# VEŽBA 2 – Upoznavanje sa ciljnom DSP arhitekturom

## 2.1 Uvod

Arhitekture digitalnih signal procesora prilagođene su izvršavanju algoritama obrade signala, odnosno obradi velikog broja podataka u realnom vremenu. Osobenosti DSP procesora u odnosu na procesore opšte namene su brojne: odvojene memorijske zone za podatke i program, namenske jedinice za generisanje adresa (eng. *Adress generators*), podrška za prenos podatka bez posredovanja samog procesora, podrška za hardverske petlje, namenske instrukcije za množenje i akumuliranje rezultata (MAC) i slično.

Bez obzira da li se ovakve arhitekture programiraju korišćenjem C programskog jezika ili asemblera, razvoj softvera na DSP platformama podrazumeva upoznavanje sa specifičnostima procesora (specifikacijom hardvera), podržanom aritmetikom i skupom asemblerskih instrukcija. Pored ovoga važno je razumeti namenu i ograničenja procesora, spregu sa okolnim sistemom (na primer mikrokontrolerom), koje sprežne podsisteme podržava i slično.

Tokom izrade ove vežbe obradiće sa posebnosti arhitekture 32-bitnog procesora *Cirrus Logic* CS48x sa aritmetikom u nepokretnom zarezu, i na koji su način realizovane sledeće komponente:

- tok podataka
- memorijski prostori
- adresni generator
- aritmetičko logička jedinica
- MAC jedinica

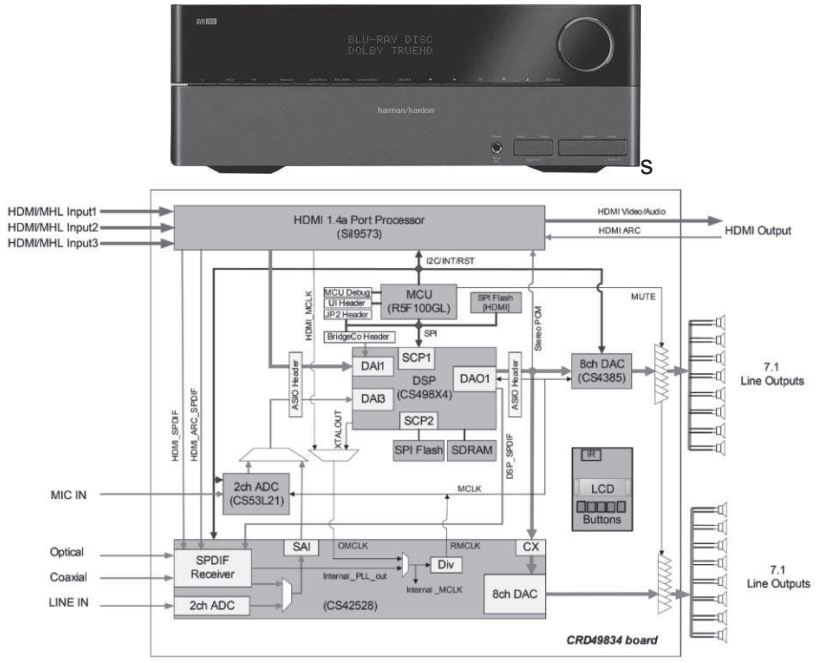
Pored toga, upoznaće se sa sintaksom asemblerskog jezika i skupom instrukcija za rukovanje CS48x procesorom. Naučiće se kako se osobine samog procesora preslikavaju na ograničenja i dodatne mogućnosti prilikom pisanja DSP aplikacija upotrebom asemblerskog jezika.

## 2.2 Arhitektura procesora CS48x

### 2.2.1 Osnovne osobine procesora CS48x

DSP procesori familije CS48x kompanije *Cirrus Logic* su namenjeni obradi audio signala u uređajima potrošačke elektronike. Prvenstveno se ugrađuju u audio/video prijemnike (AVR) i aktivne zvučnike, ali imaju i primenu u televizorima, automobilskim audio sistemima i prenosnim uređajima. Ovi DSP procesori imaju ulogu audio pod-procesora uređaja. Tipičan sistem sa pripadajućom blok šemom arhitekture je prikazan na slici 2.1. Centralno mesto u ovom primeru jeste DSP procesor CS498x4. CS498x4 prihvata kompresovane audio podatke sa HDMI (SiL9573) ili SPDIF (CS42528) serijske sprege. Primljeni audio sadržaj se obrađuje na DSP procesoru i pušta na zvučnicima. Inicijalizaciju celog sistema, kao i opsluživanje korisničke sprege izvršava procesor opšte namene označen kao MCU.

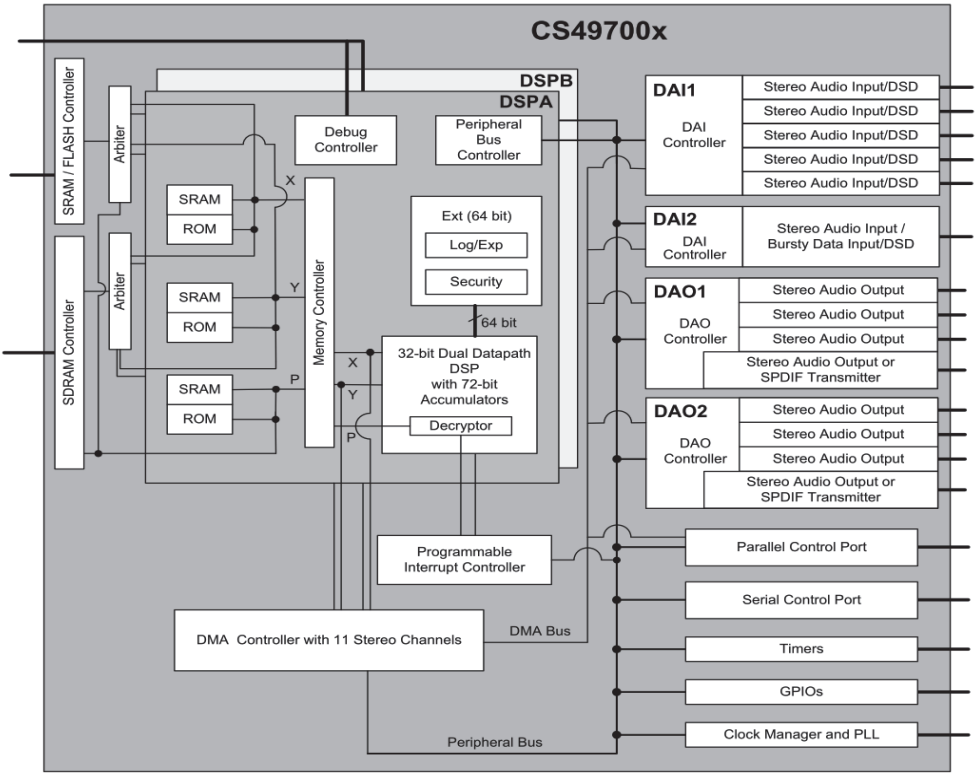
Pripadnici familije DSP procesora CS47x, CS48x i CS49x su zasnovani na istoj arhitekturi DSP jezgra pod imenom Crystal 32, a razlike između modela unutar familije se odnose prvenstveno na broj jezgara, radni takt, i količinu interne memorije.




*Slika 2.1 – Prikaz izgleda i blok dijagram prijemnika audio-video signala zasnovanog na procesoru CS49x*

DSP CS48x je DSP procesor sa dva jezgra, od kojih svako jezgro ima dve odvojene memorije za podatke, *X* i *Y*, i programsku memoriju. Svako jezgro predstavlja tridesetdvobitni *DSP* koji radi sa aritmetikom u nepokretnom zarezu i koji u sebi sadrži osam sedamdesetdvobitnih akumulatora *(a0, a1, a2, a3 i b0, b1, b2, b3)*, četiri *X (x0, x1, x2, x3)* i četiri *Y (y0, y1, y2, y3)* registra za podatke i dvanaest indeksnih registara *(i0, i1, ..., i11)* od kojih svaki ima odgovarajući modul registar *(nm0, nm1, ..., nm11)*.

Sprega između jezgara i spoljašnje memorije ostvarena je upotrebom kontrolera za neposredan pristup memoriji *(eng. Direct Memory Access, DMA)*. *DMA* ima osobinu da može da vrši prenos podataka između perifernih uređaja, spoljašnje memorije ili unutrašnje memorije jezgra, bez intervencije jezgara, što oslobađa više resursa za samu obradu signala.




*Slika 2.2 – Funkcionalni blok dijagram procesora*

Osobine procesora su:

- Harvard arhitektura

- dve odvojene memorije za podatke *(X, Y)* kao i programsku memoriju u svakom jezgru
- dva 32-bitna jezgra koja mogu da vrše po dve *MAC* operacije u jednom taktu
- svako jezgro radi na taktu od 150MHz
- dvanaest ulaznih kanala
- šesnaest izlaznih kanala
- S/PDIF predajnik
- dva serijska kontrolna porta (SPI ili $I^2C$ standard)
- paralelnu kontrolu korišćenjem Intel ili Motorola komunikacionog standarda
- digitalni audio ulaz koji podržava $I^2S$ i levo poravnati (*left justified*) format
- podrška za SDRAM i Flash memorije
- GPIO podrška
- sigurnosni ključ za osiguravanje sadržaja (*firmware*-a)
- mnoštvo realizovanih standarda/algoritama za obradu signala u ROM-u

### 2.2.2 Format asemblerskog jezika

Datoteka sa izvornim kodom predstavlja tekstualnu datoteku koja se parsira kao niz linija koda. Svaka linija se završava karakterom koji označava kraj i u sebi sadrži najviše jednu instrukciju ili asemblersku direktivu. Linija, takođe, može da sadrži samo komentare ili da bude prazna. Linija koja sadrži instrukciju ili direktivu ima sledeći format:

```
<simbol ili prazno_mesto> <operator ili direktiva>
```

Prvi karakter linije je od velike važnosti:

- ako prvi karakter grupe <*simbol ili prazno_mesto*> nije prazno mesto, definisan je simbol i njegova vrednost je postavljena na trenutnu adresu u programskoj ili memoriji za podatke ili označava asemblersku direktivu,
- ako je prvi karakter grupe <*simbol ili prazno_mesto*> prazno mesto, simbol nije definisan za trenutnu adresu.

<*operator ili direktiva*> prati sintaksu operatora i direktiva koje će biti obrađene kasnije.

Asemblerski jezik podržava samo linijske komentare koji počinju karakterom #.

Jezik nije osetljiv na mala i velika slova – labele, operatori i direktive će biti isto interpretirane bilo da su napisana malim ili velikim slovima.

#### *2.2.2.1 Definicija simbola*

Pravilno napisan simbol mora da počinje ili velikim ili malim slovom ili znakom "_". Svaki naredni znak simbola može da bude broj, malo ili veliko slovo ili znak "_". Kao što je već rečeno, simbol mora da se nalazi na početku linije. Ako se ne nalazi na početku, asembler će pretražiti listu simbola da bi utvrdio da li traženi simbol predstavlja deo nekog operatora ili direktive. Ukoliko je simbol pronađen, biće interpretiran u kontekstu operatora ili direktive, dok ukoliko nije pronađen, asembler će generisati grešku.

Neki simboli se ne mogu koristiti, na primer: imena registara, direktive za oznaku memorijskih zona nivoa, itd. Postoji i specijalan slučaj simbola koji se nazivaju *Lokalni simboli*. Konstruišu se koristeći ista pravila kao i za regularne simbole sa izuzetkom prvog znaka koji počinje %, ili, <, ili >. Ova vrsta simbola se koristi kada je vrednost simbola validna u kratkom periodu. Lokalni simboli se ne upisuju u tabelu simbola tako da je njihov opseg ograničen. Kada se jednom izađe van opsega lokalnog simbola on može, za razliku od regularnog simbola, biti definisan ponovo u kodu bez generisanja greške od strane asemblera. Primer definicije lokalnih simbola dat je u sledećem kodu:

```
# while (neki uslov) {
# }
%whiletop:
  # računanje uslova pod
  kojim se petlja prekida
  if (a==0) jmp>whileend
  # …
  # telo petlje – neka obrada
  # …

  jmp <whiletop
  %whileend:
# odavde pa nadalje simboli whileend i whiletop se mogu ponovo
# redefinisati i koristiti pod istim imenom
```

Iz primera se vidi da se simbol definiše tako što se postavi na početak linije i prvi znak postavi na %. Znakovi < i > označavaju smer u kome će lokalni simbol biti tražen – nagore ili nadole.

#### *2.2.2.2 Izrazi*

Vrednosti koje se dobijaju za vreme prevođenja (kao što su adrese, uslovi ili numeričke vrednosti) se nazivaju izrazi. Ukoliko se u izrazu nalazi vrednost u pokretnom zarezu (*float*) tada se taj operand predstavlja kao *float* tip i svi međurezultati se računaju u duploj preciznosti u pokretnom zarezu. Krajnji rezultat je u 32-bitnom formatu u nepokretnom zarezu i vrednost greške se ne nalazi u 32-bitnom opsegu -1.0 <v<= -1.0. Primer izraza je dat u tabeli 2-1, dok su operatori dati u tabeli 2.2.

Tabela 2.1 – Primeri izraza

| Izraz | Dobijanje rezultata |
|---|---|
| 32+3*(20-4) | 32+(3*16) → 32+48 → 80 |
| 77/6*6+mod(77,6) | (77/6)*6+mod(77,6) → (12*6)+mod(77,6) → 72+5 → 77 |
| strcat(“moo”,’cow’)=”moocow” | “moocow”=”moocow” →-1 (true) |
| 14>=0&14<10 | (14>=0)&14<10 → -1&(14<10) → -1&0 → 0 (false) |
| 240&0x3f \| 240&0x7f00 | (240&0x3f) \| 240&0x7f00 → 48 \| (240&0x7f00) → 48 \| 0 →48 |

Tabela 2.2 – Operatori

| Operator | Tip operatora | Opis |
|---|---|---|
| **+** | Unarni | unarni plus, operand ostaje nepromenjen |
| **-** | Unarni | unarni minus, operand je aritmetički negiran |
| **~** | Unarni | komplement, operand logički negiran |
| **!** | Unarni | NE, negiranje po Bulovoj algebri (nula za netačno, ostalo za tačno) |
| **+** | Binarni aritmetički | Sabiranje |
| **-** | Binarni aritmetički | Oduzimanje |
| ***** | Binarni aritmetički | Množenje |
| */* | Binarni aritmetički | Deljenje |
| **&** | Binarni logički | I |
| **\|** | Binarni logički | Ili |
| **^** | Binarni logički | Ekskluzivno ili |
| **=** | Poređenja* | “jednako” |
| **!=** | Poređenja* | “različito” |
| **<>** | Poređenja* | “različito” |
| **>** | Poređenja* | “veće” |
| **>=** | Poređenja* | “veće ili jednako” |
| **<** | Poređenja* | “manje” |
| **<=** | Poređenja* | “manje ili jednako” |

*Poređenja kao rezultat daju nulu za netačno a -1 za tačno

#### *2.2.2.3 Funkcije ugrađene u prevodilac*

Prevodilac u sebi ima ugrađene funkcije koje programeru mogu da pomognu prilikom konfigurisanja koda. Lista najčešće korišćenih funkcija data je u tabeli 2.3.

Tabela 2.3 – Funkcije ugrađene u prevodilac

| Ugrađena funkcija | Opis |
|---|---|
| .defined(<simbol>) | Funkcija vraća nulu (netačno) ako simbol nije definisan, dok u suprotnom vraća vrednost različitu od nule (tačno) |
| .isabsolute(<izraz>) | Funkcija vraća vrednost različitu od nule (tačno) ako izraz predstavlja numeričku vrednost ili apsolutnu adresu, dok u suprotnom vraća nulu (netačno) |
| .isfloat(<izraz>) | Funkcija vraća vrednost različitu od nule (tačno) ako izraz predstavlja numeričku vrednost u aritmetici u pokretnom zarezu, dok u suprotnom vraća nulu (netačno) |
| .isint(<izraz>) | Funkcija vraća vrednost različitu od nule (tačno) ako izraz predstavlja numeričku vrednost u aritmetici u nepokretnom zarezu, dok u suprotnom vraća nulu (netačno) |
| .isstring(<izraz>) | Funkcija vraća vrednost različitu od nule (tačno) ako izraz predstavlja niz karaktera, dok u suprotnom vraća nulu (netačno) |
| .abs(<izraz>) | Funkcija vraća apsolutnu vrednost izraza |
| .acos(<izraz>) | Funkcija vraća arkus kosinus izraza |
| .asin(<izraz>) | Funkcija vraća arkus sinus izraza |
| .atan(<izraz>) | Funkcija vraća arkus tangens izraza |
| .cos(<izraz>) | Funkcija vraća kosinus izraza |
| .sin(<izraz>) | Funkcija vraća sinus izraza |
| .tan(<izraz>) | Funkcija vraća tangens izraza |
| .exp(<izraz>) | Eksponencijalna funkcija. Vraća vrednost u aritmetici u pokretnom zarezu. Primer: *EXP1 .EQU* ***.EXP(****1.0****)*** *# EXP1 = 2.718282* |
| .log(<izraz>) | Vraća prirodni logaritam izraza. |
| .log10(<izraz>) | Vraća logaritam sa osnovom 10 izraza. |
| .log2(<izraz>) | Vraća logaritam sa osnovom 2 izraza. |
| .b2f(<izraz>) | Pretvara izraz iz celobrojnog u izraz u pokretnom zarezu |
| .f2b(<izraz>) | Suprotno od .b2f |
| .strcat(<string1>, <string2>[, …<stringN>]) | Vraća niz karaktera dobijen spajanjem više nizova. |

#### *2.2.2.4 Direktive*

Asemblerske direktive imaju format sličan instrukcijama, tj. one predstavljaju instrukcije prevodioca, a ne instrukcije ciljne DSP platforme. Direktive pomažu programeru da konfiguriše i kontroliše kod. Postoje direktive koje dozvoljavaju deljenje objekata između datoteka sa izvornim kodom (slično podeli u, na primer, C programskom jeziku kroz *.h i *.c datoteke), zatim direktive koje omogućuju zauzimanje i inicijalizaciju memorije, direktive koje omogućuju uslovno prevođenje, i na kraju, direktive koje programeru daju mogućnost da objedini često korišćene instrukcije – makroi.

##### 2.2.2.4.1 Modularnost koda

Asemblerske direktive koje omogućavaju deljenje objekata između datoteka:

- ***.include <putanja i ime datoteke>*** – ova direktiva otvara datoteku, ubacuje njen sadržaj i prevodi ga. Najčešća upotreba ove direktive je ubacivanje datoteka sa definicijama objekata čija je namena da budu vidljivi u drugim datotekama sa izvornim kodom (to su datoteke sa ekstenzijom *.h, koje po svojoj strukturi podsećaju na datoteke iz, na primer, programskog jezika C)
- ***.public <simbol>*** – definiše simbol kao javni dajući mogućnost za njegovo korišćenje u drugim datotekama sa izvornim kodom.
- ***.extern <simbol***> – Unutar datoteke u kojoj se nalazi, definiše simbol kao eksterni, odnosno označava da je simbol definisan unutar neke druge datoteke sa izvornim kodom kao javni.

##### 2.2.2.4.2 Direktive za oznaku memorijskih segmenata

Uopšteni format ovih direktiva je:

[<ime segmenta>] .<klasa segmenta> [at <adresa> | align <moduo>, gde je:

- *ime segmenta* – opcioni parametar koji predstavlja simbol koji jednoznačno određuje početak segmenta,
- *klasa segmenta* – određuje tip memorije gde će segment biti lociran. Primer je .code_ovly za programsku i .xdata_ovly, .ydata_ovly i .data_ovly za dodelu X, Y I XY(64 - bitne) memorije za podatke.
- *adresa* – opcioni parametar kojim se segment može smestiti na željenu adresu
- *modulo* – opcioni parametar kojim se povezivaču daje instrukcija da segment smesti tako da je početna adresa poravnata na zadatu vrednost.

*at* <*adresa*> i *align* <*moduo*> su međusobno isključivi, tj. ne mogu zajedno da se nalaze u definiciji jednog segmenta.

Primer segmenta:

```
Mp3_YStatics       .ydata_ovly .at(0x856)
Mp3_variable1      .dw (0xdeadbeef)
Mp3_variable2      .dw (0xdeadbeef)
Mp3_HeaderParser   .code_ovly
X_S_HeaderParserFunction:

   #Telo funkcije
ret
```

#### 2.2.2.4.3 Direktive za dodelu i inicijalizaciju memorije

- ***.bss <N>*** – Daje prevodiocu instrukciju da izvrši zauzimanje N lokacija u segmentu unutar kojeg se nalazi direktiva.

Primer:

```
.xdata_ovly
X_BY_TempBuffer      .bss (0x20)
```

- ***.bsc <N>,<V>*** – Daje prevodiocu instrukciju da izvrši zauzimanje N lokacija u segmentu unutar kojeg se nalazi direktiva, i da ih sve inicijalizuje na vrednost V.

Primer:

```
.xdata_ovly
X_BY_TempBuffer    .bsc (0x20), 0xdeadbeef
```

- ***.dw <V>*** – Daje prevodiocu instrukciju da izvrši zauzimanje jedne lokacije u segmentu unutar kojeg se nalazi direktiva, i da je inicijalizuje na vrednost V.

Primer:

```
.xdata_ovly
X_VY_TempVariable                    .dw (0xdeadbeef)
```

- ***.dw <V1><V2>*** – Predstavlja specijalan slučaj gore navedene direktive. Daje prevodiocu instrukciju da izvrši zauzimanje jedne lokacije u XY segmentu unutar kojeg se nalazi direktiva, i da njen X deo inicijalizuje na V1, a Y deo na V2.

Primer:

```
. .data_ovly
X_VL_TempVariableLong .dw (0xdeadbeef), (0xdedababa)
```

#### 2.2.2.4.4 Uslovno prevođenje koda

Prosleđivanjem prevodiocu parametara koji u sebi sadrže simbole mogu se označiti delovi koda koji se prevode samo u slučaju da je određeni simbol definisan. To se postiže kombinovanjem direktive *.defined* i direktiva *.if*, *.elseif* i *.endif*.

Primer:

```
        .if
defined(_MIPS_PROFILING_ENABLED
_) X_S_DoMipsProfile:

        #Telo funkcije
        ret
        .endif
```

Funkcija *X_S_DoMipsProfile* nije potrebna u krajnjem proizvodu, ali služi za proveru potrošnje resursa modula. Zato se ona uslovno prevodi samo onda kada je simbol *_MIPS_PROFILING_ENABLED_* definisan. Ovim se programeru daje mogućnost da pojedine delove koda isključi iz procesa prevođenja i uključi ih po potrebi, čime se izbegava ponovno pisanje funkcije svaki put kada, na primer, treba uraditi proveru potrošnje resursa.

#### 2.2.2.4.5 Definicija makroa

Makroi se definišu na sledeći način:

- počinju direktivom *.macro,*
- prva sledeća linija sadrži prototip makroa koji označava način njegovog pozivanja u kodu. Format ove linije je: *<simbol koji predstavlja ime makroa> <%arg1> <%arg2> ...,*
- zatim sledi telo makroa koje sadrži instrukcije,
- na kraju stoji direktiva koja označava kraj makroa – *.endm.*

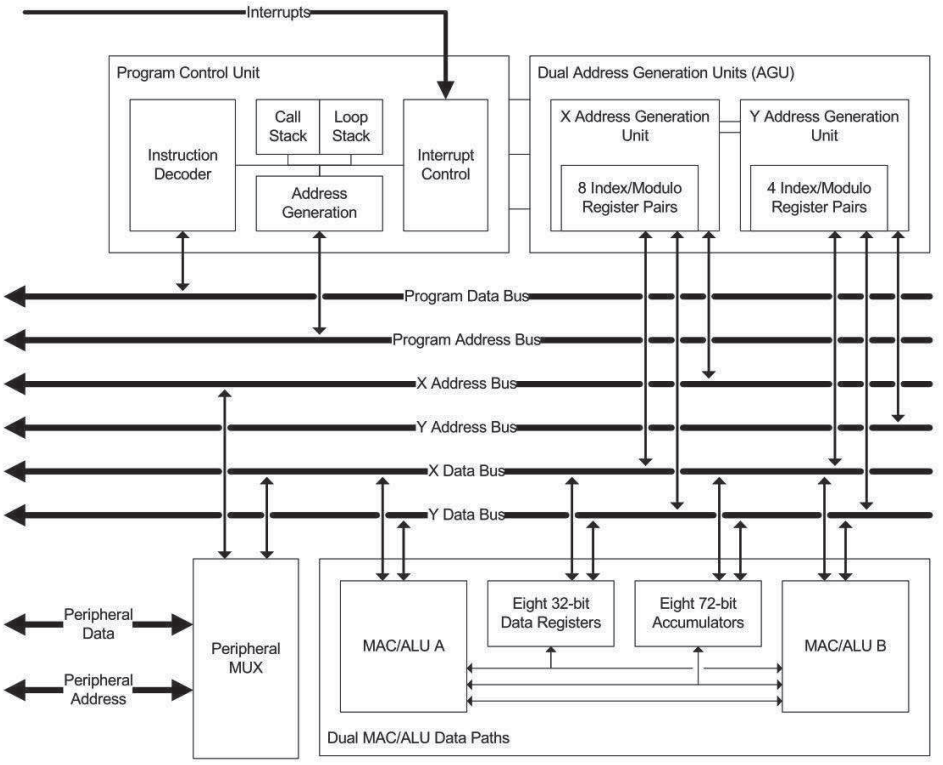
Primer:

```
      .macro
OS_MALLOC %os_malloc_function_var,
%xmem_addr_base_ptr, %size
      uhalfword(x0) = (%size)
      i1 =
xmem[%os_malloc_function_var]
      i0 = (%xmem_addr_base_ptr)
      call (i1) # %os_malloc_function
.endm
```

### 2.2.3 Arhitektura i skup instrukcija procesora CS48X

Jezgro procesora CS48x zasnovano na aritmetici u nepokretnom zarezu, predstavlja programabilan procesor čije su visoke performanse ostvarene kroz

visok stepen paralelizma. Koristi predstavu razlomljenih brojeva u drugom komplementu i sadrži magistrale za dva odvojena memorijska prostora (X i Y), i jedan programski memorijski prostor (P). Blok dijagram arhitekture prikazan je na slici 2.3.

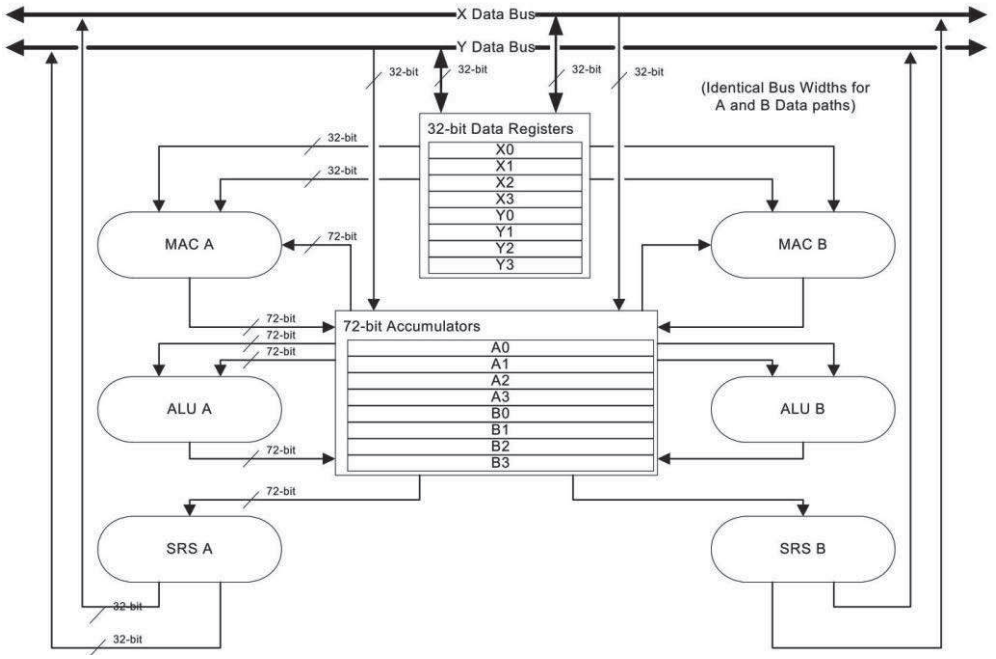



*Slika 2.3 - Blok dijagram DSP arhitekture CS48x*

### *2.2.3.1 Tok podataka*

Blok dijagram jezgra prikazan je na slici 2.4. Jezgro sadrži jedinicu za kontrolu toka programa, paralelne jedinice za generisanje adresa (AGU) i paralelne tokove podataka (A i B). Jedinice za generisanje adresa sadrže dvanaest 16-bitnih (indeksnih) registara (i0, i11) za čuvanje adresa i dvanaest (*modulo-ofset*) 16-bitnih registara (nm0, nm11) koji rade u sprezi sa indeksnim registrima u cilju obezbeđivanja različitih režima adresiranja. Svaka putanja podataka ima četiri 32-bitna registra opšte namene (x0-x3, y0-y3) i četiri 72-bitna akumulatorska registra (a0-a3, b0-b3). Akumulatori se sastoje od 3 podregistra sa oznakama: *Guard* (8-bit), *High* (32-bit) i *Low* (32-bit), pri čemu se svakom delu može zasebno pristupati. Svaka putanja podataka takođe ima jednu MAC (*Multiply-Accumulate*), SRS

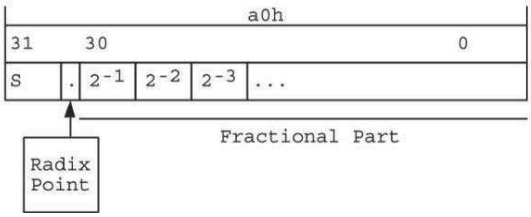
(*Shifter/Rounder/Saturator*) i ALU (*Arithmetic and Logical Unit*) jedinicu. ALU jedinica obavlja sve logičke operacije nad akumulatorskim registrima.




*Slika 2.4 - Blok dijagram DSP jezgra CS48x*

### *2.2.3.2 Prikaz brojeva u nepokretnom zarezu*

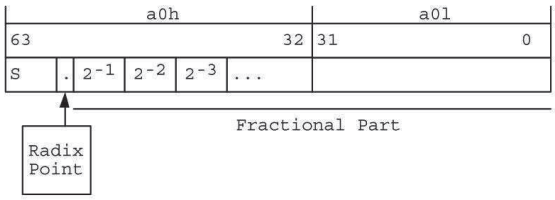
Prikaz brojeva korišćen u procesoru je notacija razlomljenih brojeva u drugom komplementu, tj. prikaz sa nepokretnim zarezom (*fixed point*). Oznaka S je bit znaka.




*Slika 2.5 - 32-bitni prikaz brojeva sa nepokretnim zarezom*

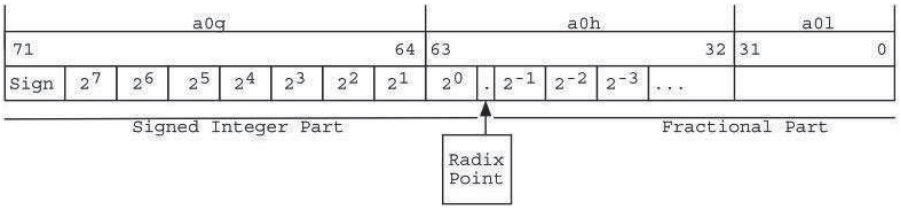
- **32-bitni prikaz** (Slika 2.5)**.** Najveći pozitivan broj koji može biti prikazan je 0x7fffffff ($1-2^{31}$ decimalno). Najveći negativan broj koji može biti prikazan je 0x80000000 (-1.0 decimalno). Ovo odgovara formatu <1.31>.

- **64-bitni prikaz** (Slika 2.6)**.** Najveći pozitivan broj koji može biti prikazan je 0x7fffffff ffffffff($1-2^{-64}$ decimalno). Najveći negativan broj koji može biti prikazan je 0x80000000 00000000 (-1.0 decimalno). Ovo odgovara formatu <1.63>.




*Slika 2.6 – 64-bitna prikaz brojeva sa nepokretnim zarezom*

- **72-bitni prikaz** (Slika 2.7)**.** Najveći pozitivan broj koji može biti prikazan je 0x7f ffffffff ffffffff ($256-2^{-63}$). Najveći negativan broj koji može biti prikazan je 0x80 00000000 00000000 (-256.0 decimalno). Ovo odgovara formatu <9.63>.




*Slika 2.7 - 72-bitni prikaz brojeva sa nepokretnim zarezom*

### *2.2.3.3 Prenos podataka u/iz akumulatora*

Prilikom upisa 32-bitne vrednosti u akumulator, sa X ili Y magistrale podataka, ona će biti upisana u gornja 32 bita (*high deo*) akumulatora. Donja 32 bita će biti postavljena na nulu, dok će se u gornjih osam bita (*guard*) upisati sve jedinice ili nule u slučaju negativnog ili pozitivnog broja.

Upisom 64-bitne vrednosti u akumulator 32 bita sa X magistrale će biti upisana u gornja 32 bita, dok će 32 bita sa Y magistrale biti upisana u donja 32 bita akumulatora. Gornjih 8 bita će se menjati na isti način kao i kod upisa 32 bita. Sa slike 2.4 se vidi da svaka grupa akumulatora i registara za podatke ima svoju SRS jedinicu koje predstavljaju jedinu spregu za upis vrednosti iz akumulatora u registre za podatke ili na X/Y magistrale. Ime je dobila po redosledu operacija koje obavlja –

pomeranje (*Shift*), zaokruživanje (*Round*) i zasićenje – ograničavanje opsega (*Saturate*).

Tabela 2.4 – Primeri operacije nad podacima prilikom prolaska kroz SRS

| Operacija | Operand | Rezultat |
|---|---|---|
| x0 = a0 (saturacija) | a0 = 0x00.c0000000.00000000 | x0 = 0x7fffffff |
| x0 = a0 (saturacija) | a0 = 0x80.c0000000.00000000 | x0 = 0x80000000 |
| x0 = a0 (saturacija) | a0 = 0xff.ffffffff.00000000 | x0 = 0xffffffff *1 |
| x0 = a0 (zaokruživanje) | a0 = 0x00.00000001.80000000 | X0 = 0x00000001 – odsecanje<br>X0 = 0x00000002 – dodavanje ½<br>X0 = 0x00000001 – zaokruž. na nulu<br>X0 = 0x00000001 ili 0x00000002 – dither |
| x0 = a0 (zaokruživanje) | a0 = 0xff.80000000.00000001 | X0 = 0x80000000 – odsecanje<br>X0 = 0x80000000 – dodavanje ½<br>X0 = 0x80000001 – zaokruž. na nulu<br>X0 = 0x80000000 ili 0x80000001 – dither |
| x0 = a0 (pomeranje sa postavljenim modom zaokruživanja – Odsecanje) | a0 = 0x01.80000001.80000000 | X0 = 0x7fffffff – bez pomeranja<br>X0 = 0x7fffffff – „>>1“<br>X0 = 0x60000000 – „>>2“<br>X0 = 0x7fffffff – „<<1“ |
| x0 = a0 (pomerajne sa postavljenim modom zaokruživanja – dodaj ½ i odseci) | a0 = 0x01.80000001.80000000 | X0 = 0x7fffffff– bez pomeranja<br>X0 = 0x7fffffff– „>>1“<br>X0 = 0x60000000– „>>2“<br>X0 = 0x7fffffff– „<<1“ |

*1 -broj je ostao nepromenjen

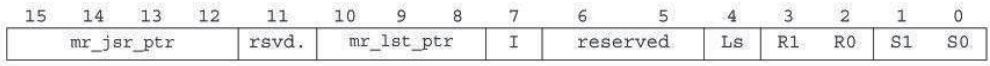
Vrednost akumulatora koja se upisuje u registre može biti pomerena za jedno ili dva mesta udesno, jedno mesto ulevo ili da bude propuštena bez pomeranja. Podaci u akumulatoru uvek ostaju nepromenjeni.

Zaokruživanje vrednosti prilikom pomeranja sadržaja akumulatora na X ili Y magistralu određuje način na koji će se tretirati donja 32 bita akumulatora:

- odsecanje – donja 32 bita će biti ignorisana,

- dodavanje ½ i odsecanje – na akumulator će biti dodana vrednost 0x00.00000000.80000000 i donja 32 bita će biti zanemarena,
- zaokruživanje na nulu – pozitivni brojevi će biti prosto odsečeni, dok će u slučaju negativnih brojeva gornja 32 bita akumulatora bit uvećana za jedan pre odsecanja,
- Dodavanje *dithera* i odsecanje – ako su gornja 4 bita donjeg dela akumulatora veća od četvorobitnog slučajnog broja (oba broja se uvek tretiraju kao pozitivni) gornja 32 bita akumulatora se uvećavaju za jedan pre odsecanja.

Primeri pomeranja vrednosti u/iz akumulatora i operacija koje vrši SRS jedinica dati su u tabeli 2.4.




*Slika 2.8 - Mode Registar*

Tabela 2.5 – MR registar

| Biti | Ime polja | Opis |
|---|---|---|
| 15:12 | mr_jsr_ptr | Pokazivač na call stek |
| 11 | rezervisano | Rezervisano |
| 10:8 | mr_lst_ptr | Pokazivač na loop stek |
| 7 | I | Prekidi uključeni/isključeni |
| 6:5 | rezervisano | Rezervisano |
| 4 | Ls | Ako je postavljen na 1 podaci pomereni iz donjeg dela akumulatora (a0l) će biti logički pomereni za jedno mesto udesno |
| 3:2 | R1, R0 | Biti koji određuju tip zaokruživanja:<br>R1, R0 – Režim zaokruživanja<br>00 – Bez zaokruživanja<br>01 – Dodaj ½ i odseci<br>10 – Zaokruži na nulu<br>11 – Dither |
| 1:0 | S1, S0 | Biti koji određuju tip pomeranja:<br>S1, S0 – Režim pomeranja<br>00 – Bez pomeranja<br>01 – “>>1”<br>10 – „>>2“<br>11 – „<<1“ |

Način na koji će se podaci menjati prilikom prolaska kroz SRS jedinicu biraju se postavljanjem bita *Mode Registra* (*MR*) čija je struktura prikazana na slici 2.8. Moguće vrednosti polja MR registra i opis njihovih značenja data su u tabeli 2.5.

### *2.2.3.4 Adresni generatori*

AGU (*Address Generation Unit*) se sastoji iz skupa od 12 parova registara – 16-bitnih indeksnih registara (i0-i11) i 16-bitnih modulo-ofset registara (nm0-nm11). NM registri sadrže 4-bitni modulo deo (biti [15:12]) i 12-bitni ofset deo. Ofset deo ažurira sadržaj indeksnog registra a modulo deo služi za specificiranje jednog od tri tipa adresiranja: *lineranog*, *reverznog binarnog* ili *modulo* adresiranja. Ofset deo se tretira kao 12-bitni označen broj, i kao takav može ažurirati adresu u odgovarajućem indeksnom registru za vrednost u opsegu od -2048 do 2047 (0x800-0x7ff). Adresni režimi su prikazani u tabela 2.6.

Tabela 2.6 – Adresni režimi

| Modulo deo NM registra (biti [15:12]) | Adresni režimi |
|---|---|
| 0x0 | Linearno adresiranje |
| 0x1 | Modulo 4 |
| 0x2 | Modulo 8 |
| 0x3 | Modulo 16 |
| 0x4 | Modulo 32 |
| 0x5 | Modulo 64 |
| 0x6 | Modulo 128 |
| 0x7 | Modulo 256 |
| 0x8 | Modulo 512 |
| 0x9 | Modulo 1024 |
| 0xa | Modulo 2048 |
| 0xb | Modulo 4096 |
| 0xc | Modulo 8192 |
| 0xd | Modulo 16384 |
| 0xe | Modulo 32768 |
| 0xf | Bit-Inverzno adresiranje |

#### 2.2.3.4.1 Adresni režimi

**Neposredno adresiranje**, koje se koristi za prenos podataka iz memorije u akumulator ili u registar za podatke, postoji u dve verzije: verzija koja zauzima celu instrukcionu reč sa kojom se mogu adresirati sve 16-bitne adrese i verzija koja ne zauzima celu instrukcionu reč sa kojom se mogu adresirati 6-bitne adrese. Druga verzija neposrednog adresiranja se može u paraleli izvršavati sa aritmetičkim i logičkim operacijama. Kao adresa može se zadati simbol ili broj.

Primer neposrednog adresiranja:

```
# Long word instruction
a0 = ymem[0x811]              #<--|a0 is 0x00.00000011.00000000

a0 = ymem[X_BY_BitReversedBuffer] #<--|a0 is 0x00.00000000.00000000 (example
with #^__|symbol as address)

# Short word instruction
b0 = xmem[3]; a0 = a0 << 1    #<--|b0 = 0x2000; a0 will be shifted let by one
bit
```

**Posredno adresiranje** je operacija koja koristi pola instrukcione reči (8 bita). X, Y i P memorijske zone adresiraju se korišćenjem indeksnih registara i0,..., i11.

Primer indeksnog adresiranja:

```
x0 = xmem[i1]
y0 = ymem[i1]
x0,y0 = xymem[i1]
a0 = xymem[i1]
```

Slede tri specifična načina posrednog adresiranja:

**Modulo adresiranje** se može iskoristiti za implementaciju kružnih bafera čija je veličina stepen dvojke, u opsegu od 4 do 32768. Početna adresa kružnog bafera mora biti umnožak njegove veličine (na primer bafer veličine 32 mora da počinje na jednoj od sledećih adresa – 0, 32, 64, ...). Ovo se postiže dodavanjem direktive za poravnanje prilikom zauzimanja memorijskog niza. Prilikom inkrementiranja indeksnog registra sa odgovarajućim NM registrom, vrednost adrese u indeksnom registru će se vratiti na početak bafera kada bude dostignuta krajnja adresa bafera. Gornja četiri bita NM registra određuju kada i kako će se koristiti modulo adresiranje dok se sa ostalih 12 bita određuje korak za koji će biti uvećan sadržaj indeksnog registra.

Primer modulo adresiranja:

```
  lesson03AGU_y_data    .ydata_dec at (0x800) #<--| modulo 32 buffer

  X_BY_Modulo32Buffer    .bss (0x20)
  lesson03AGU_code        .code_dec

    # Fill X_BY_Modulo32Buffer with
     i0 = (X_BY_Modulo32Buffer)    #<--| i0 points to the beginning of buffer
     a0 = 0
     uhalfword(b0) = (1)
     do(0x20),>Loop
  %Loop: ymem[i0] = a0; i0+=1; a0 = a0 + b0
```

```
    # Example 1. Modulo with increment step 1
    i0 = (X_BY_Modulo32Buffer)      #<--| i0 points to the beginning of buffer
    nm0 = (0x4000)                #<--| Set modulo to 32
    do(0x30),>Loop
   %Loop:    a0 = ymem[i0]; i0+=1    #<--| 33 th iteration will read value from
                      #^__| beginning of buffer - it will wrap arround

    # Example 1. Modulo with increment step 3
    i0 = (X_BY_Modulo32Buffer)     #<--| i0 points to the beginning of buffer
    nm0 = (0x4003)               #<--| Set modulo to 32 with increment step 3
    do(0x30),>Loop

   %Loop:    a0 = ymem[i0]; i0+=n   #<--| After each iteration index register i0
                     #^  | will be incremented for step size 3.
                     #|  | After 10 iteration i0 will be set to
                     #|  | 0x81e and in next iteration i0 will be
                     #|  | incremented by 3, wrapped arround and set
                     #|__| to 0x801.
```

**Linearni režim adresiranja** je sličan modulo adresiraju. Razlikuju se u tome što veličina bafera ne mora da bude umnožak broja dva i što su gornja 4 bita NM registra uvek postavljena na nulu. Ako se prilikom linearnog adresiranja dođe do kraja bafera i nastavi se čitati/upisivati dalje, indeks registar neće biti postavljen na početak bafera već će se čitanje/upisivanje nastaviti dalje.

**Posredno adresiranje sa bit obrnutim redosledom** ima primenu u FFT (*Fast Fourier Transform*) i IFFT (*Inverse Fast Fourier Transform*) algoritmima, dva najčešće primenjivana algoritma u digitalnoj obradi signala. Izbor ovog načina adresiranja se vrši postavljanjem gornja 4 bita NM registra na 0xf. Ako je bafer veličine $2^k$ donjih 12 bita NM registra treba inicijalizovati na vrednost $2^{k-1}$. Indeksni registar se inicijalizuje na bilo koju adresu koja se nalazi između donje i gornje granice koje se računaju kao $k*2^t$ i $(k*2^t) + (2^t-1)$ respektivno – t je početna adresa. Obično se indeksni registar postavlja na početak bafera.

Primer reverznog binarnog adresiranja:

```
    i0 = (X_BY_Modulo32Buffer)    #<--| Size is 32
    nm0 = (0xf010)             #<--| Set N part to 0xf and m part to 16
    i1 = (X_BY_BitReversedBuffer)
    do(0x20),>Loop
         a0 = ymem[i0]; i0+=n    #<--|We are reading from X_BY_Modulo32Buffer in
BitReverse
%Loop:    ymem[i1] = a0; i1+=1    #<--| mode and writing it in X_BY_BitReversedBuffer.
```

#### 2.2.3.4.2 Uvećavanje/Umanjivanje sadržaja indeksnog registra

Uvećavanje/umanjivanje sadržaja indeksnih registara obavlja se bez korišćenja akumulatora i registara za podatke. Za ove operacije postoje sledeće mogućnosti:

- `i#±=1` – uvećaj ili umanji sadržaj indeksnog registra za 1
- `i#±=2` – uvećaj ili umanji sadržaj indeksnog registra za 2
- `i#±=n` – uvećaj ili umanji sadržaj indeksnog registra za n, gde je n veličina koraka koja se nalazi u N delu odgovarajućeg NM registra

gde je (# = 0, 1, …11).

Ove operacije mogu da se nalaze u istoj instrukcionoj reči sa čitanjem/pisanjem podataka u/iz memorije i aritmetičkim i logičkim operacijama.

Primer:

```
i0 = (X_BY_BitReversedBuffer)     #<--| i0 points to the first element of
                                  #^__| X_BY_BitReversedBuffer
nm0 = (3)
nop
a0 = ymem[i0]; i0+=1         #<--| Read first element and increment pointer
                             #^__| to points to the second element

a0 = ymem[i0]; i0+=2         #<--| Read second element and increment pointer
                             #^__| to points to the third element

a0 = ymem[i0]; i0-=n         #<--| Read third element and decrement pointer
                             #^__| to points to the beginning of buffer

a0 = x0 * y0;  i0+=n         #<--| Perform multiplication and update i0
```

Osvežavanje sadržaja indeksnog registra može se obaviti i dodavanjem konstante na vrednost indeksnog registra i smeštanjem rezultata u isti ili drugi indeksni registar. Ovakva instrukcija zauzima čitavu instrukcijsku reč.

Primer:

```
i1 = i0 + (0x200)            #<--| i1 will be i0 incremented by 0x200
```

#### 2.2.3.4.3 Učitavanje vrednosti u indeksni registar

Učitavanje vrednosti u indeksni registar se može obaviti na jedan od sledećih načina:

- iz registra `i0 = x0`,
- iz simbola ili konstante `I5 = (0x1234)`,
- iz simbola i konstante na specijalan način `I0 = (0) + (0x1234)`.

Prilikom učitavanja vrednosti u indeksni registar postoje ograničenja. Zbog protočne strukture (*pipeline*) koja obuhvata 3 novoa: Zahvatanje (*Fetch*), Dekodiranje

(*Decode*) i Izvršavanje (*Execute*) instrukcije i sa obzirom na činjenicu da se ažuriranje indeksnih registara obavlja u drugoj fazi, nastaje situacija da će za narednu instrukciju koja koristi AGU vrednost biti nepripremljena jer će se nalaziti u fazi izvršenja. Međutim, ako se odabere treći način učitavanja vrednosti, rezultat se može koristiti odmah. Primer:

```
# Example 1. With nop
i0 = (X_BY_BitReversedBuffer)        #<--| i0 points to the first element of
                          #^__| X_BY_BitReversedBuffer

nop                      #<--| this is necessary

a0 = ymem[i0]; i0+=1         #<--| Read first element and increment pointer
                             #^__| to points to the second element

# Example 1. Without nop
i0 = (0) +  (X_BY_BitReversedBuffer)  #<--| i0 points to the first element of
                                      #^__| X_BY_BitReversedBuffer


a0 = ymem[i0]; i0+=1     #<--| Read first element and increment pointer
                         #^__| to points to the second element
```

### *2.2.3.5 Instrukcije kontrole toka izvršenja programa*

Ove operacije spadaju u grupu koje zauzimaju 32 bita instrukcione reči i ne mogu se izvršavati u paraleli sa drugim operacijama.

#### 2.2.3.5.1 Hardverska petlja

Koristi se za ponavljanje skupa instrukcija određen broj puta. Broj ponavljanja je ili 10-bitna konstanta ili 16-bitni sadržaj indeksnog registra. Nula nije dozvoljena vrednost. Nakon završetka rada hardverske petlje, programski brojač se postavlja na prvu instrukciju nakon poslednje instrukcije petlje – iz ovog sledi da ugnježdene petlje ne mogu da dele istu krajnju adresu.

Primer:

```
# Example 1. constant as loop counter
    a0 = 0
    uhalfword(b0) = (1)
    do(32),>Loop             #<--| do(32)
%Loop: a0 = a0 + b0            #<--|    a0++;

# Example 2. loop counter as value in index reg.
    i0 = (32)
    a0 = 0
    do(i0),>Loop             #<--| do(32)
%Loop: a0 = a0 + b0            #<--|    a0++;
```

#### 2.2.3.5.2 Prekidanje hardverske petlje

Kao što je već rečeno, arhitektura procesora podržava maksimalno osam ugnježdenih hardverskih petlji. Informacije o svakoj od njih se čuvaju na DO steku. Nakon što se petlja završi, pokazivač na stek se umanjuje za 1 tako da pokazuje na gornju petlju (ukoliko postoji). Ponekad se u implementaciji nekog modula zahteva prekidanje hardverske petlje pre nego što se završi poslednja iteracija zadata brojačem. Kako se pokazivač na stek umanjuje posle izvršenja poslednje instrukcije u poslednjoj iteraciji, prostim iskakanjem iz petlje u toku njenog izvršavanja pokazivač se neće umanjiti, što može da predstavlja ozbiljan problem u radu modula. Izvor ovakve greške je veoma teško naći i ispraviti problem, pošto prekidom petlje na pogrešan način problem može da se ispolji na sasvim drugom mestu u kodu. Da bi se sprečili ovi problemi koristi se instrukcija *enddo* koja u slučaju prekida petlje umanjuje pokazivač na DO stek.Primer:

```
    i0 = (32)
     a0 = 0
      uhalfword(a1) = (8)
      do(i0),>Loop                #<--| do(32)
         a0 - a1
         if (a==0) jmp >LoopBreak    #<--|     if (a0==8) break;
%Loop:     a0 = a0 + b0            #<--|    a0++;

    jmp  >LoopFinishedWithoutBreak    #<--| If loop is finished without break
                #^__| skip enddo because stack pointer is already decremented

%LoopBreak: enddo       #<--| Here loop is terminated before last iteraton
                        #^__| so decrement stack pointer

% LoopFinishedWithoutBreak:
```

#### 2.2.3.5.3 Uslovni i bezuslovni skok

Uslovni i bezuslovni skokovi se vrše korišćenjem instrukcija *if* i *jmp*. Format instrukcije za bezuslovni skok je:

- *jmp <adresa>,*

gde adresa može biti konstanta, simbol ili indeksni registar. U slučaju kada je adresa indeksni registar, vrednost indeksnog registra se može umanjiti i uvećati.

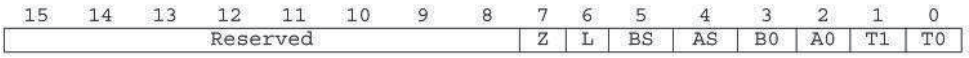
Uslovni skok će se izvršiti samo ako je ispunjen zadati uslov. Format instrukcije za uslovni skok je:

- *if (uslov) jmp <adresa>,*

gde je adresa simbol a uslov zavisi od stanja CCR registra (*Condition Code Register*). Struktura CCR registra je prikazana na slici 2.9, a opisi polja se nalaze u tabela 2.7. Spisak uslova uslovnog skoka je:

- a == 0
- a != 0
- a < 0
- a >= 0
- a <= 0
- a > 0
- z != 0
- z == 0
- b == 0
- b != 0
- b < 0
- b >= 0
- b <= 0
- b > 0
- limit (postavljen *limit bit* u *Modulo-Offset* registru)
- !limit (nije postavljen *limit bit* u *Modulo-Offset* registru)

Polja CCR registra se postavljaju izvršavanjem aritmetičkih operacija (Slika 2.9).




*Slika 2.9– Struktura CCR registra*

Tabela 2.7 – Polja CCR registra

| Biti | Ime polja | Opis |
|---|---|---|
| 15:8 | Rezervisano | Rezervisano |
| 7 | Z | Zero bit – postavlja se instrukcijama a manipulaciju bita |
| 6 | L | Limit bit – postavlja se kada se desi saturacija. Jednom kada se postavi mora se eksplicitno vratiti na nulu od strane programera |
| 5 | BS | Sign bit B grupe akumulatora – postavlja se na jedinicu ako je rezultat B akumulatora negativan |
| 4 | AS | Sign bit A grupe akumulatora – postavlja se na jedinicu ako je rezultat A akumulatora negativan |
| 3 | B0 | Zero bit B grupe akumulatora – postavlja se na jedinicu ako je rezultat B akumulatora nula |
| 2 | A0 | Zero bit A grupe akumulatora – postavlja se na jedinicu ako je rezultat A akumulatora nula |

| | | |
|---|---|---|
| 1:0 | T1, T0 | Statusni biti pomeraja kod SRS jedinice. T1 i T0 se postavljaju u zavisnosti od bita [63:59] akumulatora i vrednosti S1 i S0 mode registra. Primer:<br>Biti[63:59] T1 T0 Pomeranje<br>00000 or 11111 0 0 bez pomeranja<br>00001 11110 0 0 bez pomeranja<br>00010 11101 0 0 bez pomeranja<br>00011 11100 0 1 za jedno mesto<br>00100 11011 0 1 za jedno mesto<br>00101 11010 0 1 za jedno mesto<br>0011x 1100x 1 0 za dva mesta<br>01xxx 10xxx 1 0 za dva mesta<br>1 1 ne koristi se |

Primeri:

```
        # Example 1. Unconditional jump
    jmp >unconditionalLabel
        nop
        nop
%unconditionalLabel:

    # Example 2. Unconditional jump to adress in index registar
    i0 = (unconditionalLabel1)
    nop
    jmp (i0)
     nop
     nop
unconditionalLabel1:

    # Example 3. Conditional jump
    a0 = 0
    a0 & a0
    if (a==0) jmp >A0IsZero     #<--| Here A0 zero flag is set and we will
jump
        nop
        nop
%A0IsZero:

    uhalfword(a0) = (1)
    a0 & a0
    if (a==0) jmp >A0IsZero    #<--| Here A0 zero flag is not set and we will
not jump
        nop
        nop
%A0IsZero:
```

#### 2.2.3.5.4 Poziv funkcija

Pozivanje funkcija vrši se pomoću direktive *call* iza koje se navodi 16 bitna adresa u vidu simbola ili adrese u indeksnom registru. Kraj funkcije označava se direktivom

*ret* koja ujedno predstavlja i poslednju instrukciju funkcije. *Ret* direktiva će programski brojač postaviti na prvu sledeću instrukciju koja sledi nakon direktive *call*. Mesto na koje će se programski brojač postaviti nakon završetka pamti se na *call* steku.

Pored ove dve direktive, postoje i specijalani slučajevi: *callint* i *retcc*. Ove dve direktive imaju istu funkcionalnost kao *call* i *ret* – pored navedenih funkcionalnosti one isključuju i uključuju prekide. Za uključivanje i isključivanje prekida postoje i posebne direktive, *intdis* i *inten*. Primer:

```
# Example 1. Function call
      call X_S_SampleFunction
      nop             #<--| After function execution is finished this
                      #^__| will be next executed instruction

# Example 1. Function call
      callint X_S_SampleFunctionInterruptControll  #<--| Call functio and
                                                   #^__| disable interrupts
      nop             #<--| After function execution is finished this
                      #^__| will be next executed instruction
.....
X_S_SampleFunction:
      nop
SampleFunction_End:
      ret

X_S_SampleFunctionInterruptControll:
      nop
SampleFunctionInterruptControl_End:
      retcc             #<--| Return from funcion and enable interupts
```

#### 2.2.3.5.5 Nop i halt direktive

*Halt* direktiva zaustavlja izvršavanje i prebacuje DSP u *low-power* stanje. *Nop* direktiva predstavlja praznu instrukciju (*nop* – No operation).

### *2.2.3.6 Instrukcije toka podataka*

#### 2.2.3.6.1 Tok podataka u/iz memorije

Instrukcije za pomeranje podataka iz memorije u neki od registara/akumulatora i obrnuto, dati su u tabeli 2.8. Radi lakše čitljivosti tabele uvedene su sledeće konvencije:

- Acc – predstavlja bilo koji akumulator: *a0, a1, a2, a3, b9, b1, b2, b3*
- AnyReg – predstavlja bilo koji registar iz sledeće grupe: *x0-x3, y0-y3, a0-a3, a0h-a3h, a0l-a3l, b0-b3, b0h-b3h, b0l-b3l, i0-i11, nm0-nm11*

- DpRreg – predstavlja bilo koji registar iz sledeće grupe: *x0-x3, y0-y3, a0-a3, b0-b3* MsReg - predstavlja bilo koji registar iz sledeće grupe: *ccr, dbc_cmd, dbc_d1, dbc_d2, dbc_io, dbc_status, iic_addr, iic_mask, jsr_data, jsr_mode,jsr_ovf, jsr_unf, lp_data1, lp_data2, lst_data1, lst_data2, lst_mode, lst_ovf, lst_unf, mr, mr_jsr_ptr, mr_lst_ptr, mr_r, mr_s, mr_sr, page_p, page_x, page_y, pc, pc_bp, rand,*
- *rand_a,rand_b, rand_reset, rx_in, search_cnt, search_latch, stq_base*
- Addr – predstavlja bilo koji 16-bitnu adresu u vidu simbola, adrese ili indeksnog registra
- AddrL – predstavlja bilo koji 6-bitnu adresu u vidu simbola, adrese ili 16 – bitnu adresu indeksnog registra
- OP – odredišni ili izvorišni operand

Tabela 2.8 – Tok podataka u/iz memorije

| Instrukcija | Sintaksa | Polja CCR registra promenjena instrukcijom | Restrikcije |
|---|---|---|---|
| Prenos podataka iz memorijske zone X u registar | OP = xmem[Addr] | Nijedan | OP može biti bilo koji registar iz grupe AnyReg i MsReg |
| Prenos podataka iz registra u memorijsku zonu X | xmem[Addr] = OP | L, T0, T1[*1] | OP može biti bilo koji registar iz grupe AnyReg i MsReg |
| Prenos podataka iz memorijske zone Y u registar | OP = ymem[Addr] | Nijedan | OP može biti bilo koji registar iz grupe AnyReg i MsReg |
| Prenos podataka iz registra u memorijsku zonu Y | ymem[Addr] = OP | L, T0, T1[*1] | OP može biti bilo koji registar iz grupe AnyReg i MsReg |
| Prenos podataka iz memorijske zone P u registar | OP = pmem[Addr] | Nijedan | OP može biti bilo koji registar iz grupe AnyReg i MsReg |
| Prenos podataka iz registra u memorijsku zonu P | pmem[Addr] = OP | L, T0, T1[*1] | OP može biti bilo koji registar iz grupe AnyReg i MsReg |
| Prenos podataka iz perifernih uređaja u registar | OP = inp[Addr] | Nijedan | OP može biti bilo koji registar iz grupe AnyReg i MsReg |
| Prenos podataka iz registra u periferni uređaj | outp[Addr] = OP | L, T0, T1[*1] | OP može biti bilo koji registar iz grupe AnyReg i MsReg |

| | | | |
|---|---|---|---|
| Prenos podataka iz memorijske zone XY u registre | OP1, OP2 = xymem[AddrL] | Nijedan | OP1 i OP2 su registri iz grupe *DpReg* uz restrukcije da ne emeju pripadati istoj putanji podataka(X, Y) i da moraju imati isti indeks (npr, x0, y0) |
| Prenos podataka iz registara u memorijsku zonu XY | xymem[AddrL] = OP1, OP2 | L, T0, T1[*1] | OP1 i OP2 su registri iz grupe *DpReg* uz restrukcije da ne emeju pripadati istoj putanji podataka(X, Y) i da moraju imati isti indeks (npr, x0, y0) |
| Prenos podataka iz memorijske zone XY u akumulator | OP = xymem[AddrL] | Nijedan | OP = akumulator |
| Prenos podataka iz akumulatora u memorijsku zonu XY | xymem[AddrL] = OP | L, T0, T1[*1] | OP = akumulator |

[*1] – Polja se menjaju samo ukoliko je OP akumulator. Jednom kada se postave, moraju biti resetovana od strane korisnika. T0 i T1 mogu imati vrednosti 10, 01 ili 00

Primeri:

```
X_VX_SourceXMemVariable            .dw    0x12345678
X_VX_DestinationXMemVariable        .dw    0
X_VY_SourceYMemVariable            .dw    0x87654321
X_VY_DestinationYMemVariable        .dw    0

    #-------------------------------------------------------------
    # X data memory to register

    # Example 1. Using direct addressing
    a0 = xmem[0x800]          #<--| a0 will have value present
                              #^__| on address 0x800 in X memory

    # Example 2. Using symnol as address
    a0 = xmem[X_VX_SourceXMemVariable]    #<--| b0 will be 0x12345678

    # Example3. Using index register
    i0 = (X_VX_SourceXMemVariable)
    nop
    x0 = xmem[i0]                 #<--| x0 will be 0x12345678
    #-------------------------------------------------------------------



    #-------------------------------------------------------------------
    # register to X data memory
    uhalfword(a0) = (0xbaba)
```

```
# Example 1. Using direct addressing
xmem[0x800] = a0          #<--| address 0x800 in X memory
                          #^__| will have value present in a0

# Example 2. Using symbol  as address
xmem[X_VX_DestinationXMemVariable] = a0 #<--| X_VX_SourceXMemVariable
                                        #^__| will be 0xbaba

# Example3. Using index register
i0 = (X_VX_DestinationXMemVariable)
nop
xmem[i0] = a0             #<--| X_VX_SourceXMemVariable will be
                          #^__| 0xbaba
#-------------------------------------------------------------------


#-------------------------------------------------------------------
# Y data memory to register

# Example 1. Using direct addressing
a1 = ymem[0x800]          #<--| a1 will have value present
                          #^__| on address 0x800 in Y memory

# Example 2. Using symbol  as address
b1 = ymem[X_VY_SourceYMemVariable]    #<--| b10 will be 0x87654321

# Example3. Using index register
i0 = (X_VY_SourceYMemVariable)
nop
x1 = ymem[i0]                 #<--| x1 will be 0x12345678
#-------------------------------------------------------------------


#-------------------------------------------------------------------
# register to Y data memory
uhalfword(a0) = (0xdeda)

# Example 1. Using direct addressing
ymem[0x800] = a0          #<--| address 0x800 in Y memory
                          #^__| will have value present in a0

# Example 2. Using symbol  as address
ymem[X_VY_DestinationYMemVariable] = a0 #<--| X_VX_SourceXMemVariable
                                        #^__| will be 0xdeda

# Example3. Using index register
i0 = (X_VY_DestinationYMemVariable)
nop
ymem[i0] = a0            #<--| X_VX_SourceXMemVariable will be
                         #^__| 0xdeda
#-------------------------------------------------------------------
```

```
#-----------------------------------------------------------------------
# XY memory moves
i0 = (X_VL_SourceLMemVariable)
i1 = (X_VL_DestLMemVariable)

x0, y0 = xymem[i0]          #<--| x0 = 0x0000baba y0 = 0x0000deda
xymem[i1] = x0, y0          #<--| X_VL_DestLMemVariable has value
                            #^__| 0x0000baba.0000deda
a0 = xymem[i0]              #<--| a0 = 0x00.0000baba.0000deda

halfword(b0) = (-1)
xymem[i1] = b0          #<--| X_VL_DestLMemVariable has value
                        #^__| 0xffffffff.00000000
#-----------------------------------------------------------------------
```

#### 2.2.3.6.2 Učitavanje konstante ili vrednosti koju predstavlja simbol u registar

Asemblerski jezik definiše više načina učitavanja konstantnih vrednosti u registre, koji pokrivaju sve korisne slučajeve upisa 16-bitnih vrednosti. Tabela 2.9 daje pregled upisa konstante u 32-bitni registar, dok Tabela 2.10 opisuje upis konstanti u akumulatore.

Primeri:

```
#-------------------------------------------------------------------------------
# Immediate register loads
lo16(a0l) = (8)                 #<--| a0 = 0x00.00000000.00000008
fixed16(a0h) = (0xff00)         #<--| a0 = 0x00.ff000000.00000008
fixed16(b0) = (0x8000)          #<--| b0 = 0xff.80000000.80000000
uhalfword(y0) = (255)           #<--| y0 = 0x000000ff
halfword(x0) = (-255)           #<--| x0 = 0xffffff01
#-------------------------------------------------------------------------------
```

Tabela 2.9 – Upis konstante u 32-bitni registar

| | x0 ...x3<br>y0...y3 | | | |
|---|---|---|---|---|
| Instrukcija | 31 | 16 | 15 | 0 |
| fixed16(x0) | 16-bitni podatak | | nule | |
| ufixed16(x0) | 16-bitni podatak | | nule | |
| halfword(x0) | proširenje znaka | | 16-bitni podatak | |
| uhalfword(x0) | nule | | 16-bitni podatak | |
| lo16(x0) | bez promene | | 16-bitni podatak | |

Tabela 2.10 – Upis konstante u 32-bitni registar

| | a0..a3, b0..b3 | | | | |
|---|---|---|---|---|---|
| Instrukcija | a0g 71 64 | a0h 63 48 | 47 32 | a0l 31 16 | 15 0 |
| fixed16(a0) | proširenje znaka | 16-bitni pod. | nule | nule | nule |
| fixed16(a0h) | bez promene | 16-bitni pod. | nule | bez promene | bez promene |
| fixed16(a0l) | bez promene | bez promene | bez promene | 16-bitni pod. | nule |
| ufixed16(a0) | nule | 16-bitni pod. | nule | nule | nule |
| ufixed16(a0h) | bez promene | 16-bitni pod. | nule | bez promene | bez promene |
| ufixed16(a0l) | bez promene | bez promene | bez promene | 16-bitni pod. | nule |
| halfword(a0) | proširenje znaka | proš. znaka | 16-bitni pod. | nule | nule |
| halfword(a0h) | bez promene | proš. znaka | 16-bitni pod. | bez promene | bez promene |
| halfword(a0l) | bez promene | bez promene | bez promene | proš. znaka | 16-bitni pod. |
| uhalfword(a0) | nule | nule | 16-bitni pod. | nule | nule |
| uhalfword(a0h) | bez promene | nule | 16-bitni pod. | bez promene | bez promene |
| uhalfword(a0l) | bez promene | bez promene | bez promene | nule | 16-bitni pod. |
| lo16(a0) | bez promene | bez promene | 16-bitni pod. | bez promene | bez promene |
| lo16(a0h) | bez promene | bez promene | 16-bitni pod. | bez promene | bez promene |
| lo16(a0l) | bez promene | bez promene | bez promene | bez promene | 16-bitni pod. |

Kao odredišni operand kod svih instrukcija mogu da se pojave svi registri koji propadaju skupovima *DpReg i MsReg*.

#### 2.2.3.6.3 AnyReg instrukcija

Ova instrukcija vrši prenos podataka iz bilo kog registra u bilo koji registar. Takođe, može u paraleli da vrši dva prenosa. Sintaksa instrukcije je:

- *AnyReg*(odredište, izvor),
- *AnyReg*(odredište1, izvor1), (odredište2, izvor2),
- polja CCR registra promenjena instrukcijom su L i T1 i T0.

Primer:

```
AnyReg(a2, y0)     #<--| a2 = 0x00.000000ff.00000000
AnyReg(b3, a2), (a3, x0)     #<--| b3 = 0x00.000000ff.00000000,
                                   a3 = 0xff.ffffff01.00000000
```

#### 2.2.3.6.4 Instrukcije za manipulaciju na nivou bita

***BitTst*** instrukcija ispituje sadržaj registra u odnosu na 16-bitnu masku. Ukoliko su svi biti postavljeni na jedinicu unutar maske postavljeni na jedinicu i u registru, Z polje CCR registra će se postaviti na jedinicu, dok će u suprotnom slučaju biti postavljeno na nulu. Pseudo kod ove instrukcije je:

```
if ((reg AND mask) XOR mask) == 0x0000
       z = 1
else
       z = 0
```

dok je sintaksa:

*BitTst lo(registar), (16-bitna maska)*
*BitTst hi(registar), (16-bitna maska)*

Nakon završetka ove intrukcije sadržaj registra ostaje nepromenjen.

***BitSet*** instrukcija vrši operaciju *ILI* nad registrom i 16-bitnom maskom i rezultat smešta nazad u registar. Sintaksa instrukcije je:

*BitSet lo(registar), (16-bitna maska)*
*BitSet hi(registar), (16-bitna maska)*

***BitClr*** instrukcija vrši ispitivanje opisano u instrukciji *BitTst* i zatim vrši operaciju i nad specificiranim delom registra i nad negiranom 16-bitnom maskom i rezultat smešta nazad u registar. Sintaksa instrukcije je:

*BitClr lo(registar), (16-bitna maska)*
*BitClr hi(registar), (16-bitna maska)*

***BitChg*** instrukcija vrši ispitivanje opisano u instrukciji *BitTst* i zatim vrši operaciju ekskluzivno ILI nad specificiranim delom registra i nad 16-bitnom maskom i rezultat smešta nazad u registar. Sintaksa instrukcije je

*BitChg lo(registar), (16-bitna maska)*
*BitChg hi(registar), (16-bitna maska)*

Nijedna od instrukcija za manipulaciju na nivou bita ne može da se izvršava nad akumulatorima.

Primeri:

```
        BitTst lo(y0), (0x1)         #<--| y0 = 0x000000ff
        if (z==0) jmp >         #<--| is bit 0 of y0 set to one?
                nop             #<--| yes, so this instruction will be executed
%
        BitTst hi(y0), (0x1)         #<--| y0 = 0x000000ff
        if (z==0) jmp >         #<--| is bit 16 of y0 set to one?
            nop             #<--| no, so this instruction will not be executed
%
        BitSet hi(y0), (0x1)         #<--| y0 will be 0x000100ff

        BitClr hi(y0), (0x1)         #<--| y0 will be 0x000000ff

        BitChg lo(y0), (0x1003)     #<--| y0 will be 0x000010fc
```

#### 2.2.3.6.5 Tok podataka između registara

Poštujući navedenu podelu registara instrukcije toka podataka između registara se mogu podeliti u dve grupe:

- *DpReg* = *AnyReg* – primer ovog tipa instrukcije je y0 = b0g. Izvršenjem ove instrukcije ne menja se ni jedno polje CCR registra
- *AnyReg* = *DpReg* – primer ovog tipa instrukcije je nm0 = a2. Izvršenjem ove instrukcije menjaju se polja L, T0 i T2 CCR registra.

### *2.2.3.7 MAC instrukcija*

MAC predstavlja skraćenicu engleske reči *Multiply And Accumulate*. MAC instrukcija podrazumeva množenje vrednosti dva registra i sabiranje rezultata sa trećim registrom u jednoj instrukciji.

Tabela 2.11 – MAC Instrukcije

| Instrukcija | Sintaksa | Ppromenjena polja CCR registra | Restrikcije |
|---|---|---|---|
| Množi i/ili akumuliraj | Acc ?= ±Xn*Xm<br>Acc ?= ±Xn*(unsigned)Ym<br>Acc ?= ±Xm*Yn<br>Acc ?= ±Yn*Xm<br>Acc ?= ±Yn*Ym<br>Acc ?= ±(unsigned)Xn*(unsigned)Ym<br>Acc ?= ±Xn*(unsigned)Ym | Nijedno | Odredišni operand može da bude samo jedan od osam akumulatora. |
| Množenje sa jedinicom uz opciono akumuliranje | Accum ±= Xn<br>Accum ±= Yn<br>Accum = ±Xn*1<br>Accum = ±Yn*1 | Nijedno | Odredišni operand može da bude samo jedan od osam akumulatora. |

MAC instrukcije su karakteristične za digitalne signal procesore. Njena osnovna primena je u optimalnom izvršavanju operacije diskretne konvolucije, FFT, i srodnih algoritama obrade. Pseudo kod ove instrukcije je:

$$A = B + C*D$$

Jezgro CS48x ima dve MAC jedinice koje rade u aritmetici u nepokretnom zarezu. Činjenica da postoje dve MAC jedinice daje ovom jezgru mogućnost da obavlja dve MAC operacije u jednom taktu – paralelni MAC. Spisak MAC instrukcija dat je u tabela 2.11, dok će paralelne MAC instrukcije biti obrađene u poglavlju Tehnike optimizacije. Rezultat operacija označenih sa *1 je isti kao i kod druge grupe instrukcija koje su opisane u poglavlju *Tok podataka između registara* s tim što se kod instrukcija koje koriste MAC jedinicu za samu instrukciju koristi manji broj bita pa je moguće instrukcionu reč iskoristiti za još neke instrukcije.

Primeri:

```
#-------------------------------------------------------------------
# Real Multiply, Multiply and Accumulate
ufixed16(x0) = (0x4000)             #<--| x0 = 0.5
ufixed16(x3) = (0x2000)             #<--| x3 = 0.25
fixed16(y3)  = (0xc000)             #<--| y3 = -0.5
fixed16(y0)  = (0xe000)           #<--| y0 = -0.25
AnyReg(x1, y3), (y1, y0)

a1 = x0*x3                      #<--| a1 = 0x00.10000000.00000000 = 0.125

b3 = x3*(unsigned)y3        #<--| b3 = 0x00.30000000.00000000 = 0.375
                            #^  | unsigned value of y3 means that MSB bit
                            #|  | will not be treat as sign bit - so
                            #|  |0xc0000000 is 1.5 in fixed point number
                            #|  | representation (0xc0000000/2^31 =
                            #|__| 3221225472/2^31 = 1.5)

b3 += x0*x3                 #<--| b3 = 0.375 + 0.5*0.25 = 0.5 =
                            #^__|      0x00.40000000.00000000

a1 -= x0*x3                 #<--| a1 = 0.125 - 0.5*0.25 = 0 =
                            #^__|         0x00.00000000.00000000

b2 = x0*y0                  #<--| b0 = 0.5 * (-0.25) = -0.125 =
                            #^__|      0xff.f0000000.00000000

b0 = (unsigned)x1*(unsigned)y1  #<--| b0 = |-0.5] * |-0.25| = 1.5 * 1.75
                                #^__| = 2.625 = 0x01.50000000.00000000
#-------------------------------------------------------------------
```

```
#-------------------------------------------------------------------
# Multiply by One with Optional Accumulate
b3 = +x3            #<--| b3 = 1 * 0.25 = 0.25 = 0x00.20000000.00000000
b3 -= x0            #<--| b3 = 0.25 - 1*0.5 = -0.25 =
                    #^__|      0xff.e0000000.00000000
#-------------------------------------------------------------------
```

### *2.2.3.8 Operacije nad akumulatorima*

Operacije nad akumulatorima čini grupa aritmetičko-logičkih operacija koje su date u tabeli 2.12. Jezgro CS48x ima dve ALU jedinice, što daje ovom jezgru mogućnost da obavlja dve ALU operacije u jednom taktu. Paralelne ALU operacije u okviru jedne instrukcione reči biće obrađene u šestoj vežbi.

Tabela 2.12 – ALU Instrukcije

| Instrukcija | Sintaksa | Polja CCR registra promenjena instrukcijom | Restrikcije |
|---|---|---|---|
| Sabiranje uz opciono pomeranje ulevo za jedno mesto jednog operanda | Ap=An ± Am<br>Bp=Bn ± Bm<br>Ap=An ± Bm<br>Bp=Bn ± Am<br>Ap=(An * 2) ± Am<br>Bp=(Bn * 2) ± Bm<br>Ap=(An * 2) ± Bm<br>Bp=(Bn * 2) ± Am | A0, AS, B0, BS | Odredišni operand može da bude samo jedan od osam akumulatora. |
| Operacija za traženje maksimuma | if (Bn>Bm) An=Am<br>if (An>Am) Bn=Bm<br>if (Bn>Am) An=Bm<br>if (An>Bm) Bn=Am | A0, AS, B0, BS | Odredišni operand može da bude samo jedan od osam akumulatora. |
| Operacija za traženje minimuma | if (Bn<Bm) An=Am<br>if (An<Am) Bn=Bm<br>if (Bn<Am) An=Bm<br>if (An<Bm) Bn=Am | A0, AS, B0, BS | Odredišni operand može da bude samo jedan od osam akumulatora. |
| Operacija za traženje apsolutnog maksimuma | if (\|Bn\|>\|Bm\|) An=Am<br>if (\|An\|>\|Am\|) Bn=Bm<br>if (\|Bn\|>\|Am\|) An=Bm<br>if (\|An\|>\|Bm\|) Bn=Am | A0, AS, B0, BS | Odredišni operand može da bude samo jedan od osam akumulatora. |
| Operacija za traženje apsolutnog minimuma | if (\|Bn\|<\|Bm\|) An=Am<br>if (\|An\|<\|Am\|) Bn=Bm<br>if (\|Bn\|<\|Am\|) An=Bm<br>if (\|An\|<\|Bm\|) Bn=Am | A0, AS, B0, BS | Odredišni operand može da bude samo jedan od osam akumulatora. |
| Prebacivanje podataka iz akumulatora u akumulator koristeći ALU[*1] | An =+ Am<br>Bn =+ Bm<br>An =+ Bm<br>Bn =+ Am | Nijedan | Odredišni operand može da bude samo jedan od osam akumulatora. |

| | | | |
|---|---|---|---|
| Komplement jedinice svih 72 bita akumulatora | Accum =~ Accum;<br>An =~ Am<br>Bn =~ Bm<br>An =~ Bm<br>Bn =~ Am | A0, AS, B0, BS | Odredišni operand može da bude samo jedan od osam akumulatora. |
| Negiranje vrednosti akumulatora | Accum =- Accum;<br>An =- Am<br>Bn =- Bm<br>An =- Bm<br>Bn =- Am | A0, AS, B0, BS | Odredišni operand može da bude samo jedan od osam akumulatora. |
| Apsolutna vrednost akumulatora | Accum = \|Accum\|<br>An = \|Am\|<br>Bn = \|Bm\|<br>An = \|Bm\|<br>Bn = \|Am\| | A0, AS, B0, BS | Odredišni operand može da bude samo jedan od osam akumulatora. |
| "ILI" | An = An \| Am<br>Bn = Bn \| Bm<br>An = An \| Bm<br>Bn = Bn \| Am | A0, AS, B0, BS | Odredišni operand može da bude samo jedan od osam akumulatora. |
| "Ekskluzivno ILI" ("XOR") | a0 = a0 ^ a3<br>b3 = b3 ^ b3<br>a1 = a1 ^ b2<br>b2 = b2 ^ a1 | A0, AS, B0, BS | Odredišni operand može da bude samo jedan od osam akumulatora. |
| "I" | An = An & Am<br>Bn = Bn & Bm<br>An = An & Bm<br>Bn = Bn & Am | A0, AS, B0, BS | Odredišni operand može da bude samo jedan od osam akumulatora. |
| Postavljanje vrednosti akumulatora na nulu | Accum = 0 | A0, AS, B0, BS | Odredišni operand može da bude samo jedan od osam akumulatora. |
| Pomeranje ulevo za jedno mesto | An = An << 1<br>Bn = Bn << 1 | A0, AS, B0, BS | Odredišni operand može da bude samo jedan od osam akumulatora. |
| Pomeranje ulevo za četiri mesta | An = An << 4<br>Bn = Bn << 4 | A0, AS, B0, BS | Odredišni operand može da bude samo jedan od osam akumulatora. |
| Pomeranje ulevo za osam mesta | An = An << 8<br>Bn = Bn << 8 | A0, AS, B0, BS | Odredišni operand može da bude samo jedan od osam akumulatora. |
| Pomeranje udesno za jedno mesto | An = An >> 1<br>Bn = Bn >> 1 | A0, AS, B0, BS | Odredišni operand može da bude samo jedan od osam akumulatora. |

| | | | |
|---|---|---|---|
| Testiranje vrednosti akumulatora (koristi se kod uslovnog skoka) | An & Am<br>Bn & Bm<br>An & Bm<br>Bn & Am | A0, AS, B0, BS | Oba operanda mogu da budu samo akumulatori. |
| Upoređivanje vrednosti akumulatora (koristi se kod uslovnog skoka) | An - Am<br>Bn - Bm<br>An - Bm<br>Bn - Am | A0, AS, B0, BS | Oba operanda mogu da budu samo akumulatori. |
| Upoređivanje apsolutnih vrednosti akumulatora (koristi se kod uslovnog skoka) | \|An\| - \|Am\|<br>\|Bn\| - \|Bm\|<br>\|An\| - \|Bm\|<br>\|Bn\| - \|Am\| | A0, AS, B0, BS | Oba operanda mogu da budu samo akumulatori. |

Primeri:

```
#-------------------------------------------------------------------
# Add with Optional Shift Left by one position
uhalfword(a2) = (0x5)                #<--| a2 = 0x00.00000005.00000000
uhalfword(a1) = (0x2)                #<--| a1 = 0x00.00000002.00000000

a3 = a2-a1                     #<--| a3 = 0x00.00000003.00000000
a0 = (a2*2)-a1                 #<--| a0 = 0x00.00000008.00000000
#-------------------------------------------------------------------

#-------------------------------------------------------------------
# Conditional Operation - Maximum
uhalfword(b2) = (0x3)                #<--| b2 = 0x00.00000003.00000000
uhalfword(b1) = (0x4)                #<--| b1 = 0x00.00000004.00000000

if (a1>a2) b1=b2             #<--| b1 remains unchanged

if (b1>b2) a1=a2             #<--| a1 will become 0x00.00000005.00000000
#-------------------------------------------------------------------

#-------------------------------------------------------------------
# Conditional Operation - Minimum
uhalfword(a1) = (0x2)            #<--| a1 = 0x00.00000002.00000000

if (b1<b2) a1=a2             #<--| a1 remains unchanged
if (a1<a2) b1=b2             #<--| b1 will become 0x00.00000003.00000000
#-------------------------------------------------------------------

#-------------------------------------------------------------------
# Bitwise Accumulator Move
```

```
    halfword(b3) = (-5)                 #<--|
    ufixed16(b3l) = (0x8000)            #^__| b3 = 0xff.fffffffb.80000000

    a0 =+ b3                      #<--| a0 = 0xff.fffffffb.80000000
    a1 = b3                       #<--| a1 = 0xff.fffffffb.00000000
    #-------------------------------------------------------------------

    #-------------------------------------------------------------------
    # Bitwise Complement
    a2 =~ b3                      #<--| a2 = 0x00.00000004.7fffffff
    #-------------------------------------------------------------------

    #-------------------------------------------------------------------
    # AccumNegative Accumulator Move
    a0 =- b3                      #<--| a0 = 0x00.00000004.80000000
    #-------------------------------------------------------------------

    #-------------------------------------------------------------------
    # Absolute Value Accumulator Move
    b0 = |b3|                      #<--| b0 = 0x00.00000004.80000000
    #-------------------------------------------------------------------

    #-------------------------------------------------------------------
    # Bitwise OR, XOR, AND, ZERO
    uhalfword(a1) = (0x1200)
    uhalfword(b1) = (0x34)
    a0 = 0                         #<--| Zero out a0 (0x00.00000000.00000000)
    a1 = a1 | b1                   #<--| a0 = 0x00.00001234.00000000

    uhalfword(a0) = (0x1035)
    a0 = a0 ^ a1                     #<--| a0 = 0x00.00000201.00000000

    uhalfword(a1) = (0x1)
    a0 = a0 & a1                     #<--| a0 = 0x00.00000001.00000000
    #-------------------------------------------------------------------

    #-------------------------------------------------------------------
    # Shift Examples
    ufixed16(a0) = (0x0009)          #<--| a0 = 0x00.00090000.00000000

    a0 = a0 >> 1                     #<--| a0 = 0x00.00048000.00000000

    a0 = a0 << 1                     #<--| a0 = 0x00.00090000.00000000

    a0 = a0 << 4                     #<--| a0 = 0x00.00900000.00000000

    a0 = a0 << 8                     #<--| a0 = 0x00.90000000.00000000
        #   | Note that when MSB bit of high acc part gets set to one as
        #   | a result of right shift sign of number will not be changed
#-------------------------------------------------------------------
```

## 2.3 Zadaci za samostalnu izradu

### 2.3.1 Zadatak 1: Primeri upotrebe asemblerskog jezika na procesoru CS48x

U okviru ovog zadatka, na priloženom primeru, upoznaje se sa praktičnom primenom opisanih asemblerskih instrukcija:

1. Uvući priloženi projekat *asmLekcije* u radno okruženje.
2. Pokrenuti priloženi projekat u režimu kontrolisanog izvršavanja (obeležite opciju *Start debugging during boot* i pritisnite dugme *Slave Boot*).
3. Izvršavati kod korak po korak. Potrebno je proći kroz sedam datih funkcija.
    a. X_S_Introduction – Uvodna lekcija.
    b. X_S_SRSUnit – Primer rada jedinice za logički pomeraj, zaokruživanje i saturaciju.
    c. X_S_AddressGenerationUnit – Primer rada jedinice za generisanje adresa.
    d. X_S_ProgramFlow – Primer korišćenja instrukcija za kontrolu toka izvršenja programa.
    e. X_S_MemoryMoves – Primer čitanja i pisanja u memoriju.
    f. X_S_MACInstructions – Primer korišćenja MAC jedinice.
    g. X_S_ALUUnitInstructions – Primer korišćenja jedinice za aritmetičko logičke operacije. Za svaku operaciju je dat komentar.
4. Koristeći mogućnosti koje nudi razvojno okruženje (*RegisterView*, *MemoryView*, *ExpressionView*...) proveriti rezultate operacija.

### 2.3.2 Zadatak 2: Primena upotrebe asemblerskog jezika na procesoru CS48x

U ovom zadatku je potrebno definisati dva niza jednake dužine u različitim memorijskim zonama, inicijalizovati date nizove i izračunati aritmetičku sredinu odgovarajućih parova elemenata.

1. U okviru priloženog projekta otvoriti datoteku *simpleTasks.a*
2. Zauzeti dva niza veličine 128 memorijskih lokacija i to
    a. *X_BX_Buffer1* u memorijskoj zoni X,
    b. *X_BY_Buffer2* u memorijskoj zoni Y,
    c. voditi računa da adresa prvog niza bude poravnata na 128 kako bi se omogućilo modulo adresiranje.
3. Definisati funkciju *X_S_init* koja vrši inicijalizaciju nizova.
4. U okviru *X_S_init* popuniti *X_BX_Buffer1* brojevima od 0 do 127.

5. U okviru *X_S_init* popuniti *X_BY_Buffer2* upisivanjem svakog trećeg elementa prvog niza, pri tom indeksni registar za čitanje elemenata prvog niza uvećavati po modulu 128. Za uvećavanje indeksnog registra sa korakom 3 koristiti *n* deo odgovarajućeg *nm* registra. Inicijalizaciju izvršiti prateći naredne korake:

```
X_BY_Buffer2[i] = X_BX_Buffer1[3*i]%128
```

   a. postaviti vrednost jednog indeksnog registra na *X_BX_Buffer1,*
   b. postaviti vrednost drugog indeksnog registra na *X_BY_Buffer2,*
   c. postaviti vrednost *nm* registra koji odgovara prvom indeksnom registru tako da se vrši adresiranje po modulu 128 i da je vrednost *n* dela registra 3.
   d. U okviru hardverske petlje koja traje 128 iteracija:
      i. učitati vrednosti koristeći indeksno adresiranje sa prvim indeksnim registrom u akumulator,
      ii. upisati vrednost akumulatora u memoriju na adresu koja se nalazi u drugom indeksnom registru,
      iii. uvećati oba indeksna registra.
   e. Vratiti vrednost nm registra na 0.
6. Napisati funkciju *X_S_mean* koja vrši računanje srednje vrednosti parova elemenata oba niza Rezultat smestiti u X_BX_Buffer1. Za deljenje sa dva iskoristiti SRS jedinicu.

   X_BX_Buffer1[i] = (X_BX_Buffer1[i] + X_BX_Buffer2[i])/2
7. Funkciju realizovati prateći sledeće korake:
   a. Postaviti vrednost jednog indeksnog registra na *X_BX_Buffer1.*
   b. Postaviti vrednost drugog indeksnog registra na *X_BY_Buffer2.*
   c. Postaviti odgovarajuću vrednost SRS registra.
   d. U okviru hardverske petlje koja traje 128 iteracija:
      i. učitati vrednosti koristeći indeksno adresiranje sa prvim indeksnim registrom u akumulator,
      ii. učitati vrednosti koristeći indeksno adresiranje sa drugim indeksnim registrom u drugi akumulator,
      iii. sabrati vrednosti koje se nalaze u akumulatorima i smestiti rezultat u akumulator 3,
      iv. upisati vrednost akumulatora 3 u memoriju koristeći indeksno adresiranje sa prvim indeksnim registrom,
      v. uvećati indeksne registre za 1.
   e. Vratiti vrednost SRS registra na nulu.
8. Unutar funkcije *X_S_simpleTask* izvršiti poziv funkcija *X_S_init* i *X_S_mean* redom.

### 2.3.3 Zadatak 3: Medijana

U okviru ovog zadatka potrebno je napisati funkciju za izračunavanje vrednosti medijane niza. Funkcija kao ulazni parametar očekuje adresu niza u registru *i0* i dužinu niza u registru i1. Podrazumevano je da niz sadrži paran broj elemenata. Računanje vrednosti medijane se vrši na u dva koraka:

- uređivanje niza tako da elementi budu poređani po veličini,
- uzimanje aritmetičke sredine dva središnja elementa X(N/2-1) i X(N/2).

Za uređivanje niza koristiti algoritam za uređivanje sa selekcijom:

```
for i = 1:n,
    k = i
    for j = i+1:n
        if a[j] < a[k]
            k = j // a[k]  je najmanji elemenat unutar
a[i..n]
    end
    swap a[i,k] //zameni mesta a[i] i a[k]
end
```

1. Unutar datoteke *simpleTasks.a* definisati funkciju *X_S_median.*
2. Implementirati dati pseudo kod za sortiranje niza. Podrazumevati da se početna adresa niza (a) nalazi u i0 a veličina niza (n) u i1.
3. Izračunati aritmetičku sredinu dva središnja elementa.
4. Rezultat smestiti u registar a0. Funkcija koja poziva funkciju za računanje medijane očekuje povratnu vrednost u ovom registru.