = X(1)*a0+X(0)*a1. Tento proces sa dopĺňa a opakuje pre každú časovú jednotku a je vyjadrený:

Y0 = X0*a0 + X1*a1 + X2*a2

Aký je vzťah medzi X(1),X1 a X(n-1)?

7.1.2 Definícia FIR filtra

Lineárne posunutý nemenný systém môže byť opísaný pomocou rovnice:

$$y(n)= \sum_{k=1}^{N} ak * y(n-k) + \sum_{k=0}^{M} bk * x(n-k)$$

FIR systém je jediný, ktorého odozva na jednotkový vzorkovací impulz má ohraničené trvanie. Vo FIR systémoch všetky *ak* stavy sú nulové, preto výstup FIR systému môže byť opísaný

$$y(n)= \sum_{k=0}^{M} bk*x(n-k)$$

To môže byť vysvetlené tak, že výstup *y(n)* FIR systému je vážený súčet vstupného toku *x(n)* a predchádzajúcich *M* vstupných vzoriek. A tak odpoveď na jednotkový impulz bude nulová pre *n>M.*

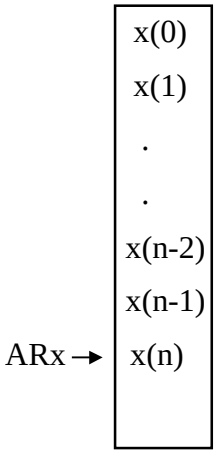
FIR filter je charakterizovaný úplnou stabilitou, a preto môže byť použitý na výrobu lineárnej fázovej odozvy.

7.2 Použitie oneskorovacích liniek

V predchádzajúcej časti ste sa zoznámili so štruktúrou FIR filtrov a ďalším cieľom je rozvíjať účinnosť modelov s 'C5x. FIR je v podstate súčtom výsledkov operácií aplikovaných na pole hodnôt, obsiahnutých v oneskorovacej linke. Ako ste videli, pri použití 'C5x, je matematické sčítavanie produktov jednoduché. Ale ako sa dá oneskorovacia linka použiť?

7.2.1 Lineárny buffer

Najjednoduchšie využitie oneskorovacej linky v mikroprocesore je lineárny buffer, kde sa na N najčerstvejších vzoriek v poli aplikuje N filtrových operácií. Celý čas je FIR činný, nová dátová hodnota je získaná a pridaná na koniec zoznamu dát. V tomto spôsobe môže byť jednoduchý pomocný register dekrementovaný priamo v poli, počas výpočtov FIR, a inkrementovaný do ďalšej voľnej pozície v poli, keď vstúpi nová dátová hodnota, znova počas výpočtov FIR.

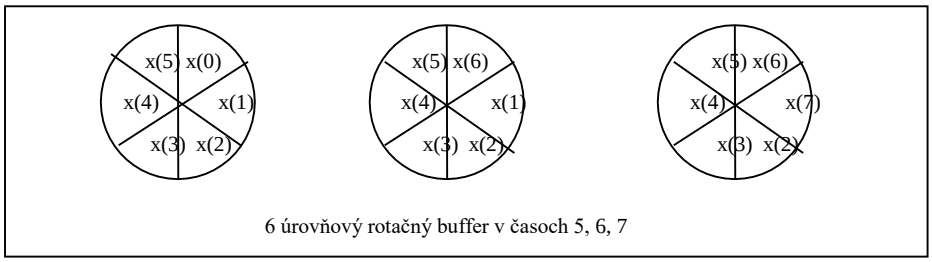


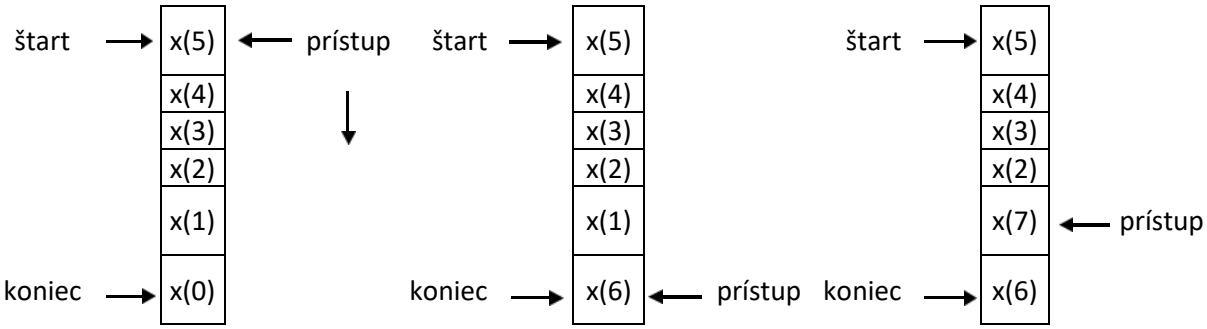
Výhoda tohto prístupu je v efektivite, s ktorou 'C5x môže obsluhovať oneskorovaciu linku. AR smie byť automaticky dekrementovaný aj inkrementovaný počas sčítania produktov výpočtov, pokiaľ všetky schopnosti procesora môžu byť venované vykonávaniu výpočtov skôr, než ovládaniu oneskorovacej linky. Nevýhoda lineárneho buffera je v používaní pamäte. Ak je prijatá ďalšia nová dátová hodnota, je nutné ďalšie obsadenie pamäte. V spojitých systémoch by to vyžadovalo nekonečne veľa pamäte - nepraktické riešenie.

7.2.2 Rotačné buffery

Jedno z možných použití lineárneho pamäťového buffra je rotačný buffer. Ak kruh z N pamäťových segmentov odpovedá N stupňovému filtru, dostatok dát by bol použitelný pre vykonanie sumácie výsledkov operácií. Po každom výpočte by bola pripočítaná nová hodnota a nahradila by starú hodnotu. Takto najnovších N hodnôt je umiestnených v N pozíciách - veľmi efektívne použitie pamäte.

obr.7.3 Šesť-stupňový rotačný buffer



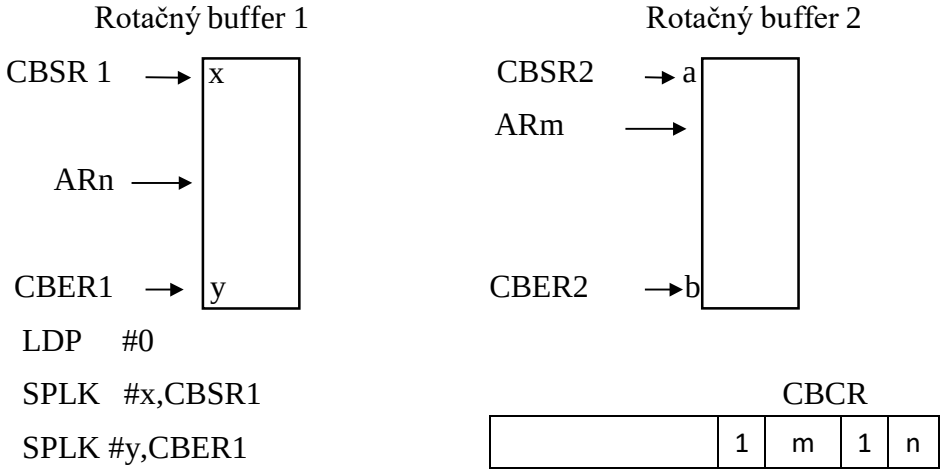


Bohužiaľ, rotačné pamäte neexistujú ako fyzické súčiastky. Dajú sa však modelovať pomocou štandardných pamätí adresovaných v "cirkulačnom" móde so softverovým riadením. Na uskutočňovanie tohto rotačného buffra pomocou softvéru je potrebné riadiť počítadlá pre začiatok a koniec plochy používanej ako rotačný buffer. Tretie počítadlo (smerník) sprístupňuje pamäť vo vnútri tejto plochy. V každom čase je smerník menený a porovnávaný so začiatočným a koncovým počítadlom. Ak je smerník znížený, alebo zvýšený nad koncové počítadlo, je mu priradená hodnota začiatočného počítadla. Efekt rotačného buffra je v presnej realizácii použitej štandardnej pamäte. Výhodou tejto metódy je efektívne využívanie pamäte. Nevýhodou je dodatočný čas použitý na simuláciu cirkulačnej štruktúry. Tento extra proces vyžaduje výsledky v redukovanom vykonaní, alebo je potrebný rýchlejší procesor.

7.2.3 Rotačný buffer - hardver v 'C5x

'C5x zahrňuje hardverovú podporu pre rotačné adresovanie, ktoré neprináša problémy. Hardver umožňuje dva rotačné buffre rôznej veľkosti, ktoré sú spravované rovnako. Registre smerníkov začínajú a končia adresáciu bufferov a definujú, ktorý pomocný register bude spojený s každým rotačným buffrom, ako ukazuje nasledujúci obrázok:

obr.7.4 Inicializácia rotačného buffera




```
LDP    #0
SPLK   #x,CBSR1
SPLK #y,CBER1
LAR    AR3,#x
SPLK  #1011,CBCR
```

Je dôležité si všimnúť, že rotačný buffer pracuje vo veľmi špecifickom riadení. Keď sa vybraný AR rovná CBER, ďalšia modifikácia AR spôsobí, že do AR bude daná hodnota CBSR, t.j. keď AR = CBER, práve prírastok o hodnotu inú než 1, alebo zníženie spôsobí, že AR = CBSR! Ďalej rotačné balenie je vykonané iba vtedy, keď je prispôsobenie medzi vybraným AR a CBER. Ak AR je zvýšený nad koncový register, neprispôsobenie bude detekované a rotačný buffer nebude dobre fungovať. Preto by ste mali byť pozornejší pri používaní kroku iného než 1. Ďalšia možnosť využitia rotačného, alebo modulo adresovania poľa je opísaná v TMS620C5x-User's guide, časť 7.5 strany 7.13-7.14.

7.2.4 Vlnový buffer

'C5x dovoľuje ďalší typ buffra, ktorého schéma sa líši od ukázaných metód. Namiesto rotačného prístupu vlnový buffer vkladá hodnotu oneskorovacej linky do radu. Po každom novom vyrátanom výstupe je údaj posunutý z pozície do pozície vo vlnovom móde skoro rovnakým spôsobom, ako naznačuje signálová schéma. Použitím štandardného procesora by čas potrebný na pohyb dát v tomto móde bol nereálny, tj. čas potrebný na prečítanie a zápis operácie pre každý okruh filtra. S 'C5x je jedno prečítanie a zápis operácie kombinované s DMOV inštrukciou

DMOV <dma>

Hodnota v pozícii <dma> je prepísaná do pozície <dma>+1. Táto inštrukcia nepoužíva ani akumulátor ani pozíciu <dma>. Proces môže byť ešte viac efektívny, pretože inštrukcia DMOV môže byť spojená s funkciou LTA (načítaj prechodný register a sčítaj s predchádzajúcim akumulátorom) ako LTD

LTD <dma>

Funkcia DMOV je zahrnutá v inštrukciách LTD, MACD a MADD. Inštrukcia DMOV pracuje iba s RAM blokmi 0-2 umiestnenými na čipe, keď sú konfigurované ako pamäť dát. Ak sa pokúsite použiť DMOV v externej pamäti, pozícia bude prečítaná, ale zápis nebude uskutočnený. Nakoniec poznámka, že ak uskutočňujeme operácie vo vlnovom bufferi, je nutné postupovať od najstarších hodnôt k najnovším. Prechádzaním od nových k starým by s DMOV nepracovala oneskorovacia linka, ale namiesto toho by kopírovala najnovšiu hodnotu cez celú oneskorovaciu linku.

7.2.5 Porovnanie rotačného a vlnového buffera

Máme dve metódy pre prácu oneskorovacích liniek, ktorá je lepšia? Vlnový buffer sa používa, ak je potrebná štandardná oneskorovacia linka a vnútorné RAM bloky 0-2 sú používané k odkladaniu dát. Rotačný buffer je potrebný pre neštandardné oneskorovacie linky (meniace sa vstupné data počas prepočítavania FIR výstupu), alebo ak sú použité vonkajšie pamäte.

7.3 I/O operácie

'C5x má 2 inštrukcie, IN a OUT, ktoré umožňujú prenos medzi pamäťami dát a I/O zariadeniami. Tieto inštrukcie sú optimále pre použitie s paralelnými A/D a D/A prevodníkmi. Syntax inštrukcií je:

```
IN <dma>,<pa>          ; čítaj data z periférie <pa>
                       ; a zapíš do pamäte dát na
                       ; pozíciau <dma>
OUT <dma>,<pa>         ; zapíš data nájdené v pamäti dát
                       ; na pozícii <dma>
                       ; na port s adresou <pa>
```

kde

<dma> je priama, alebo nepriama adresa pamäte dát

<pa> je adresa portu s hodnotou medzi 0-0FFFFh

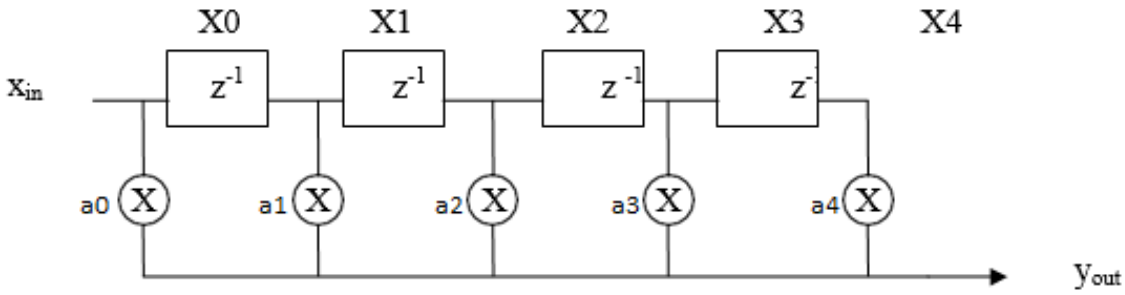
Existujú dve metódy pre prístup vonkajších zariadení v I/O mape. MMR umiestnené na strane 0 pamäte dát obsahujú 16 prístupovsch pozícií I/O portov. Tieto sú umiestnené v <dma> 50h-5Fh a prepájajú s I/O pozíciami 50h-5Fh. Môžu byť tiež prístupné pod názvom PA0-PA15. V systéme s málo I/O zariadeniami, schopnosť sprístupniť I/O zariadenie cez pamäť dát môže zvýšiť pohodlie, alebo urýchliť vykonanie od I/O zariadení. Smú byť prístupné pomocou rôznej normálnej aritmetickej operácie (LACC, ADD, SUB, SACH, SACL atď.) IN alebo OUT.

Poznámka IN a OUT operácie medzi I/O pamäťou a pamäťou dát. Sprístupnenie I/O cez porty v MMR znamená, že dáta z I/O zariadenia sú už v pamäti dát a že ďalšie pozície v pamäti dát nemusia byť použité ako sprostredkovacie medzi I/O zariadením a matematickými registrami.

7.4 Realizácia FIR filtra

Teraz môže byť celé riešenie FIR filtra realizované pomocou kombinácie inštrukcie DMOV a IN/OUT inštrukcií so súčtom produktov.Uvažujme ukázaný blokový diagram FIR filtra 4 -ho rádu.

obr. 7.5 FIR filter



$$y(n) = a0*x(n) + a1*x(n-1) + a2*x(n-2) + a3*x(n-3) + a4*x(n-4)$$

alebo

$$Y0 = a0*X0 + a1*X1 + a2*X2 + a3*X3 + a4*X4$$

Riešenie tohto filtra môže byť realizované niekoľkými rozličnými spôsobmi,ako to je opísané v nasledujúcich častiach.

7.4.1 FIR filter použitím LTD a MPY

Ako ste videli skôr, inštrukcie LTA a MPY môžu byť použité na riešenie sumátora produktov opercií. Pridaním operácie DMOV do inštrukcie LTA cez inštrukciu LTD, vlna dát sa drží prostrdníctvom oneskorovej linky, keď je pole prístupné zdola nahor. Nakoniec obsah IN a OUT inštrukcií umožňuje filtru poslať a prijať data z ADC a DAC. Dole ukázaný kód segmentov je rovnaký, s výnimkou použitia spôsobu adresovania.

obr. 7.6 FIR filter -lineárny buffer

	priame adresovnie		nepriame adresovanie	
	LDP	#X0	LDP	#X0
			MAR	*,AR1
			LOOP LAR	AR2,#A4
			LAR	AR1,#X4
START	ZAP		ZAP	
	LT	X4	LT	*-,AR2
	MPY	A4	MPY	*-,AR1
	LTD	X3	LTD	*-,AR2
	MPY	A3	MPY	*-,AR1
	LTD	X2	LTD	*-,AR2
	MPY	A2	MPY	*-,AR1
	LTD	X1	LTD	*-,AR2
	MPY	A1	MPY	*-,AR1
	LTD	X0	LTD	*,AR2
	MPY	A0	MPY	*,AR1
	APAC		APAC	
	SACH	Y,1	SACH	Y,1
	OUT	Y,PA0	OUT	Y,PA0
	IN	X0,PA1	IN	X0,PA1
	B	START	B	LOOP

7.4.2 FIR filter použitím priamych koeficientov

Hore uvedené kódy používajú koefcienty zapísané v pamäti dát. Inštrukcia MPY môže tiež použiť priamu 13 bitovú hodnotu.Táto redukuje pamäť dát,pretože zlúči 2 polia hodnôt do jedného, s koeficientom poľa "postav do" inštrukie MPY. Ako uvidíte dole, je to dosť podobné priamemu adrsovaniu.

obr.7.7 FIR filter realizovaný s LT a násobený konštantami

```
        LDP     #X0
START   ZAP
        LT      X4
        MPY     #A4
        LTD     X3
        MPY     #A3
        LTD     X2
        MPY     #A2
        LTD     X1
        MPY     #A1
        LTD     X0
        MPY     #A0
        APAC
        SACH    Y,4
        OUT     Y,PA0
        IN      X0,PA1
        B       START
```

7.4.3 FIR filter použitím MACD

Ako bolo skôr ukázané, inštrukcia MAC vykonáva obidva operácie LTA aj MPY. Pre vykonanie MAC, je jedno pole umiestnené v pamäti programu.Vo FIR filtri je potrebné pre inštrukciu DMOV pridať MAC v MACD. Pretože DMOV iba pričíta hodnoty oneskorenia v pamäti dát, je nutné uložiť hodnotu data v pamäti dát, ktoré sú prístupné zdola nahor. Hodnota koeficienta bude preto umiestnená v pamäti programu. Pretože aj koeficienty sú v programovom priestore a môžu byť prístupné iba inkrementovaním, musia byť zapísané od posledného k prvému v poradí, v akom bude hodnota dát prístupná.

obr. 7.8 FIR použitím MACD

```
        .text
        LAR     AR1,#x+99       ;koncový bod oneskorovacej linky
        MAR     *,AR1           ;aktívne arl
        LAR     AR0,#99         ;dĺžka poľa
FIR:    RPTZ    #99             ;100 opakovaní
        MACD    coeff,*-        ;
        APAC                    ;celkový súčet
        SACH    *,1,Y           ;zápis výsledku do Y
        OUT     *+,PA1          ;zápis do DAC, inkrem. X0
        BD      FIR             ;spätná slučka
        IN      *0+,PA2         ;čítanie ADC do X0,presun do X99
        .data
coeff:  .word   a99,a98,...,a0  ;zápis poľa od starých k novým
y       .usect  *d_line*,1      ;výstuná hodnota
x       .usect  *d_line*,100    ;vstup poľa- linka do DARAM
```

7.4.4 FIR filter použitím MADD

Ako bolo poznamenané skôr, MADS umožňuje vykonať funkciu MAC na < pma > špecifikovanú pomocou BMAR lepšie, než pevná hodnota v inštrukcii MAC. MADS + DMOV sú zahrnuté v inštrukcii MADD a ináč je rovnaká ako MACD.

obr. 7.9 FIR filter použitím MADD

```
LACC    #coeff
SAMM    BMAR        ;načítaj adresu A0 do MMR *BMAR* -<pma> pre MADD
        *
        *
MADD    *-          ; mpy, acc, dmov
```

7.4.5 FIR filter riešený úvahami.

FIR filter môže byť na ‘C5x vyriešený niekoľkými rôznymi spôsobmi. Použitý prístup bude závisieť na systémových požiadavkách. Výber umožňuje užívateľovi robiť rozhodnutia medzi vykonaním, využitím pamäte programu a pamäte dát. Keď riešite FIR filter použitím ‘C5x, budete chcieť uvažovať podmienky opísané v nasledujúcich odstavcoch.

Uchovávanie koeficientov

Koeficienty môžu byť umiestnené aj v pamäti dát aj programu. V pamäti programu môžu byť koeficienty prístupné ako priame hodnoty , alebo cez skupinu inštrukcií MAC (násobiť / hromadiť). Najlepšie vykonanie bude dosiahnuté s koficientami v pamäti programu, pretože operandy môžu byť menené súčasne s násobením cez programovú a dátovú zbernicu.

In-line, alebo slučkový kód

Filtre môžu byť riešené aj použití In-line, alebo slučkovým kódom. In-line kód dáva vyšší výkon než slučkový kód, ale všeobecne za cenu zvýšenia užívania pamäte programu. Schopnosť ‘C5x opakovať blok . dovoľuje použiť kompaktný slučkový kód bez softverových pomocných operácií všeobecne združených so slučkovým kódom.

Opakovanie jednej, alebo bloku inštrukcií

‘C5x má hardverovú podporu pre jednoinštrukčné opakovanie (RPT) ako aj pre opakovanie blokov (RPTB). Najvyšší výkon FIR filtrov môže byť dosiahnutý použitím jednoinštrukčných opakovaní, ale to vyžaduje, aby koeficienty boli umiestnené v pamäti programu. Jednoinštrukčné opakovania nemôžu byť prerušené, čo môže byť výhoda aj nevýhoda.

Typy násobiacich operácií

‘C5x poskytuje niekoľko násobiacich operácií. Obidve násobiace argumenty smú byť privedené do násobenia z pamäte dát, alebo jeden môže byť z pamäte programu a jeden z pamäte dát. Pre dlhé filtre použitie skupiny násobiacich a zhromažďujúcich (MAC) inštrukcií, ktoré berú operandy z pamäte dát aj z pamäte programu to prináša vysoký výkon. Pre krátke filtre môže byť uprednostnené použitie skupiny inštrukcií načítaj-T/násob (LT/MPY), pretože tam je menej nastavovacích pomocných inštrukcií.

7.4.6 Nastavenie FIR filtra

V prehľade všeobecných prístupov realizácie FIR filtra sú elementy spoločné s riešením ďalších algoritmov.Program musí nastaviť CPU zdroje, používanú pamäť a používané registre.

CPU

Mali by ste zaručiť, že konfigurácia CPU, tak ako aj výsledok posúvania, existencia znamienka, pretekanie a konfigurácia pamäte sú vhodne nastavené. Pretože tieto konfigurácie sú nastavené resetom, nesmú byť nastavené explicitne, ale musíte starostlivo zaručiť, aby celá konfigurácia bola taká,akú žiadate.

Pamäť

Koeficienty a vzorky oneskorovacej pamäte musia byť nastavené. V závislosti na použitom prístupe, koeficienty môžu byť umiestnené v pamäti dát aj programu. Vzorka oneskorovacej pamäte(vstup dát) môže byť nastavená celá na nulu.

Registre

Registre používané na adresáciu a zápis prechodných výsledkov môžu potrebovať nastavenie. To zahrňuje pomocné registre, smerník AR a môže obsahovať aj akumulátor a P register.

7.5 FIR filter - cvičenie

7.5.1 FIR filter, konzultácie s inštruktorom

DSP konzultácie s inštruktorom ukazujú prácu FIR filtra. Je tu opísaná metóda pre úpravu koeficientov a ponúka vniknutie do matematických aspektov FIR filtrov. Program vedie rozvoj filtra a dovoľuje pozorovať ako filter spracuváva vstupný signál.

Koncepčné informácie môžete nájsť vnútri tohto programu, ktorý máte na vašej domácej diskete. Neskôr môže byť ukázaná zmena, ktorú pochopíte až s rastom vašeho vybavenie v DSP. Začiatok vašich konzultácií s inštruktorom ukazujú tieto body:

1. Vybrať disk C

   C:>DSP5

2. Zavolať program

   BASIC TUTOR

3. Sledovať inštrukcie na displeji

7.5.2 Kód FIR filtra

Napíšte jadro kódu komponentov riešenia 10 stupňového FIR použitím **externej** pamäte.

Procedúry

1. Založiť tabuľku koeficientov v externej ROM: a[10]=(1,2,3...10).
2. Rezervovať miesto v externej RAM pre oneskorovaciu linku.
3. Inicializovať smerník oneskorovacej linky.
4. Zapísať jadro kódu riešenia FIR filtra.

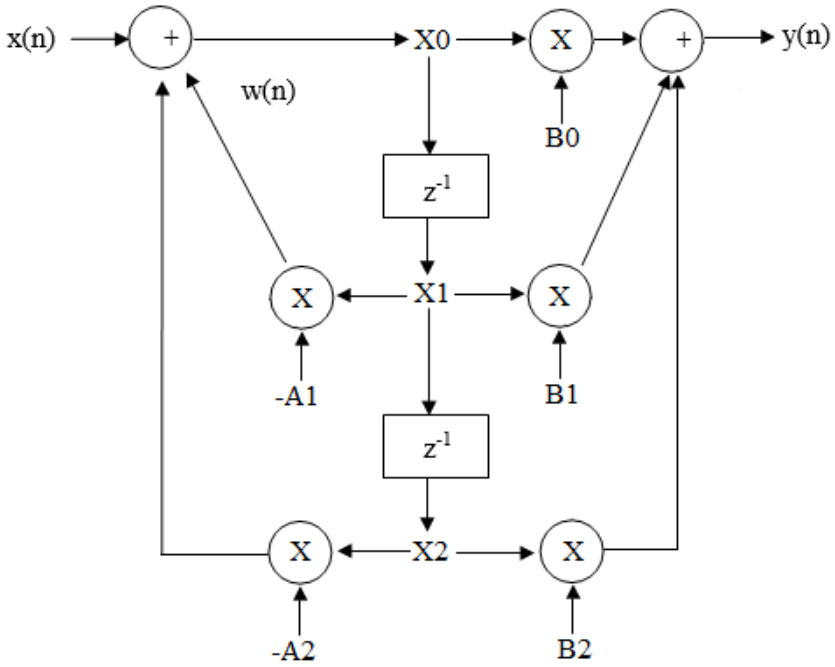
7.6 IIR filtre

V tejto časti preskúmame IIR (Infinite Impulse Response- nekonečná imulzová odozva) filtre.Budeme uvažovať o ich koncepcii a riešení. Ukážeme štandardné metódy riešenia. Táto časť sa uzavrie diskusiou o šumových zdrojoch a o ich minimalizácii.

7.6.1 IIR filter- principiálna schéma

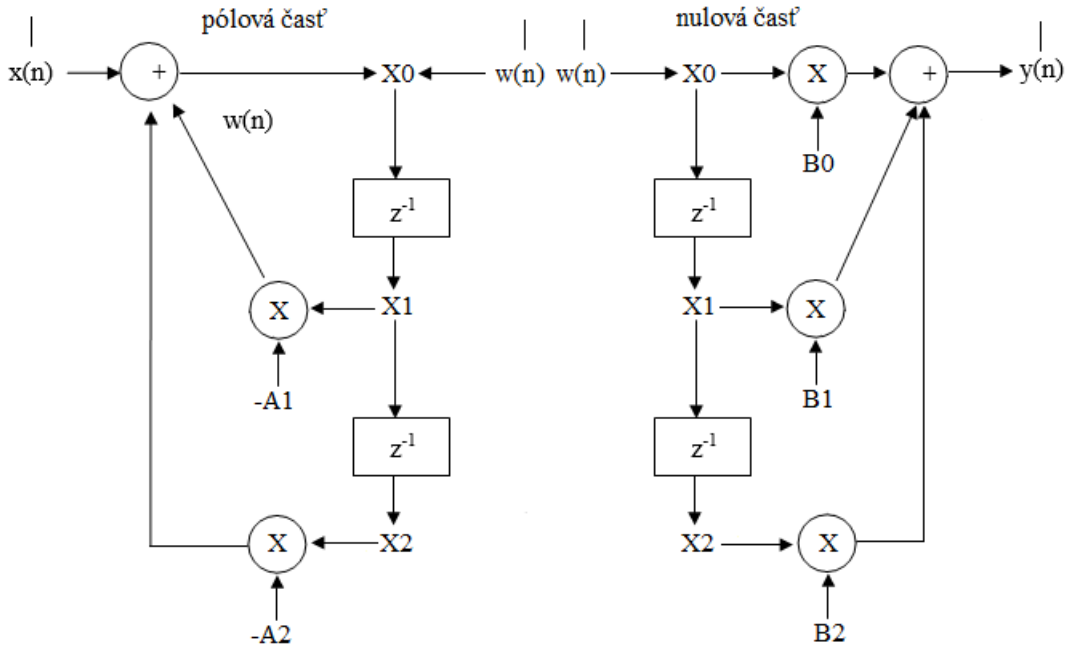
IIR filter je podobný dvom spojeným FIR filtrom. Na prvý pohľad je ťažké v nasledujúcom obrázku IIR filtra ukázať a predstaviť zrejmú metódu softwerového riešenia.

obr. 7.10 IIR filter 2-ho rádu



Na lepšie pochopenie môže byť celá schéma na hornom obrázku ukázaná ako dva jednotlivé bloky: spätnoväzbová (pólová) časť a predná (nulová) časť ako to ukazuje nasledujúci obrázok.

obr. 7.11 Náhradný IIR filter 2-ho rádu



7.6.2 IIR filter- riešenie

Softwérové riešenie IIR filtra je odvodené priamo z náhradnej schémy. Prvá časť označená ako spätnoväzbová, ktorá bude poskytovať nové medzivýsledky, potrebné pre FIR časť. Len čo je tento nový medzivýsledok (X0) určený, FIR časť môže pracovať ako ste to videli skôr v tomto kurze.

Metóda na prevod signálového diagramu na 320-ový kód požaduje pracovať späť z výstupu. Výsledok , X0, je odvodený zo sčítania jedného vstupu a 2 spätnoväzbových signálov. Takto je sčítanie prvým kódovaním. Výsledok je zapamätaný v akumulátore tak, že drží Q30 čísla ako výsledok zlomkového násobenia a súčet vstupnej hodnoty (Q15) musí dať rozdiel (30-15=15) usporiadaných binárnych bodov.

```
        LDP     #X
        SPM     0           ;neposúvanie, ACC=Q30
IIR     IN      X,PA1       ;nový vstup z portu 0, uloženie do X
        LACC    X,15        ;načítaj acc so vstupom ako Q30 (q15+s15)
        LT      X1          ;T=X71
        MPY     A1          ;P=X1*A1
        LTA     X2          ;T=X2,Acc=Xin + X1*A1
        MPY     A2          ;P=X2*A2
        APAC                ;Acc = Xin + X1*A1 + X2*A2 v Q30 formáte
        SACH    X0,1        ;zápis Acc do X0 ako Q15
        LACC    #0          ;štart s Acc=0
        MPY     B2          ;P=X2*B2(posledná hodnota T znova použitá)
        LTD     X1          ;T=X1, Acc =X2*B2 , X1 do X2
        MPY     B1          ;P=X1*B1
        LTD     X0          ;T=X0, Acc= X2*B2 + X1*B1 , X0 do X1
        MPY     B0          ;P=X0*B0
        APAC                ;Acc=X2*B2 + X1*B1 + X0*B0
        SACH    Y,1         ;zapíš Acc do Y ako Q15
        OUT     Y,PA0       ;pošli výstup na port 0
        B       IIR         ;spätná slučka
```

7.6.3 Šumy v IIR filtri

V tejto časti bude preskúmaný dôležitý detail v IIR filtroch. Pozrieme sa na zdroje šumu a ako ich treba minimalizovať. Veľa otázok tu prezentovaných je použiteľných tiež vo FIR filtroch, ale sú málo zaujímavé a neopakovateľné. IIR filter používa výstupy ako časť ďalšieho vstupu, takže chyby môžu byť spájané celý čas.

Sú možné dva typy nezdaru s IIR filtrom: pretečenie hore a dole. Horné pretečenie nastáva keď signál prekročí systémové hranice. Dolné pretečenie môže nastať v chýbaní dostatočne veľkého vstupného signálu.

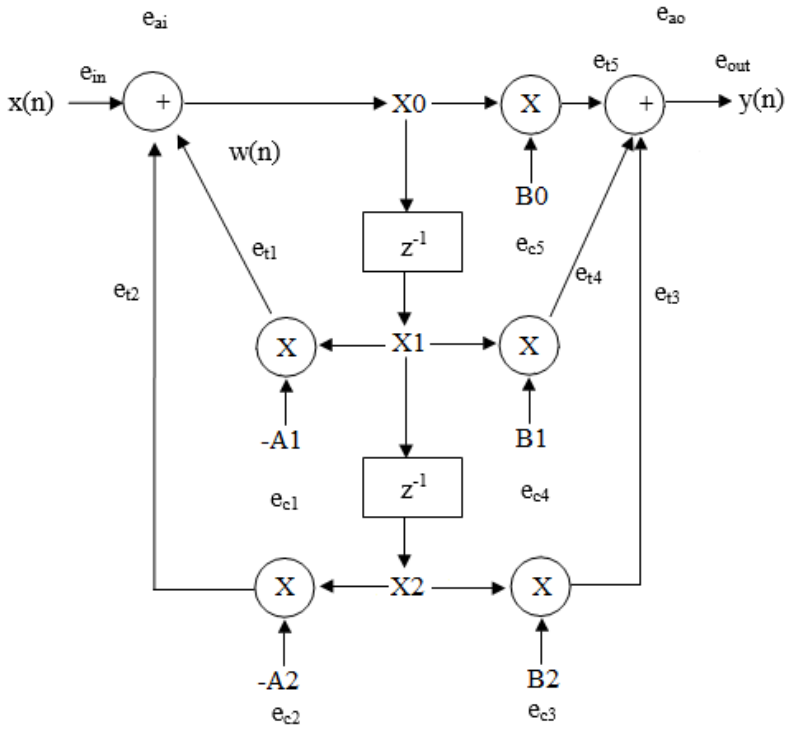
Na zabezpečenie vlastnej činnosti systému, musia byť tieto podmienky znova strážené. V takom prípade, zdroje chýb musia byť nájdené a možnosti katastrofických chýb eliminované. Keď sa riadite týmito vedomosťami, môžete správne navrhnúť systém k požiadavkám s vysokým stupňom dôvery v túto spoľahlivosť.

7.6.3.1. Zdroje chýb

Numerické výpočty prezentujú niekoľko zdrojov chýb vrátane:

1. **I/O kvantovanie signálu** - číslo bitu presnosti umožňuje v A /D a D /A prevodníkoch určiť chyby tak, že sú vlastnosťou prevodu. Dominantný faktor je v A /D prevode, vstupná chyba koluje vo filtri, zatiaľ čo D/A chyba nemá vplyv na samotný filter.
2. **Kvantovanie filtračných koeficientov** - výsledok nepresného koeficienta je reprezentovaný odklonom od želaného frekvenčného rozsahu systému.Vo veľa prípadoch 16 bitové koeficienty postačujú. Takže v prípade vysoko kvalitných filtrov s pólmi blízko jednotkovej kružnice, malá chyba koeficientov by mohla nakoniec posunúť pól mimo kružnice. Táto podmienka znamená nestabilný stav.
3. **Nekorelovaný Round Off (odseknutý) šum** - tento šum je výsledkom systému, ktorého výstupy sú príliš malé na prezentáciu s postačujúcou presnosťou. Číslo bitov umožňuje vo výstupe pokles, náhodná LSB chyba sa stane významnou.
4. **Korelovaný Round-Off (odseknutý) šum** - existencia limitných cyklov a ich najhorší prípad, pretečenie nahor, je vlastnosťou efektu akumulovaných chýb. Tento úkaz je ťažké modelovať. To sa často lepšie správa s použitím rýchlych, presných koeficientov a zabezpečením platnosti použiteľného dynamického rozsahu pre prezentované vstupné a výstupné signály.
5. **Obmedzenie dynamického rozsahu** - 16 bitové operandy používané TMS 320 dovoľujú 96 dB dynamický rozsah. Dynamický rozsah je zväčšený na 192 dB v matemtickom kanáli (násobič a akumulátor) pre medzivýpočty.

obr. 7.12 IIR filter 2-ho rádu, chybové zdroje



Pre najlepšie spojenie chybových javov s IIR filtrom, uvažujme schému na hornom obrázku. Upozorňujeme, že zdroje chýb existujú v každom kroku signálového toku.

e_{in}	A/D kvantizačná chyba
e_{out}	D/A kvantizačná chyba
$e_{t1}...e_{t15}$	chyba neupĺného odseknutého výsledku
e_{ai},e_{a0}	akumulačná chyba
$e_{c1}...e_{c5}$	chyba kvantovania koeficientov

Odhaduje sa, že úplná akumulačná chyba ukázaného komplexu, pokiaľ neviete, že interné zdroje sú, alebo môžu byť vyrobené, je menšia než I/O kvantizačná chyba. Ak starostlivo navrhujete systém, môžete uvažovať chyby iba I/O základne,čím získate jednoduchý model.

Na dosiahnutie optimálneho výsledku z IIR filtra ste požadovali zachovať čo najväčšiu presnosť. Niekoľko programových prístupov je ponúknutých ďalej ako riešenie veľa podobných problémov.

7.6.4 Scalovanie vstupu, rotácia doprava a doľava

Ako si všimnete v šumovom modeli IIR filtra, je žiadúce mať koeficienty veľké a použiteľné v ďalšom udržiavaní ich presnosti. V takejto práci je možné získať systém, ktorého zosilnenie je večšie ako 1.

V predchádzajúcej diskusii sme odvodili, že všetky operandy musia byť menšie ako 1 a reprezentované ako zlomky. Ak vstup X je obmedzený jednotkou a maximálna citlivosť H filtra je 7, čo musí predchádzať maximálnemu výstupu Y? Ten môže byť reprezentovaný rovnicou

$$X * H = Y$$

kde $|X| < 1$ a $Hmax = 7$

preto $Ymax = 7$

Pretože hodnota 7 nemôže byť reprezentovaná ako zlomok, musí sa niečo urobiť s obidvoma *X* alebo *H* , aby *Y* mohlo byť reprezentované. Dve možnosti sú:

$$Y = X * H/7$$

$$Y = H * X/7$$

Na prvý pohľad, sa zdá , že obidva alternatívne prípady zníženia presnosti pomocou redukcie X alebo H sedmičkou znamenajú skoro 3 bity presnosti. Od rastu H je prípad pretečenia nahor, návrhár bude často redukovať zosilnenie filtra. V takom prípade, samozrejme, bude redukovať presnosť všetkých koeficientov.

Lepšia metóda je obmedzenie vstupu. To je zvlášť ľahké, ak vstup je z takého A/D prevodníka, ktorý poskytuje menej než 16 bitov dát. Napríklad, ak je v tomto systéme použitý 12 bitový A/D prevodník, zrejme vstupný rozsah ±1 by znamenal pripojenie A/D výstupov na 12 najvyšších bitov(MSBs) dátovej zbernice TMS 320 (a uzemnením 4 najnižších bitov (LSBs)). Napojenie týchto 12 A/D liniek na 12 najnižších bitov dátovej zbernice (a 4 zvyšné najvyššie bity dátovej zbernice na najvyššií bit A/D - vytvorenie hardwarového znamienkového módu),tj na vstupnom rozsahu sa to prejaví ako delenie 16. Aká veľká presnosť je stratená v tomto procese? Od týchto stálych 12 bitov rozlíšenia sa nestratí presnosť výsledkov a je dosiahnutý reprezentovateľný výstup.

Dokonca ak máme dostupný 16 bitový vstup , vstupný scaling v IIR filtri je možný bez straty presnosti. Upozorňujeme,že v IIR filtri je vstup najprv zapamätaný do pamäte dát a až potom načítaný do akumulátora v Q30 formáte pomocou posunutia o 15. Ak je posunutie redukované na 14, vstup by bol

efektívne delený 2-kou. Delenie 7-kou v hornom príklade je lepšie realizované pomocou 3 bitového posunu (delenie 8-kou). Delenie ďalšou najbližšou vyššou mocninou 2 umožňuje realizovať výstup so stratou menšou ako 1 bit presnosti.

7.6.5 Dvojitý APAC

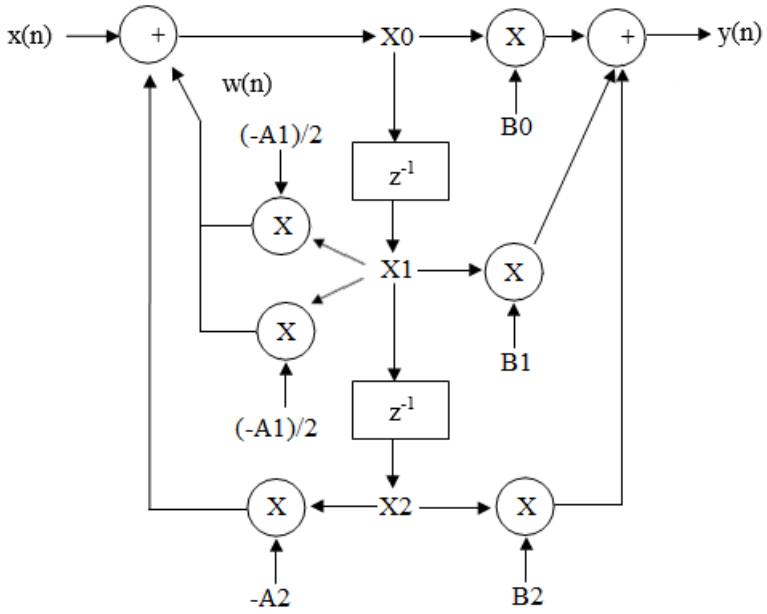
Koeficienty odvodené vo IIR filtri 2-ho rádu často obsahujú jeden, ktorý prekračuje 1. Nasledujúca vzorka nastavenia:

a1 = 1.93	b0 = 0.77
a2 = 0.83	b1 = 0.02
	b2 = 0.34

Koeficient a1 je príliš veľký na to, aby reprezentoval zlomok. Ako potom môže byť tento systém realizovaný na 320? Jedno riešenie je deliť všetky koeficienty 1.93 tak, že najväčšia hodnota bude 1. Tento prístup stojí skoro 1 bit v existujúcich 5 koeficientoch, najmä v požadovanom koeficiente b1. Druhá metóda je deliť iba a1 nasledujúcim väčším celým číslom (v tomto prípade 2). Bez korekcie by to premenilo funkciu filtra. Preto musí byť výsledok *a2*X2* toľkokrát pričítaný do akumulátora, koľkými bolo a1 delené (v tomto prípade opäť 2). Akumulácia je ľahko realizovateľná pomocou opakovaného použitia inštrukcie *APAC* počas tvorby výsledku *a2*X2*. S˘peciálny inštrukčný čas je malá cena za extra presnosť v požadovaných aplikáciach filtra.

Ďalej, *a0* je medzi spätnoväzbovým sčítaním a *X0*. V tomto modeli je *a0* definované ako 1, takže proces násobenia 1 je preskočený a je ušetrený čas. Ak koeficienty scalujete dole, musíte pamätať, že treba scalovať dole aj *a0*.

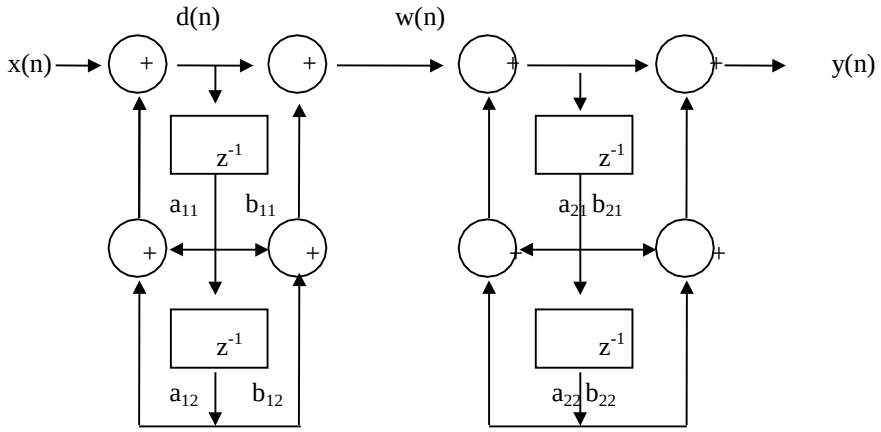
obr. 7.13 IIR s dvojitým APAC



7.6.6 Delenie viacstupňových systémov

Naša diskusia má zvážiť dvojstupňové filtre. Čo ak opísané použitie vyžaduje systém vyššieho rádu? Jasný model pre IIR filter 4-rádu je jedno zoskupenie 4 oneskorovacích liniek a dokopy je predĺžená verzia systému 2-ho rádu. Hoci je to koncepčne v poriadku, táto metóda je nevhodná k aplikácii. Ohraničenia 16 bitového slova sú rýchlo dosiahnuté so systémami vyšších rádov, preto je lepšie rozdeliť systémy vyšších rádov do sérií kaskádových častí 2-ho rádu, ako to ukazuje nasledujúci obrázok. Ďalší úžitok tohto prístupu je, že kód IIR, uvádzaný skôr, sa stane štandartom pre všetky IIR systémy.

obr. 7.14 IIR filter v kaskádovej forme



$$d(n) = a_{11}d(n-1) + a_{12}d(n-2) + x(n)$$

$$w(n) = d(n) + b_{11}d(n-1) + b_{12}d(n-2)$$

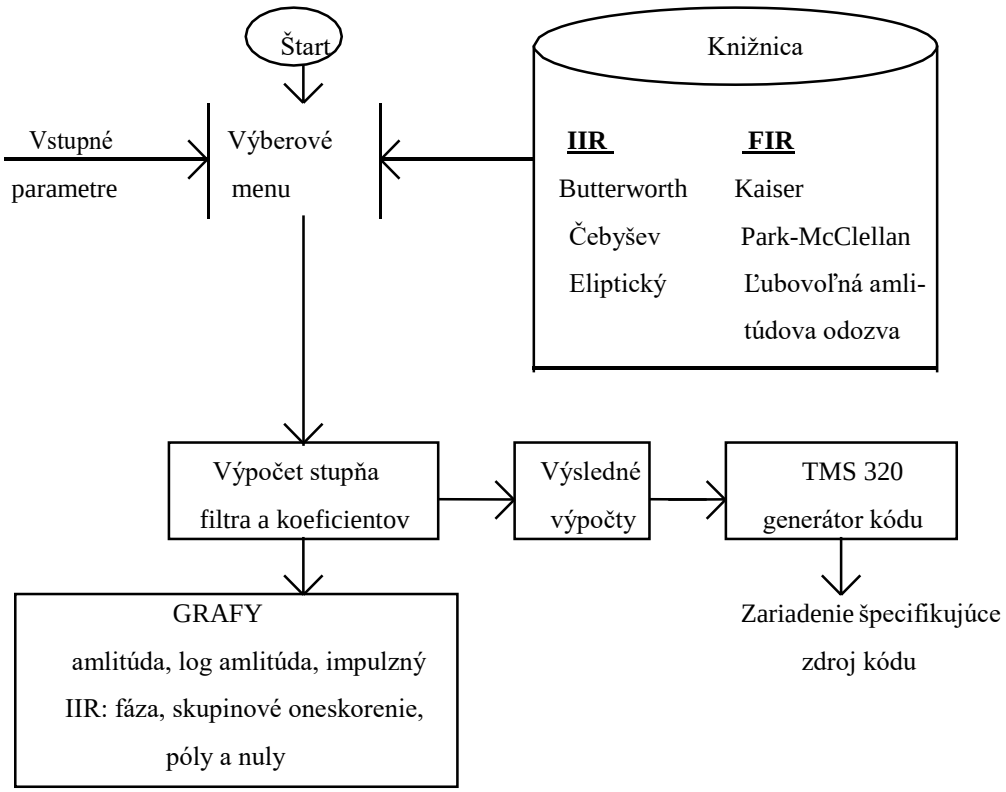
$$H(z) = A \prod_{k=1}^{2} \left[\frac{1 + b_{k1} z^{-1} + b_{k2} z^{-2}}{1 - a_{k1} z^{-1} - a_{k2} z^{-2}} \right]$$

7.6.7 IIR príklad kódovania

Návrh daného filtra bude ukázaný na demonštráciu množstva konceptov využitia IIR filtra.

Prvý krok je definovať požadované vykonávanie. Raz je to urobené, musia byť stanovené ďalšie potrebné filtre a hodnoty ich koeficientov. Tento prípravný krok môže byť vykonaný rôznymi spôsobmi, dané dostatočnými podkladmi v DSP teórii a využití. Pre tých, ktorí nemajú rozsiahle vedomosti o DSP teórii, existujú jednoduché riešenia v softverovom balíku nazývanom Digital Filter Design Package (DFDP) (pozri nasledujúci obrázok).

obr. 7.15 Digital Filter Design Packabe (DFDP)



DFDP vám pomôže pri detailoch systému: vzorkovacia rýchlosť, požadovaný frekvenčný rozsah, typ použitého filtra. Program bude potom určovať hodnoty koeficientov nutných na dosiahnutie daných kritérií.

Sú ponuknuté voľby na pozeranie a kraslenie výkonov filtra v rôznych spôsoboch ako ukazujú obr. 7.18-7.21. Ak ste spokojný s vykonaním, môžete potom žiadať, aby koeficienty boli včlenené do kódu, ktorý bude použitý vo funkcii filtra na ‘C5x. Tvorba kódu je komentovaná a obsahuje rôzne, skôr prebraté scalingové operácie.

obr. 7.16 IIR eliptický dolnopriepustný filter

Nekonečná impulzová odozva (IIR)
Eliptický dolnopripustný filter
16-bitové kvantovanie koeficientov

Stupeň filtra = 4
Vzorkovacia frekvencia = 8.000 kilohertz

I	A(I,1)	A(I,2)	B(I,0)	B(I,1)	B(I,2)
1	-1.276855	.513092	.107407	.016239	.107250
2	-1.322205	.885345	.620819	-.690979	.620758

*** Charakteristiky navrhovaného filtra ***

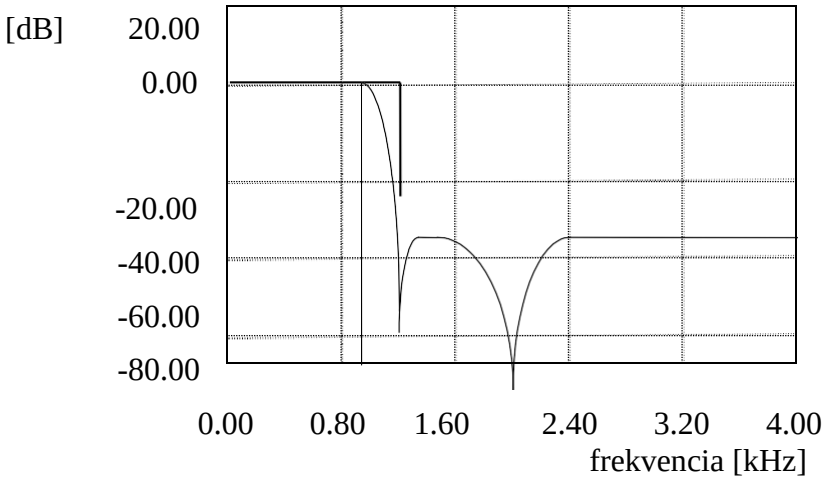
	skupina1	skupina2
dolná hranica skupiny	.00000	1.20000
horná hranica skupiny	1.00000	4.00000
nominálna citlivosť	1.00000	.00000
nominálna vlna	. 05000	.05000
maximálna vlna	.04438	.04287
vlna v dB	.37714	-27.35642

obr 7-17 Kód filtra 2-ho rádu

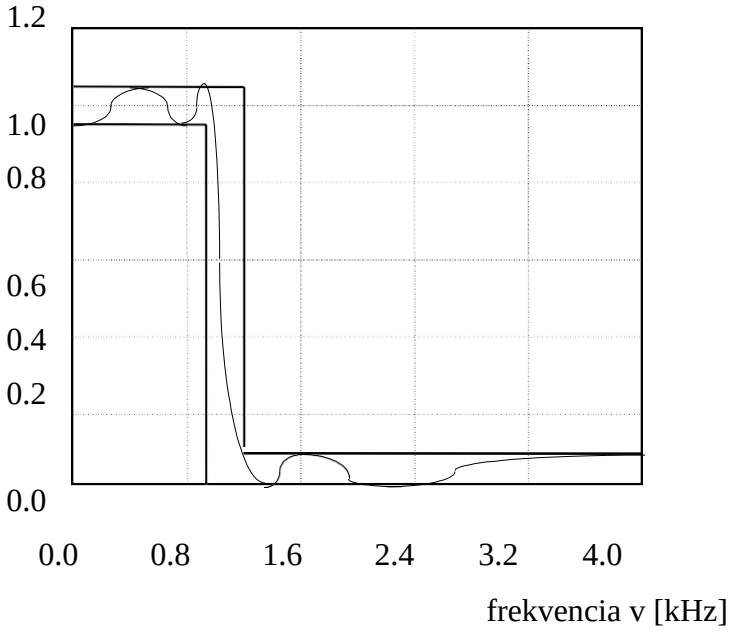
```
FILTER
* 2-STUPŇOVÝ FILTER ĆASŤ 1
        LACC    VSAMPL, 13          *ZOBER A SCALUJ VSTUP
        SACH    FILTT               *ULOŽ SCALOVANÝ VSTUP
        LT      FILTT               *ZOBER SCALOVANÝ VSTUP
        MPY     B010                *P=(B0*VSTUP)/2
        ZALH    Z011
        ADDH    Z011                *AC=Z-1
        APAC
        APAC                        *AC=Z-1+(B0*VSTUP)
        MPY     B011                *P=(B1*VSTUP)/2
        SACH    VSAMPL      *ULOŽ VO VÝSTUPE
        PAC
        LT      VSAMPL      *AC=(B1*VSTUP)/2
        ADDH    Z012                *AC=(Z-2+(B1*VSTUP))/2
        MPY     Z011                *P=(A1*VÝSTUP)/2
        APAC
        APAC                F*AC=(Z-2+(B1*VSTUP)+(A1*VÝSTUP))/2
        MPY     A012                * P=(A2*VÝSTUP)/2
        SACH    Z011                *ULOŽ V Z-1
        PAC                         *AC=A2*VÝSTUP
        LT      FILTT
        MPY     B012                **P=(B2*VSTUP)/2
        APAC                    *AC=((B2*VSTUP)+(A2*VÝSTUP))/2
        SACH    Z012                *ULOŽ V Z-2
* 2-STUPŇOVÝ FILTER ĆASŤ 2
        ZALH    VSAMPL      *ZOBER STAV VSTUPU
        SACH    FILTT               *ULOŽ VSTUP
        LT      FILTT               *ZOBER SCALOVANÝ VSTUP
        MPY     B020
        ZALH    Z021
        ADDH    Z021
        APAC
        APAC                        *AC=Z-1+B0*VSTUP
        MPY     B021                *B1*VSTUP/2
```

```
SACH    VSAMPL          *ULOŽ VO VÝSTUPE
PAC
LT      VSAMPL          *AC=(B1*VSTUP)/2
ADDH    Z022            *AC=(Z-2+(B1*VSTUP))/2
MPY     A021            *P=(A1*VÝSTUP)/2
APAC
APAC                    * AC=(Z-2+(B1*VSTUP)+(A1*VÝSTUP))/2
MPY     A022            *P=A2*VÝSTUP/2
SACH    Z021            *ULOŽ V Z-1
PAC                     *AC=A2*VÝSTUP
LT      FILTT
MPY     B022            *P=B2*VSTUP/2
APAC                    *AC=((B2*VSTUP)+(A2*VÝSTUP))/2
SACH    Z022            *ULOŽ V Z-2
RET                     *RETURN
```

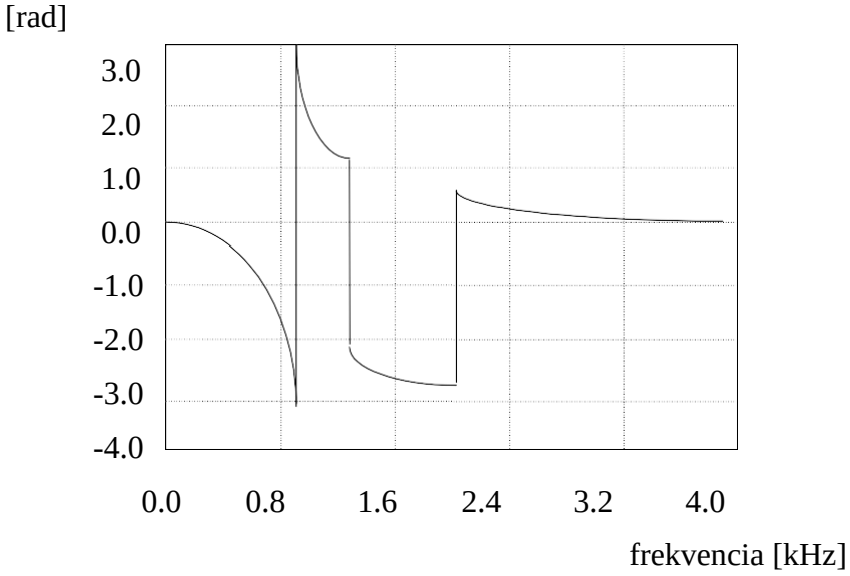
obr. 7.18 Log amlitúdovo frekvenčná charakteristika



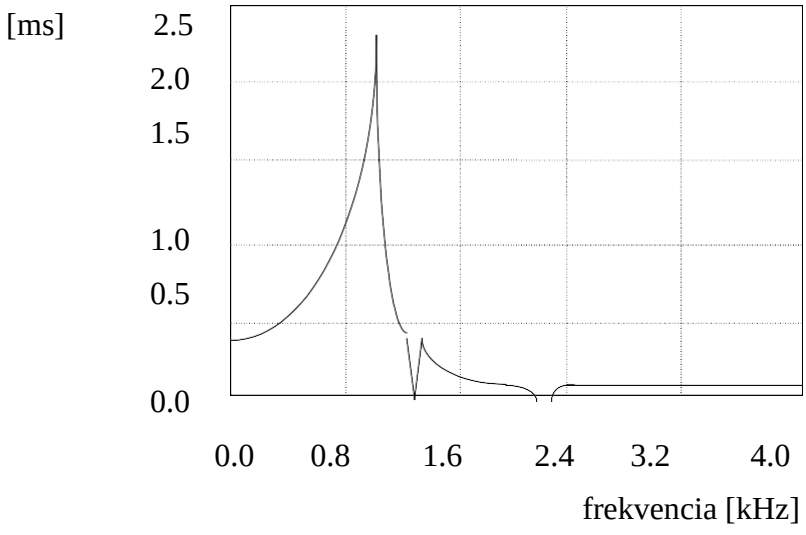
obr. 7.19 amlitúdovo frekvenčná charakteristika



obr.7.20 Fázovo-frekvenčná charakteristika



obr. 7.21 skupinové oneskorenie



7.7 Porovnanie IIR a FIR filtrov

V tejto časti budú porovnané IIR a FIR filtre.

7.7.1 IIR filter

IIR filter sa používa tam, kde amplitúdová odozva je základné vykonávacie kritérium. IIR filter má viac účinkov na citlivosť horných frekvencií než FIR filter pre danú dĺžku filtra a číslo inštrukčných cyklov. Je fakt, že IIR filter je 5-10 krát účinejší než FIR filter v riadení amplitúdovej odozvy.

7.7.2 FIR filter

FIR filter je uprednostený v systémoch, kde fázová odozva je dôležitý parameter. Normálne FIR filter povoľuje priamku, v IIR filtri nie je možné predpovedať fázovú chybu.

Druhá výhoda FIR filtra je stabilita. IIR filter obsahuje spätnú väzbu. Všetky rekurzívne obvody môžu vypadnúť za určitách podmienok, preto požadujú starostlivejšiu analýzu, ak majú byť použité. FIR filter neobsahuje rekurzívne časti, je odolný voči týmto problémom a poskytuje jednoduché a isté riešenia pre programátora.

Konečný faktor v prospech použitia FIR filtra je veľká pravdepodobnosť existencie použitia inštrukcie *MPY* s 13-bitovou konštantnou a spájanie údajov z dátovej RAM s ich inicializáciou.

7.8 Filter -cvičenie

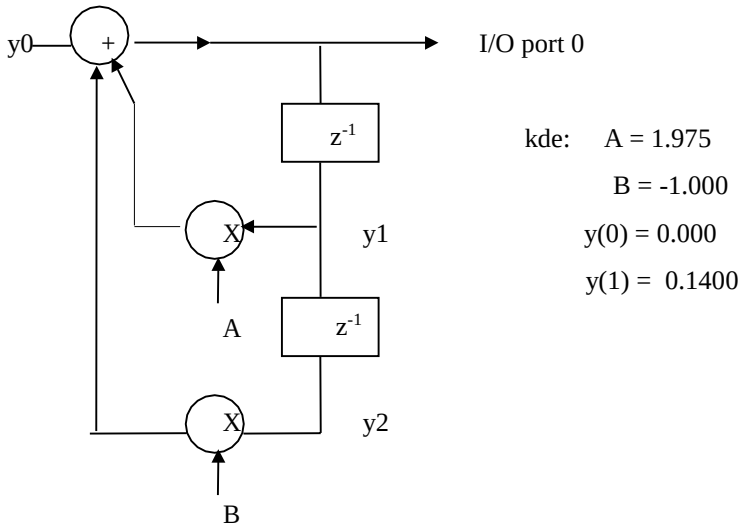
7.8.1 Cieľ

Cieľom tohto cvičenia je aplikovať technickú debatu v časti 7 pre realizáciu signálových schém na ‘C5x. Vytvorte nové súbory obsahujúce kód potrebný na realizáciu signálovej schémy uvedenej nižšie. Pre A,B použite hodnoty a inicializačné podmienky uvedené vpravo.

Skončite váš kód napísaním obnovených hodnôt y0 na I/O port 0. Opatrenia robené v simultátore nahrávajú hodnoty napísané na port 0 do DOS súboru nazývaného *OUT.DAT*. Keď simulujete kód, beží najmenej 40 výstupov, potom ukončite simulátor a prehľad vašich výsledkov je okamžite v *PLOT OUT.DAT* v DOSe. Ukážte vaše výsledky inštruktorovi na kontrolu.

obr. 7.22 Rekurzívny filter

Realizujte následovnú signálovú schému na 320:



7.8.2 Poznámky

1. Ako teraz viete, y0 je spojitá výstupná hodnota , v tomto systéme je to rovné váženemú súčtu 2 minulých hodnôt výstupov y1 + y2. V tomto systéme, získané výstupy z časov 0 a 1 dovoľujú kódu začať s riešením *y* v čase 2.
2. Tu nie je potrebná pozícia v pamäti pre y0, pretože to nie je vstupná hodnota vo výpočtoch.
3. Smiete si zvoliť realizáciu tejto rutiny založenej na tejto osamotenej informácii, alebo smiete použiť ukázanú procedúru.

7.8.3 Procedúry cvičenia

1. Rezervovať RAM pre koeficienty a oneskorovaciu linku.
2. Založiť ROM tabuľku pre koeficienty a inicializačné podmienky.
3. Inicializovať RAM pre ROM tabuľku.
4. Inicializovať mód procesora.
5. Postaviť vektor resetu.
6. Napíšte kód realizácie sčítavania produktov a funkciu oneskorovacej linky.
7. Zostaviť (asemblovať) program.
8. Zlinkovať program.
9. Spustiť program na simulátore.
10. Prebehnutie cez posledných 40 výstupov.
11. Opustenie simulátora a pohľad do dátového súboru *PLOT OUT.DAT*. Ukážte vášmu inštruktorovi výsledky na skontrolovanie.
12. Voliteľné: Ak vám dovoľuje čas, pouvažujte nad optimalizáciou algoritmu.

7.9 Opakovanie

1. Dajte správnu jazykovú definíciu FIR filtra.
2. Čo je úlohou rotačného buffera v realizácii FIR filtra?
3. Dajte správne jazykové vysvetlenie operácií IIR filtra.
4. Napíšte a vysvetlite niektoré numerické výpočtové problémy s IIR filtrami.
5. Aké sú rozdiely medzi vybratými IIR a FIR filtrami?

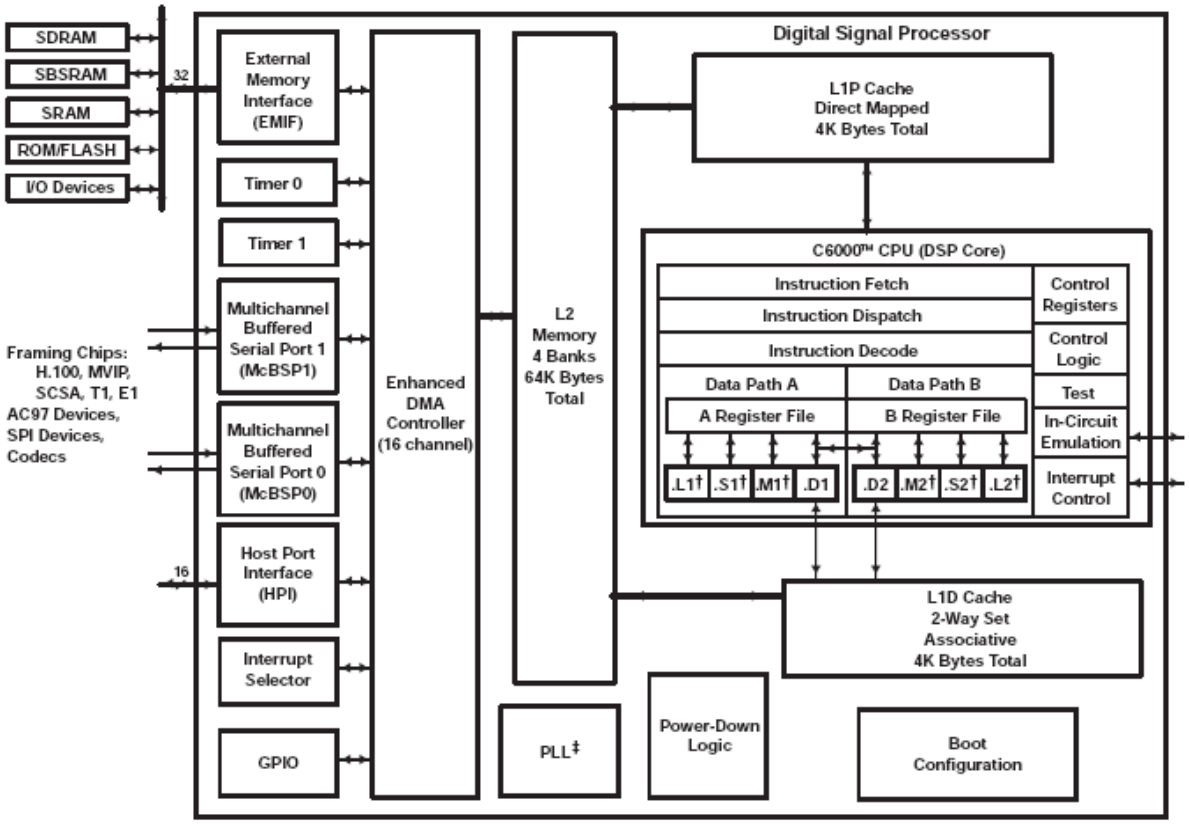
Použité signálové mikroprocesory

Mikroprocesor DSP TMS320C6711

Ako bolo už spomenuté hlavnou výpočtovou jednotkou je DPS TMS320C6711. Je to 32 bit signálový procesor pracujúci s pohyblivou rádovou čiarkou. Pri taktovacej frekvencii 150MHz (teraz sú vyrábané s pracovnou frekvenciou 165MHz, 200MHz a 225MHz)dokáže spraviť až 1200MIPS s pevnou čiarkou a 900MIPS s pohyblivou čiarkou. Je to dané najmä jeho architektúrou jadra, ktoré je rozdelené na dve dátové cesty A a B, kde v každej z nich sa nachádzajú 4 jednotky (.L,.M,.D,.S), tak isto každá dátová cesta má šestnásť 32bitových registrov pre všeobecné použitie. Takouto architektúrou je možné pararelne vykonať v jednom cykle až 8 inštrukcií.

Vnútorná štruktúra TMS320C6711

Jeho bloková schéma je na obrázku 5. Vnútorná pamäť má 72kB a je rozdelená na 2x4kB L1 cache pre program a dáta a 64kB L2 cache, ktorá je rozdelená na 4 banky. Táto L2 pamäť sa dá využiť ako vnútorná pamäť programu a dát, alebo ako vyrovnávacia, tak isto je to možné rozdeliť (4 banky, takže 2-2 alebo 3-1...). Pripojenie vonkajších pamätí a zariadení riadi EMIF, ktorý robí obnovenie dynamickým pamätiam (generovanie signálov RAS, CAS, ..), a redukciu dátovej a adresovacej zbernice. Vďaka nemu je možné jednoducho pripojiť či už 32bit zariadenia , alebo 8bit zariadenia bez potrebnej prídavnej logiky, tak isto aj synchrónne pamäte (SDRAM a SBSRAM) aj asynchrónne pamäte (napríklad SRAM, FLASH, EEPROM). Ďalej obsahuje vnútorné periférie ako sú 2 časovače Timer0 a Timer1, ktoré je možné použiť na počítanie udalostí, alebo na generovanie signálu (frekvencia, šírka), prípadne ako zdroj časovaného prerušenia. Ďalej 2 multikanálové bufferované sériové porty McBSP0 a McBSP1 pre pripojenie prevodníkov a iných zariadení pripojiteľných týmto rozhraním, tieto rozhrania dokážu generovať prerušenia na príjem znaku i na vyprázdnenie vysielacieho bufra. Tým že sú buferované umožňujú kontinuálnosť toku dát. Interrupt selecor je riadiaci obvod maskovateľných prerušení. GPIO sú výstupy/výstupy pre všeobecné použitie. HPI je interfejs ktorý umožňuje priamy prístup druhého procesora do pamäte, čím je umožnená medzi procesorová výmena údajov (namerané výsledky, údaje pre spracovanie, povely...). Všetky výmeny dát medzi priferiami a vnútornou pamäťou sa deje pomocou EDMA kontroléru.



Obr. P1 Vnútorná štruktúra TMS320C6711.

Jadro TMS320C6711

Samotné jadro (na obrázku P1 časť označená C6000 CPU (DSP Core)) je rozdelené na dve dátové cesty A a B, s registrovými sadami A a B, registre sú 32 bitové, pri čom je možné spojiť dva registre do páru a dostať tak 40 bitový register, alebo 64bitový (napríklad A0:A1, párny je LSB, pri 40bit sú MSB bity 64 bitového páru nulové), takto máme dokopy 16 párov. Podrobná štruktúra dátových ciest je na obrázku **Chyba! Nenašiel sa žiaden zdroj odkazov.**.

Pri výbere inštrukcií z pamäte sa naraz načíta 8 inštrukcií (čo je 256 bitov), ktoré tvoria takzvaný vybraný paket (Fetch packet). Inštrukcie v tomto pakete môžu byť vykonávané naraz, alebo postupne, pripadne v zvolených blokoch , ktoré tvoria jeden vykonávaný paket (Execute packet). Inštrukcie ktoré sú v jednom vykonávanom pakete sa spracúvajú naraz a preto každej vykonávacej jednotke môže byť adresovaná iba jedna inštrukcia.

Funkčné jednotky .L1, .L2 slúžia na 32 a 40 bitové aritmetické operácie s pevnou rádovou čiarkou, sčítanie , absolútna hodnota, odčítanie, zmena znamienka, porovnávania,

z logických operácií sú to logický súčin, logický súčet, negácia bit po bite, a presuny. U pohyblivej rádovej čiarke to robí sčítanie, odčítanie a konverzie medzi číslami s pohyblivou rádovou čiarkou jednoduchej precíznosti a dvojitej precíznosti a integerom.

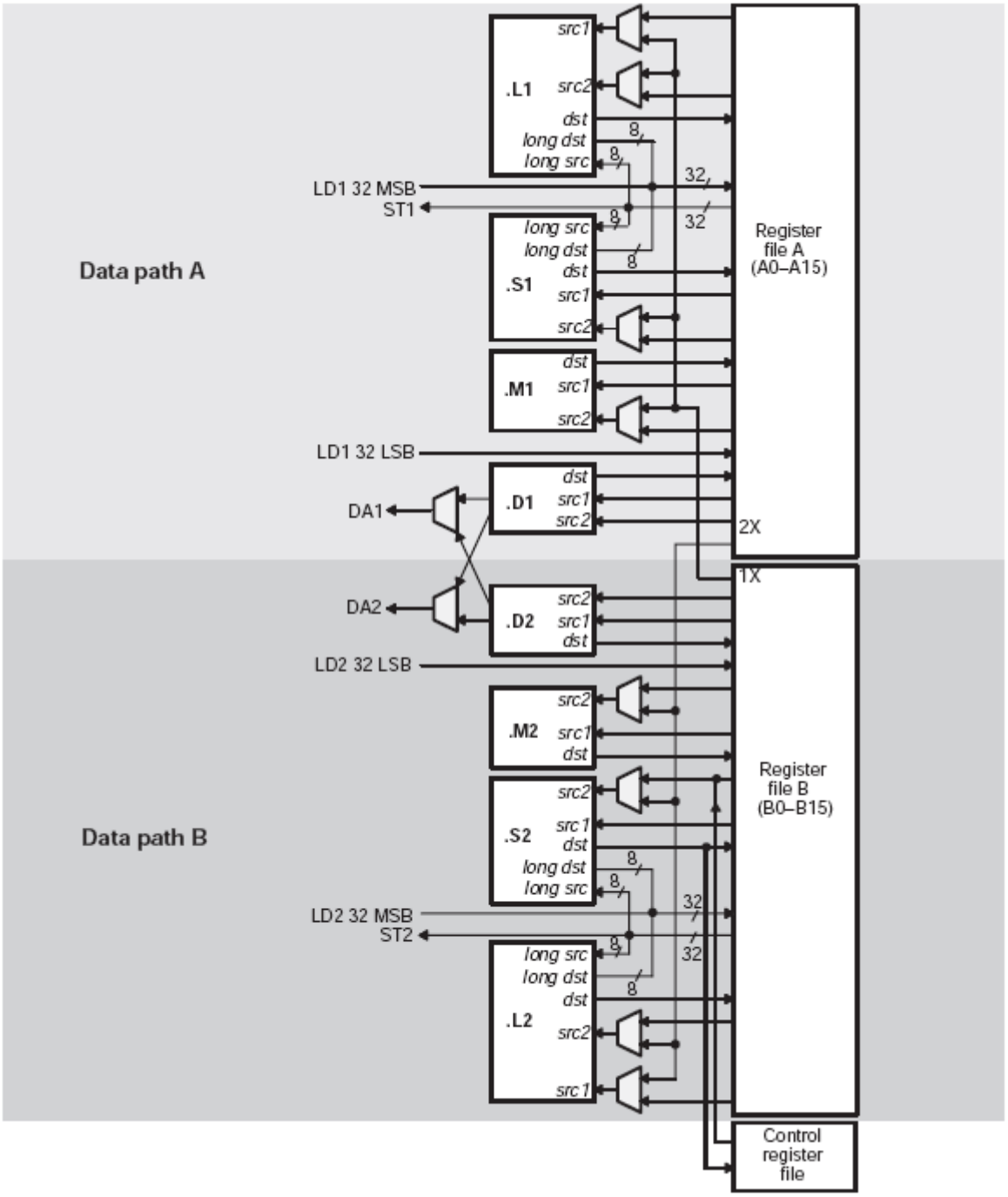
Jednotky .M1, .M2 ako to už ich označenie napovedá sú určené na výkon násobenia, a to platí aj u čísel s pohyblivou rádovou čiarkou.

Jednotky .S1 a .S2 sú pre aritmetické operácie s pevnou rádovou čiarkou (sčítanie, odčítanie, zmena znamienka), logické operácie (logický súčet, logický súčin, exkluzivný súčet, operácie posuvu, negácia bit po bite,), operácie presunu, pri čom tieto jednotky sú zamerané na inštrukcie obsahujúce konštanty. Ďalej vykonávanie operácií skoku buď priameho, alebo určeného registrom IRP, NRP, B0....B15, pri čom tieto registrom určené skoky sú vykonateľné na jednotke .S2.

Na jednotkách .D1, .D2 sa dajú vykonávať matematické operácie súčtu, ale sú hlavne predurčené na výmenu údajov medzi pamäťou a registrami, umožňujú adresovanie pomocou registra, alebo registra + ofset (5bit, alebo 15bit na jednotke .D2), tak isto preinkrement, postinkrement o zvolenú hodnotu (N*sizeof()), alebo pre N=1. S touto jednotkou je možné s vypočítavať adresu pomocou registra AMR, ktorý určuje adresný mód (kruhový, lineárny), a veľkosť bloku.

Na rozdiel od iných architektúr, táto nemá priamo inštrukcie pre vetvenie programu, ale každá inštrukcia môže byť podmienená, podmienka sa vzťahuje na registre B0, B1, B2, A1, A2, pri čom sa testuje ich obsah na nulovú alebo nenulovú hodnotu.

Tak isto je možné použiť ako zdroj údajov pre jednotku obsah registra z druhej dátovej cesty (v rámci rešpektovania vnútorného usporiadania a v jednom vykonávanom pakete môže byť iba jedna inštrukcia tohto charakteru), taktiež je možné pri výmene údajov s pamäťou adresu zobrať z jednej sady registrov a cieľ určiť do druhej sady registrov, samozrejme taktiež s rešpektovaním usporiadania dátovej cesty.



Obr. P2 Štruktúra dátových ciest.