VEŽBA 7 – Radno okruženje digitalnih signal procesora

7.1 Uvod

Osnovni zadatak operativnih sistema za rad u realnom vremenu (eng. *Real Time Operating System* - RTOS) je da se ulazni događaji moraju uneti i obraditi sistemom unutar unapred definisanog intervala vremena. Treba naglasiti da upotreba RTOS-a ne garantuje sama po sebi da će sistem zadovoljiti zadata ograničenja. Ovakvi operativni sistemi su zapravo alati koji projektantima sistema omogućavaju kontrukciju sistema za rad u realnom vremenu. Operativni sistemi za digitalne signal procesore nekada nastaju kao modifikacija operativnih sistema za mikroprocesore, a nekad se namenski pišu za DSP platformu.

U prethodnim vežbama je opisan razvoj DSP aplikacija, sa naglaskom na funkcije obrade. Nakon modifikacija referentog koda, profilisanja, optimizacija i verifikacija ispravnosti rada, ove funkcije je potrebno integrisati sa sistemskim softverom digitalnog signal procesora, odnosno, potrebno je sinhronizovati funkciju obrade sa događajima u realnom vremenu (kao što je prijem podataka). Ovaj sistemski softver se često zove i radno okruženje ili radni okvir (eng. *Framework*) digitalnih signal procesora.

Radni okvir obezbeđuje raspoređivanje zadataka (funkcija obrade), rukovanje vremenski kritičnim zadacima, upravljanje tokom podataka između različitih zadataka, komunikaciju između programskih celina unutar samog procesora, kao i komunikaciju sa spoljnim uređajima.

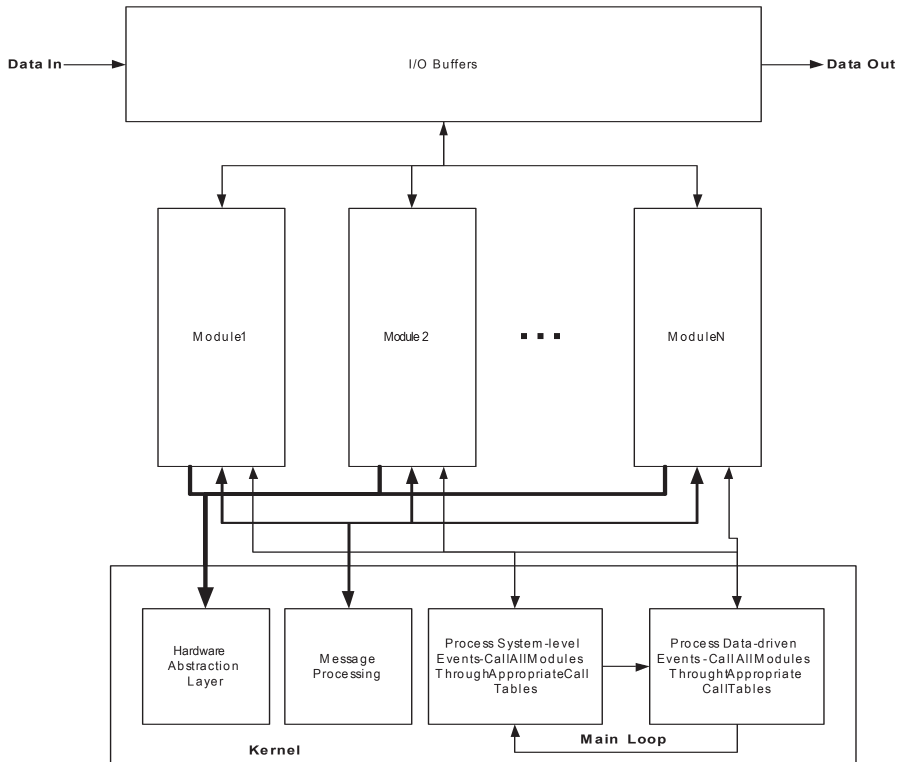
U okviru ove vežbe prikazaće se struktura radnog okvira procesora CS48x i osnovni koncepti operativnog sistema ovog procesora. Obradiće se:

- na koji način je organizovana jedna DSP aplikacija,
- kako izgleda sprega između operativnih sistema i modula,
- na koji način je omogućeno korisniku ili spoljnom uređaju da komunicira sa modulima u toku njihovog izvršenja,
- na koji način radni okvir upravlja prenosom podataka od ulazne sprege, kroz sve nivoe obrade do izlazne sprege,
- na koji način mora biti organizovan programski modul da bi se mogao integrisati u radni okvir.

Pokazaće se kako se realizuje aplikacija ili kako se prilagođava postojeće rešenja izvršenju u realnom vremenu na ciljnom DSP uređaju.

7.2 Radni okvir procesora CS48x

Cirrus Logic radni okvir (*framework*) (slika 7.1) predstavlja sistemski softver procesora koja skraćuje vreme za razvoj aplikacije, uvodeći neke od ideja i metodologija iz objektno orijentisanog programiranja u svet asemblerskog koda.



Slika 7.1 – Osnovna struktura radnog okvira procesora CS48x

Jezgro radnog okvira se sastoji od jednostavnog operativnog sistema (OS), čija je glavna uloga da bude raspoređivač za određen broj procesnih entiteta (modula). Uslovno rečeno, OS predstavlja monitorsku petlju koja poziva rutine odgovarajućih modula po unapred definisanom redosledu.

Da bi moduli mogli da reaguju na događaje u sistemu, za svaki tip događaja (*event*) definisane su odgovarajuće ulazne tačke modula (*entry points*). Osnovne ulazne tačke povezane su sa događajima vezanim za sistemski nivo, kao što je inicijalizacija sistema u početno stanje *(reset)* ili obrada zahteva za dinamičku dodelu memorije. Pored toga, budući da je radni okvir namenjen sistemima koji

svoje stanje menjaju na osnovu ulaznog toka podataka *(data driven systems)*, takođe su definisane i ulazne tačke za događaje pokrenute ulaznim tokom podataka, kao što su, promene stanja modula uzrokovane dotokom određene količine ulaznih podataka, promene u sistemu uzrokovane menjanjem parametara od strane sistemskog mikrokontrolera, odnosno korisnika.

Pored dela za rukovanje događajima, važan deo radnog okvira čini sistemski ulazno/izlazni memorijski niz koji služi za smeštanje audio podataka koji ulaze u sistem, nad kojima moduli vrše obradu i koji potom izlaze iz sistema (*eng. Input-Output Buffers)*.

7.2.1 Moduli

Osnovnu komponentu sistemske programske podrške CS48x procesora čine moduli. Moduli su definisani kao objekti sastavljeni od rutina i podataka, u skladu sa radnim okruženjem. Svi moduli moraju da ispunjavaju sledeće uslove:

1. da su nezavisni od drugih modula,
2. obavljaju obradu/generisanje PCM podataka u blokovima od bar po 16 odbiraka (=***1 brick***),
3. da budu usaglašeni sa standardnim načinom pozivanja modula od strane OS-a,
4. da koriste standardizovane načine komunikacije između modula,
5. da razmenjuju podatke sa OS-om o promenama u sistemu (npr. promena konfiguracije kanala, učestanosti odabiranja, itd.) na standardizovan način.

Svaki modul ima svoj jedinstveni sprežni podsistem (*Module Interface* – **MIF**), kojim je modul povezan sa OS-om. Njega čini MIF tabela koja sadrži pokazivače na tabele sa ostalim sprežnim informacijama. Dve najvažnije tabele su MCT tabela (*Module Call Table*), i MCV tabela (*Module Control Vector*). Sa OS strane, sprega ka modulima se sastoji od ODT tabele (*Overlay Definition Table*) koja sadrži pokazivače na MIF tabele svih učitanih modula, slika 09.

MIF tabela ima sledeću formu (<module name> predstavlja identifikator, tj. ime modula):

```
X_BY_MIF_<module name>
            .dw X_BY_<module name>MCV
            .dw X_BY_<module name>MCT
            ...
```

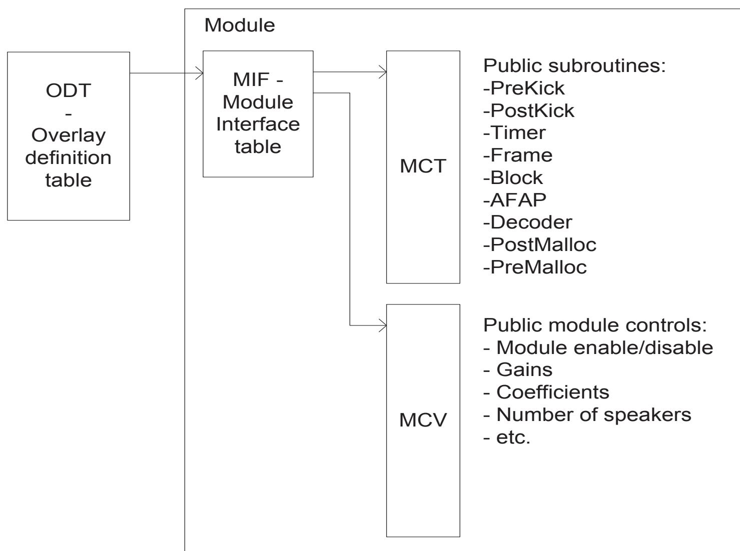
MCV tabela predstavlja niz javno dostupnih konfiguracionih parametara datog modula, i ona omogućava konfigurisanje modula od strane glavnog kontrolera uređaja (*host*). Struktura ove tabele nema neku unapred definisanu formu i programeru je prepušteno da formira njen sadržaj i strukturu.

```
X_BY_MCV_<module name>
            .dw X_VX_<control variable name>
            .dw X_VX_<control variable name>
            .dw X_VX_<control variable name>
            .dw X_VX_<control variable name>
            ...
```

MCT tabela je niz od devet elemenata – pokazivača na osnovne javne (*public*) rutine. Redosled elemenata u tabeli je unapred definisan, a ukoliko neka od rutina nije definisana za dati modul, na mestu njenog pokazivača se nalazi nula. Ove rutine poziva OS kao odgovor na pojavu odgovarajućih događaja u sistemu (*eng. event handles*).

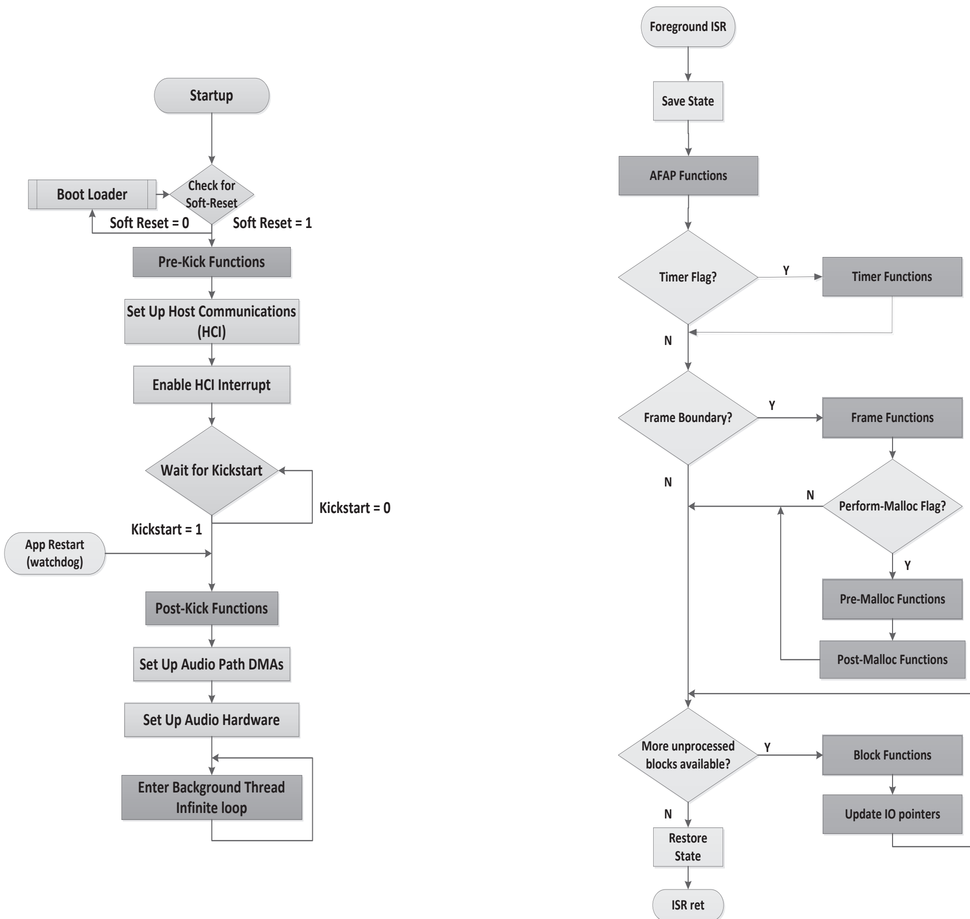
```
X_BY_MCT_<module name>
            .dw X_S_<module name>PreKickstart
            .dw X_S_<module name>PostKickstart
            .dw X_S_<module name>Timer
            .dw X_S_<module name>Frame
            .dw X_S_<module name>Block
            .dw X_S_<module name>AFAP
            .dw X_S_<module name>Background
            .dw X_S_<module name>PostMalloc
            .dw X_S_<module name>PreMalloc
```

Rutine koje se nalaze u MCT tabeli su jedine funkcije modula kojima OS pristupa. Svaka od njih ima svoju specifičnu namenu i njihovim pozivanjem od strane OS-a, modul odgovara na događaje u sistemu.

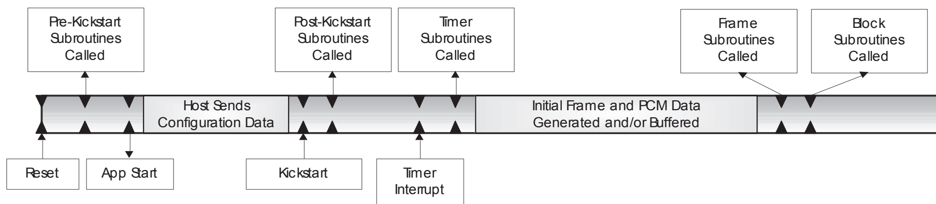


Slika 7.2 – Blok dijagram sprege modula sa operativnim sistemom

Treba napomenuti da su sve rutine osim *Background* **neprekidive**. One imaju isti prioritet, pa se izvršavanje jedne rutine ne može započeti pre završetka prethodno započete rutine. Izuzetak je jedino *Background* rutina koja je **prekidiva**. Ona je najnižeg prioriteta i izvršava se kao pozadinski proces. Njeno izvršavanje može da bude prekidano drugim rutinama višeg prioriteta (npr. *Timer*, *Frame*, *Block* ili *AFAP* rutinama) pri čemu se OS brine o očuvanju zatečenog stanja rutine – konteksta. Na slici 7.3 prikazan je pojednostavljeni algoritam rada OS-a, odnosno redosled pozivanja rutina koje su definisane unutar MCT tabele svakog modula, dok je na 7.4 prikazan primer redosleda izvršavanja rutina nakon resetovanja sistema.



Slika 7.3 - Pojednostavljeni algoritam rada OS-a



Slika 7.4 - Redosled pozvanja rutina

U nastavku je dat opis pojedinačnih rutina modula.

7.2.1.1 Pre-Kickstart rutina

Ovu rutinu OS poziva samo nakon prijema inicijalizacione poruke (*reset*) i pre uspostavljanja komunikacije sa sistemskim kontrolerom, slika 7.4. Ona omogućava inicijalizaciju modula, prvenstveno elemenata MCV tabele na njihove podrazumevane vrednosti. Nakon što su izvršene *Pre-Kickstart* rutine svih učitanih modula, OS prelazi u stanje čekanja na *kickstart* poruku od kontrolera. Po prijemu ove poruke, nastavlja se sa pozivima *Post-Kickstart* rutina.

7.2.1.2 Post-Kickstart rutina

Poziva se nakon što je uspostavljena komunikacija sa kontrolerom, slika 7.4. Drugim rečima ona omogućava obradu konfiguracionih podataka prosleđenih modulu od strane kontrolera. Nakon što su izvršene *Post-Kickstart* rutine svih modula, završena je inicijalizaciona faza i OS prelazi na izvršavanje normalnog ciklusa izvršavanja modula.

7.2.1.3 Pre-Malloc rutina

Prvi put se poziva nakon završetka inicijalizacionih rutina ukoliko je bilo koji modul u sistemu tokom inicijalizacije od OS-a zatražio inicijalizaciju dinamički dodeljene memorije. Takođe, ona može biti inicirana slanjem zahteva za reinicijalizaciju OS-u. Po prijemu zahteva, OS poziva *Pre-Malloc* rutine aktivnih modula, unutar kojih moduli prosleđuju zahteve za dodelu memorije OS-u. Pri tome se navodi memorijska zona (X, Y ili L) i veličina zahtevane memorije, kao i pokazivač koji treba da se inicijalizuje.

7.2.1.4 Post-Malloc rutina

Poziva se nakon završetka zauzimanja dinamičke memorije. Ova rutina omogućava inicijalizacija dinamički dodeljene memorije modula. Pre/Post-Malloc rutine se po potrebi izvršavaju pre prvog poziva *Block* i *Frame* rutina, slika 7.4.

7.2.1.5 Timer rutina

Poziva se periodično na svakih N milisekundi (podrazumevano je 1ms) kao odgovor na prekid (*interrupt*) generisan od strane brojača realnog vremena. Prvi put se poziva odmah nakon *Post-Kickstart* rutine i pre poziva *Frame* rutine, slika 7.4. Ona može da se koristi za proveru ulaznih konfiguracionih podataka modula, npr. da nadgleda promenu parametara koje postavlja korisnik (host).

7.2.1.6 Block i Frame rutine

Izvršavanje ***Block*** i ***Frame*** rutine je upravljano ulaznim tokom podataka pri čemu se *block* rutina izvršava pri prijemu svakih 16 PCM odbiraka u U/I nizu, dok se *frame* rutina izvršava na svakih N blokova. Ovaj broj blokova zavisi od dekoderskog modula u sistemu, odnosno od njegove jedinice dekodovanja (*decoding frame*). Na primer za AC3 dekoder to je 1536 odbiraka, DTS dekoder to može biti 512 i tako dalje. U slučaju ne-kompresovanog ulaznog toka (PCM) perioda *frame* rutine je usvojena da bude 256 odbiraka.

7.2.1.7 AFAP rutina

AFAP je skraćenica od "*As Fast As Possible*". Ova rutina se poziva kada god se desi neki događaj u sistemu, naravno, uz uslov da ne prekida druge rutine istog prioriteta.

AFAP, *Timer*, *Frame*, i *Block* rutine čine takozvanu *Foreground thread* (nit).

7.2.1.8 Background rutina

Background rutina se izvršava periodično u pozadinskoj niti (*Background thread*), koja ima niži prioritet od *Foreground* niti.

U slučaju dekodera, unutar ove rutine se preuzimaju podaci iz ulaznog komprimovanog toka (*Input FIFO*), obavlja se dekodovanje pri čemu se dekodovani PCM odbirci privremeno smeštaju u lokalne memorijske nizove, odakle se kasnije korišćenjem AFAP rutine kopiraju u sistemski ulazno/izlazni memorijski niz.

Kod algoritama završne PCM obrade postoje dva pristupa korišćenju *Background* rutine. Jedan pristup jeste da se u *Block* rutini vrši obrada nad blokom odbiraka, dok se u Background rutini vrši osvežavanje vrednosti lokalne MCV tabele. Drugi pristup podrazumeva upotrebu dvostrukog skladištenja. U *Block* rutini se kopira novi, neobrađeni blok od 16 PCM odbiraka (*brick*) iz ulazno-izlazne sprežne memorije u lokalni memorijski niz, nad kojim se potom u *Background* rutini vrši obrada. U *block* rutini se istovremeno sa čuvanjem ulaznih odbiraka vrši kopiranje prethodno obrađenih (izlaznih) odbiraka u ulazno-izlaznu sprežnu memoriju. Ovakav pristup omogućava da ukoliko obrada ne stigne da se izvrši u predviđenom vremenskom intervalu, ta greška utiče samo na trenutni blok odbiraka.

7.2.2 Kontrolne promenljive radnog okvira

Radni okvir omogućava čitanje vrednosti i podešavanje određenih parametara od strane aplikacije u toku izvršenja. Ovi parametri se odnose na sam radni okvir i omogućavaju da se sama aplikacija parametrizuje po njima. Neke od kontrolnih promenljivih su:

- ___X_BY_IOBUFFER_PTRS – pokazivač na tabelu sa pokazivačima na aktivne ulazno izlazne memorijske nizove.
- ___X_VX_PPM_INPUT_CHANNELS – maska ulaznih kanala. Jedan bit odgovara jednom kanalu. Sadrži vrednost 1 za svaki kanal koji je aktivan na ulazu u sistem i 0 za sve ostale. Bit najniže važnosti odgovara nultom kanalu.
- ___X_VX_PPM_OUTPUT_CHANNELS – maska aktivnih kanala na izlazu iz čitavog sistema.
- ___X_VX_PPM_VALID_CHANNELS – maska aktivnih kanala na izlazu iz prethodnog modula u lancu izvršenja. Ukoliko neki modul ima drugačije aktivne kanale na izlazu u odnosu na aktivne kanale na ulazu (npr. mikser 2 na 4 kanala), neophodno je da podesi vrednost ove promenljive, kako bi naredni modul u nizu imao informaciju o tome koji kanali su trenutno aktivni.
- ___X_VX_PPM_SAMPLERATE – frekvencija odabiranja signala.
- ___X_VY_BLOCK_SIZE (u nekim verzijama ___X_VY_BRICK_SIZE) – broj odbiraka po kanalu u okviru jednog bloka obrade.

7.2.3 Nivoi programske podrške

Sistemska programska podrška CS48L20 procesora, u odnosu na vrstu aplikacije, posmatra se u okviru četiri aplikaciona nivoa. Nivoi su definisani modulima koje sadrže i njihovim redosledom izvršavanja. Četiri osnovna aplikaciona nivoa su:

- OS ili nivo 0 (Operativni sistem) – programska podrška niskog nivoa. Obavlja poslove vezane za komunikaciju sa glavnim kontrolerom uređaja (*host*), rukovanje prekidima, pribavljanje podataka iz eksterne memorije, upravljanje i pozivanje modula, obrada grešaka, itd.
- Dekoderski nivo ili nivo 1 – u okviru ove programske podrške realizuju se moduli koji na ulazu primaju kompresovani (kodirani) bitski tok iz ulaznog FIFO memorijskog niza (npr. *AC-3*, *DTS*, *mp3* itd.), a dekodovane PCM odbirke upisuju u sistemski U/I niz.
- MPM (*Mid-Processor Module*) ili nivo 2 – nivo među-obrade – je namenjen modulima koji obrađuju PCM podatke koje preuzima iz U/I niza. Obrađene PCM podatke moduli vraćaju nazad u U/I niz. Ovo su obično moduli koji vrše promenu broja izlaznih kanala, tj. smanjenje broja izlaznih kanala (*eng.*

downmix) ili povećanje broja izlaznih kanala (eng. *upmix*). Primeri modula koji spadaju u ovaj nivo obrade su *Dolby ProLogic* i *DTS NEO:6*.
- PPM (*Post-Processor Module*) ili nivo 3 – nivo završne obrade – ovi moduli vrše završnu obradu nad podacima koji su rezultat nižih nivoa obrade. Moduli koji spadaju u ovaj nivo su *Bass Management*, *Audio Manager*, *Tone Control*, *EQ, Delay*..

Za svaki nivo je statički rezervisana jedinstvena zona (segment) radne memorije. Time je omogućeno nezavisno učitavanje modula u programsku memoriju, a time i dobijanje željene kombinacije različitih tipova obrade. Na ovaj način se smanjuje veličina potrebne memorije, jer se čuvaju samo pojedinačni moduli, umesto svih predviđenih kombinacija dekodera, MPM i PPM modula. Tako na primer, ako je potrebna promena dekodera usled promene tipa komprimovanog bitskog toka na ulazu sistema, nije neophodno ponovo učitavanje OS-a, ili bilo kog drugog modula u nivoima među i završne obrade. Recimo, za dekodovanje mp3 ulaznog vektora potrebna je sledeća kombinacija modula: OS (nivo 0), mp3 modul (nivou 1) i *Bass Management* modul (nivo 2). Ako se na ulasku u sistem pojavi bitski tok u AAC formatu, učitava se samo AAC dekoderski modul umesto mp3 modula, bez izmena nivoa 0 i 2.

Nivoi su definisani modulima koje sadrže i njihovim redosledom izvršavanja. Svaki nivo može da sadrži do 16 modula. Sadržaj nivoa i redosled modula je definisan preko ODT tabele (*Overlay Definition Table*) koja sadrži pokazivače na MIF tabelu svakog modula (detaljnije obrađeno u poglavlju 1.1)

7.2.3.1 Asemblerske direktive za nivoe

Kao što je opisano u vežbi 2, DSP ima dve odvojene 32-bitne memorijske zone podataka, X i Y, koje se mogu koristiti zajedno, kao jedna 64-bitna L zona, i programsku memoriju. Asemblerske direktive koje omogućavaju izbor ovih zona su:

1. `.code [at] [align] [value]` – za programsku memoriju
2. `.xdata [at] [align] [value]` – za memoriju X
3. `.ydata [at] [align] [value]` – za memoriju Y
4. `.data [at] [align] [value]` – za memoriju L

Za svaki nivo je rezervisan određeni deo memorije za podatke i programske memorije koji osiguravaju da svaki modul koristi sopstvenu memoriju, odnosno memoriju koja se nalazi unutar granica koje su određene za nivo kojem modul pripada. Važno je napomenuti da je ova podela memorije isključivo logičke prirode i da ona nije uslovljena hardverskim ograničenjima. Primer ovih asemblerskih direktiva dat je u donjem primeru:

```
.code_0/  .data_0/   .xdata_0/   .ydata_0 [at] [align] [value]
.code_1/  .data_1/   .xdata_1/   .ydata_1 [at] [align] [value]
.code_2/  .data_2/   .xdata_2/   .ydata_2 [at] [align] [value]
.code_3/  .data_3/   .xdata_3/   .ydata_3 [at] [align] [value]
```

7.2.3.2 Memorijska mapa nivoa

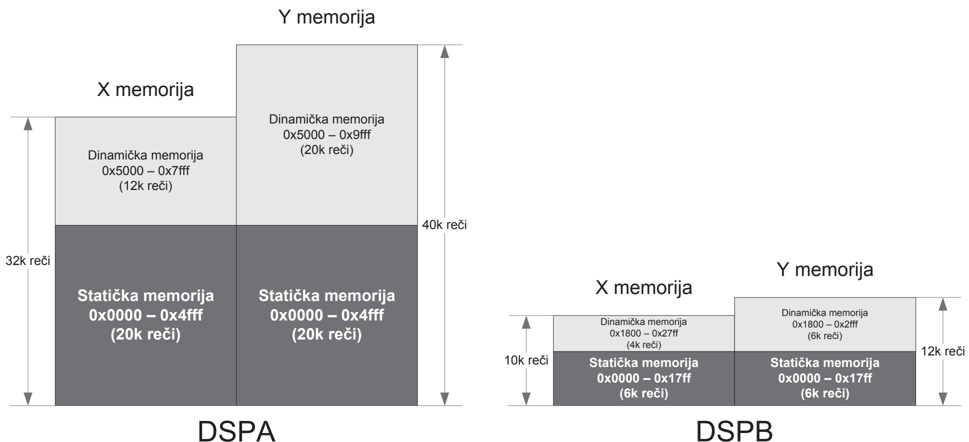
Svako jezgro CS48L20 procesora ima dve odvojene memorije za podatke, *X* i *Y*, i programsku memoriju. Veličina *Y* i programske memorije može da varira u zavisnosti od konfiguracije hardvera (ukupno 4 moguće kombinacije, gde se povećanjem memorijske zone Y smanjuje programska memorija i obrnuto), dok je memorijska zona X uvek iste veličine. U tabeli 7.1 su prikazane podrazumevane (*default*) vrednosti (hardverska konfiguracija 1) za DSP verziju Finn CS48L20.

Tabela 7.1 – Raspoloživa memorija Finn CS48L20 procesora

Tip memorije	DSP A	DSP B
X – data	32k SRAM, 2k ROM	10k SRAM, 1k ROM
Y – data	40k SRAM, 2k ROM	12k SRAM, 1k ROM
P – code	24k SRAM, 6k ROM	10k SRAM, 4k ROM

Radnim okvirom je dodatno uvedena logička podela memorije za podatke na dva segmenta (Slika 7.5):

1. memorija rezervisana za statičku dodelu.,
2. dinamički zauzeta memorija (*heap*) koja se dodeljuje pojedinim modulima na njihov zahtev (*malloc*).



Slika 7.5 - Organizacija SRAM memorije CS48L20 procesora

Pored ova dva glavna segmenta memorije, treba izdvojiti još i nekoliko rezervisanih memorijskih zona. To su:

- Sistemski ulazno/izlazni memorijski niz (eng. *I/O buffer*), koji se nalazi u *Y* dinamičkoj memoriji i služi za prenos/smeštanje PCM podataka. Opseg zauzetih adresa zavisi od željenog broja kanala na ulazu/izlazu i podržano je maksimalno 16 kanala.
- Ulazni FIFO memorijski niz (eng. *Input FIFO buffer*) se takođe nalazi nalazi u *Y* dinamičkoj memoriji i služi za prihvat ulaznog toka komprimovanih podataka (eng. *compressed stream*). Opseg zauzetih adresa zavisi od veličine FIFO memorijske strukture.
- Statička memorija je podeljena na segmente koji su dodeljeni odgovarajućem nivou programske podrške. Na taj način je omogućen nezavistan rad modula u različitim nivoima. Ovakvom podelom je na primer moguće promeniti dekoder bez ponovnog učitavanja operativnog sistema ili bilo kog modula koji se nalazi u nivou među ili završne obrade.

U zavisnosti od odabrane verzije radnog okvira postoje definisane **.xml* datoteke koje opisuju podelu statičke memorije.

Prilikom pravljenja projekta namenjenog za rad na razvojnoj ploči koriste se gore navedene datoteke, dok se u ranim fazama projekta, kada se razvoj obavlja u simulatorskom okruženju, koristi posebna **.xml* datoteka koja se prilagođava modulu koji se u simulatoru razvija. Primer dela jedne takve datoteke je (prikazana je samo podela X RAM segmenta):

```
<memory_map>
...

<class name="X" image_section="X" output_type="RAM">
  <start>0x0</start>
  <size>0x8000</size>
  <subclass>
     <class name="X_NEAR_OS" >         <start>0x0</start>     <end>0x1F</end>
</class>
     <class name="X_NEAR" >           <start>$</start>        <end>0x3F</end>
</class>
     <class name="X_0" >             <start>$</start>       <end>0x4FF</end>
       <aliases> <alias name="X_OS_RAM" />  </aliases>
     </class>
     <overlay_area>
         <!-- individual overlay areas -->
        <class    name="X_REGULAR">                    <start>X_0.e+1</start>
<end>X_HEAP.e</end>
          <subclass>
```

```
            <class   name="X_1"   >             <start>X_REGULAR.s</start>
<size>0x3200</size> </class>
            <class    name="X_2"    >                    <start>$</start>
<size>0x800</size>  </class>
            <class    name="X_3"    >                    <start>$</start>
<size>0x800</size>  </class>
            <class name="X_4" >     <start>$</start>        <end>X_HEAP.s-
1</end>    </class>
            <class  name="X_HEAP" >   <start>0x5000</start>   <end>X.e</end>
</class>
          </subclass>
        </class>
        <!-- spanning overlay areas -->
        <class name="X_1_2" >       <start>X_1.s</start>   <end>X_2.e</end>
</class>
        <class name="X_1_2_3" >     <start>X_1.s</start>   <end>X_3.e</end>
</class>
        <class    name="X_4_HEAP"    >                <start>X_4.s</start>
<end>X_HEAP.e</end>     </class>
      </overlay_area>
  </subclass>
</class>

...
</memory_map>
```

7.2.4 Ulazno-izlazna sprežna memorija (*I/O Buffer*)

Unutar radnog okvira definisan je jedinstveni ulazno-izlazni niz koji služi za smeštanje podataka koji pristižu u sistem i odlaze iz njega (eng. Input-Output Buffer).

Svi nivoi, pa samim tim i svi moduli, pristupaju jedinstvenom U/I nizu. Ulazno/izlazna sprežna memorija je izdeljena na 16 segmenata po 128 lokacija za svaki od podržanih kanala. Zauzima se od strane operativnog sistema u Y dinamičkoj memoriji. OS je zadužen da redovno ažurira niz pokazivača na svaki od kanala, a moduli pristupaju podacima u ulazno/izlaznom memorijskom nizu putem ovih pokazivača. Obrada podataka nad ulazno/izlaznim memorijskim nizom se obavlja u blokovima od po 16 odbiraka unutar *Blok* pod-rutine. Pokazivači (tabela 7.2) se nalaze u memorijskoj zoni Y na adresi X_BY_IOBUFFER_PTRS (X_VY_IOBUFFER_0_PTR – X_VY_IOBUFFER_15_PTR). Moduli ne smeju da menjaju ove pokazivače (tj. oni su *read-only*).

Tabela 7.2 – Pokazivači na ulazno-izlaznu sprežnu memoriju

Rastojanje od prve lokacije	Ime pokazivača	Opis
0	X_BY_IOBuffer_Ptrs X_VY_IOBuffer_0_Ptr	Left Channel IO Buffer
1	X_VY_IOBuffer_1_Ptr	Center Channel IO Buffer
2	X_VY_IOBuffer_2_Ptr	Right Channel IO Buffer
3	X_VY_IOBuffer_3_Ptr	Left Surround IO Buffer
4	X_VY_IOBuffer_4_Ptr	Right Surround IO Buffer
5	X_VY_IOBuffer_5_Ptr	Surround Back Left IO Buffer
6	X_VY_IOBuffer_6_Ptr	Surround Back Right IO Buffer
7	X_VY_IOBuffer_7_Ptr	LFE0 IO Buffer
8	X_VY_IOBuffer_8_Ptr	Left High IO Buffer
9	X_VY_IOBuffer_9_Ptr	Right High IO Buffer
10	X_VY_IOBuffer_10_Ptr	Left Wide IO Buffer
11	X_VY_IOBuffer_11_Ptr	Right Wide IO Buffer
12	X_VY_IOBuffer_12_Ptr	Left DualZone IO Buffer
13	X_VY_IOBuffer_13_Ptr	Right DualZone IO Buffer
14	X_VY_IOBuffer_14_Ptr	Left Auxiliary IO Buffer
15	X_VY_IOBuffer_15_Ptr	Right Auxiliary IO Buffer

7.2.5 Softverska biblioteka za pristup ulaznoj FIFO memoriji

Prihvatanje komprimovanih podataka iz ulazne FIFO memorije u dekoder se vrši pozivanjem sistemskih funkcija koje su realizovane u okviru *BitRipper* biblioteke koji pripada OS-u. Ova biblioteka obezbeđuje sve operacije potrebne za pristup ulaznim podacima u dekoderima, tako da nema potrebe za neposrednim pristupom ulazu. Osnovna namena ove grupe funkcija je da omogući pristup ulaznim podacima na nivou pojedinačnih bita radi parsiranja kodiranog ulaznog toka. Spisak funkcija kao i njihov opis dat je u tabeli 7.3.

Tabela 7.3 – Funkcije BitRipper biblioteke

Ime Funkcije	Ulaz	Izlaz	Opis
X_S_BR_Init	–	–	Poziva se od strane jezgra operativnog sistema. Nakon završetka funkcije sadržaj sledećih registara će biti promenjen: x0, a0, a1
X_S_BR_SwitchInputFIFO	A0	–	Prebacuje sve operacija na ulazni niz određen sadržajem akumulatora A1. Nakon završetka funkcije sadržaj

			sledećih registara će biti promenjen: x0, a0, a1
X_S_BR_Extract	A0 – broj bita za čitanje sa ulaza	B0 – pročitani biti	Čita broj bita definisan akumulatorom A0 sa izlaza i smešta ih u akumulator B0. Prilikom završetka čitanja stanje pokazivača za čitanje se menja. Nakon završetka funkcije sadržaj sledećih registara će biti promenjen: a0, a1, a2, b0, b1, b2, x0, y0, i7, i6, nm6, nm7
X_S_BR_Peek	A0 – broj bita za čitanje sa ulaza	B0 – pročitani biti	Čita broj bita definisan akumulatorom A0 sa izlaza i smešta ih u akumulator B0. Prilikom završetka čitanja stanje pokazivača za čitanje se ne menja. Nakon završetka funkcije sadržaj sledećih registara će biti promenjen: a0, a1, a2, b0, b1, b2, x0, y0, i7, i6, nm6, nm7
X_S_BR_SaveMainState	–	–	Pozivom ove funkcije sačuvaće se i zamrznuti stanje glavnog pokazivača za čitanje, tako da svaka dalja operacija sa ulaznim nizom neće imati uticaja na njega. Nakon završetka funkcije sadržaj sledećih registara će biti promenjen: a0
X_S_BR_RestoreMainState	–	–	Pozivom ove funkcije učitaće se i odmrznuti stanje glavnog pokazivača za čitanje. Nakon završetka funkcije sadržaj sledećih registara će biti promenjen: a0
X_S_BR_LoadMainState	I6 – pokazivač na memorijski	–	Pozivom ove funkcije učitava se glavno stanje iz pomoćnog stanja zadatog indeks registrom i6. Trenutno stanje BitRipper

	blok u memoriji X gde je sačuvano ulazno stanje		modula se menja u glavno stanje, koje koristi glavni pokazivač za čitanje. Nakon završetka funkcije sadržaj sledećih registara će biti promenjen: a0
X_S_BR_SaveAuxState	I6 – pokazivač na memorijski blok dugačak 8 tridesetdvobitnh reci, smešten u memoriju X.	–	Pozivom ove funkcije sačuvaće se trenutno stanje modula BitRipper u pomoćno stanje definisano indeksnim registrom I6. Nakon završetka funkcije sadržaj sledećih registara će biti promenjen: a0
X_S_BR_LoadAuxState	I6 – pokazivač na blok u memoriji X gde je pomoćno stanje sačuvano	–	Pozivom ove funkcije biće vraćeno pomoćno stanje sačuvano u bloku na koji pokazuje indeks registar I6. Nakon završetka funkcije sadržaj sledećih registara će biti promenjen: a0
X_S_BR_SkipBits	A0 – broj bit koji se preskače	–	Preskače određeni broj bit pomeranjem pokazivača za čitanje unapred ako je broj u akumulatoru A0 pozitivan, odnosno unazad ukoliko je broj u akumulatoru AO negativan. Nakon završetka funkcije sadržaj sledećih registara će biti promenjen: a0, a1, b0, b1
X_S_BR_ReadDipstick	–	B0	Proverava koliko je bit ostalo za čitanje u ulaznom nizu i tu informaciju smešta u akumulator B0. Nakon završetka funkcije sadržaj sledećih registara će biti promenjen: a0, a1, a2, b0, b1

X_S_BR_WaitOnDipstick	A0 – broj bit koji treba da stignu u ulaz	–	Rad ove funkcije će se završiti onda kada ulazni niz stigne broj bita određen vrednošću akumulatora A0– Nakon završetka funkcije sadržaj sledećih registara će biti promenjen: a0, a1, a2, b0, b1
X_S_BR_BitCnt_States	I6, I7 – pokazivači na pomoćna stanja ulaza u memoriji X	B0 – broj bita	Vraća broj bita, odnosno distancu, između dva pomoćna stanja u ulaznom nizu (I7 – I6). Nakon završetka funkcije sadržaj sledećih registara će biti promenjen: a0, a1, b0, b1
X_S_BR_BitCnt_Main_State	I7 – pokazivač na pomoćno stanje u memoriji X	B0 – broj bita	Vraća broj bita, odnosno distancu, između pomoćnog i glavnog stanja. Nakon završetka funkcije sadržaj sledećih registara će biti promenjen: a0, a1, b0, b1
X_S_BR_SaveAlignment	A0 – broj bita od kojih je ulaz poravnat (A>=0)	I7 – pokazivač na glavno stanje	Pozivom ove funkcije sačuvaće se poravnanje ulana na 8, 16 ili 31 bita, počevši od bita udaljenog A0 bita od trenutnog stanja. Nakon završetka funkcije sadržaj sledećih registara će biti promenjen: a0, a1
X_S_BR_AlignToByte	–	–	Podešava stanje ulaza, pomerajući trenutni pokazivač za potreban broj bita da bi se ulaz poravnao na 8 bita uzimajući u obzir informacije sačuvane pozivom funkcije X_S_BR_SaveAlignment. Nakon završetka funkcije sadržaj sledećih registara će biti promenjen: a0, a1, b0, b1
X_S_BR_AlignToWord	–	–	Podešava stanje ulaza, pomerajući trenutni pokazivač za potreban broj bita da bi se ulaz poravnao na 16 bita,

			uzimajući u obzir informacije sačuvane pozivom funkcije X_S_BR_SaveAlignment. Nakon završetka funkcije sadržaj sledećih registara će biti promenjen: a0, a1, b0, b1
X_S_BR_AlignToDword	–	–	Podešava stanje ulaza, pomerajući trenutni pokazivač za potreban broj bita da bi se ulaz poravnao na 32 bita uzimajući u obzir informacije sačuvane pozivom funkcije X_S_BR_SaveAlignment. Nakon završetka funkcije sadržaj sledećih registara će biti promenjen: a0, a1, b0, b1
X_S_BR_ReadInputFIFO WrPtr	–	–	Vraća trenutni pokazivač za pisanje u ulazni niz.

7.2.6 Uprošćen prikaz sprege modula sa ulazom i izlazom

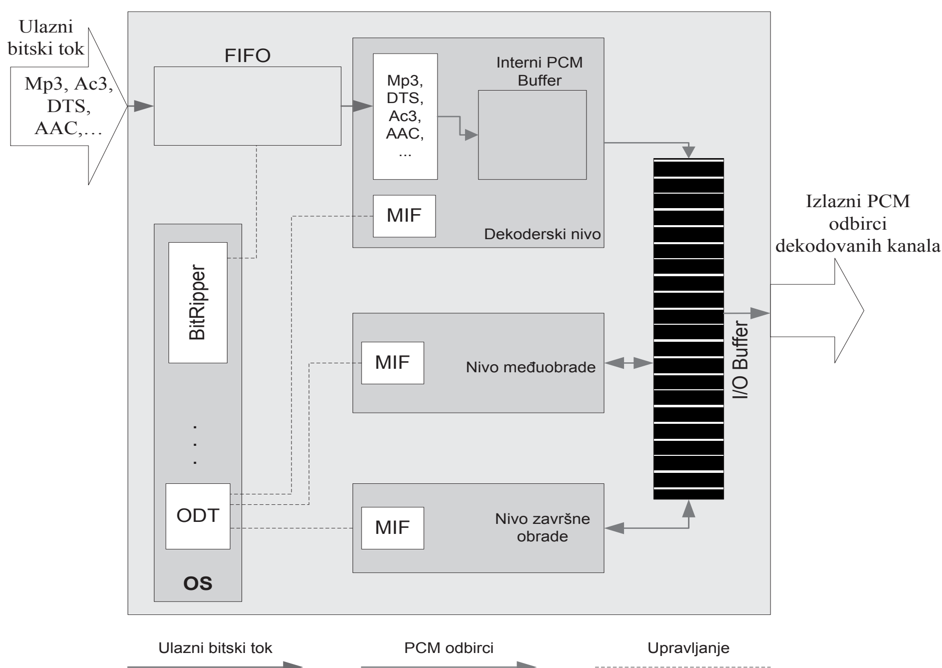
Na primeru jedne dekoderske aplikacije će se razmotriti redosled i tok obrade podataka unutar procesora, slika 7.6. Neka se na ulazu u sistem nalazi audio sadržaj kompresovan mp3 algoritmom. Ulazni podaci se putem DMA kanala smeštaju u ulaznu FIFO sprežnu memoriju. OS, koristeći informacije o prisutnim modulima koje se nalaze u ODT tabeli, poziva module po unapred definisanom redosledu koristeći sledeće pravilo: uvek se izvršavaju podrutine iste namene svih modula pre nego se krene na druge podrutine – što bi značilo da se prvo izvrši *Pre-Kickstart* rutina dekodera, pa *Pre-Kickstart* rutina modula u nivou među-obrade, zatim *Pre-Kickstart* nivoa završne obrade, pa se tek onda kreće sa izvršavanjem *Post-Kickstart* dekodera, nivoa među i završne obrade, itd.

Pozivom modula *mp3* u dekoderskom nivou vrši se dekodovanje sadržaja koji se nalazi u FIFO memoriji. Koristeći rutine *BitRipper* modula *mp3* dekoder čita sadržaj iz FIFO bafera i dekoduje ih. Dekodovani PCM odbirci se ne mogu neposredno upisati u IO memorijski niz, oni se smeštaju u interni PCM memorijski niz odakle se kopiraju u IO memorijski niz. Razlozi za ovo su:

- IO memorijski niz može da uskladišti samo 128 PCM odbiraka po kanalu dok se dekodovanjem minimalne jedinice jednog ulaznog bitskog toka može dobiti više odbiraka,

- da se izbegne čekanje da dekoder završi sa dekodovanjem jedne minimalne jedinice dekodovanja pre nego što se pozovu moduli iz druga dva nivoa obrade.

Da bi se zadovoljila ova dva uslova, interni PCM memorijski niz ima posebnu strukturu. Pretpostavimo da se dekoduje stereo *mp3* bitski tok čija minimalna jedinica dekodovanja (u daljem tekstu celina) sadrži 256 PCM odbiraka. Kada bi interni PCM memorijski niz imao takvu strukturu da može da skladišti po 256 odbiraka svakog kanala, pre nego što se počne sa dekodovanjem druge celine, sistem bi morao da čeka da se svi PCM-ovi oba kanala prebace u IO memorijski niz i da se nad njima završi sva obrada definisana u nivou međuobrade i nivou završne obrade.

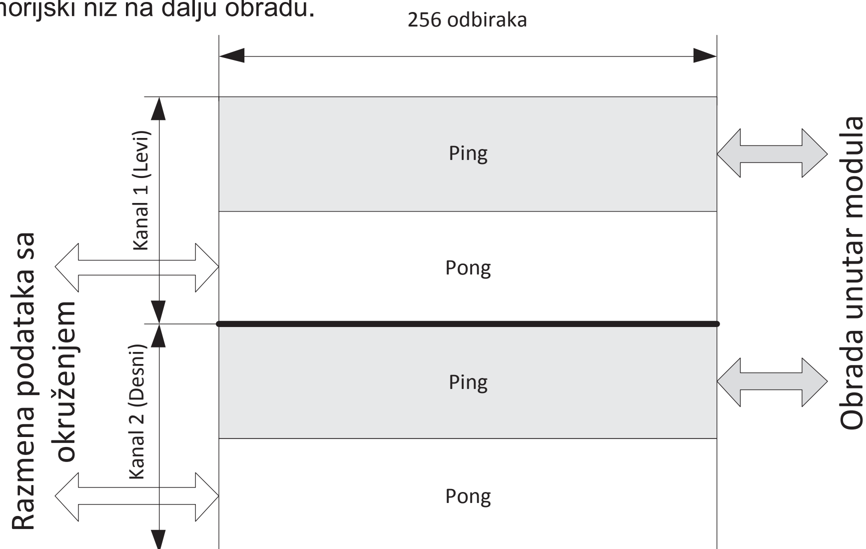


Slika 7.6 - Sprega dekodera sa ulazom izlazom

Da bi se čekanje izbeglo, koristi se pristup koji se naziva dvostruko skladištenje podataka (eng. *double buffering*). Ovaj pristup podrazumeva da se istovremeno vrši učitavanje novih podataka i obrada nad već učitanim podacima. Dva primera dvostukog skladištenja podataka su *Ping Pong* šema skladištenja i horizontalno-vertikalna šema skladištenja (*Criss Cross*).

7.2.6.1 *Ping Pong skladištenje PCM odbiraka*

Ping Pong skladištenje PCM odbiraka podrazumeva modifikaciju PCM memorijskog niza tako da može da skladišti dvostruko veći broj odbiraka po kanalu. Ovakva struktura se naziva *Ping-Pong Memorijski niz* i prikazana je na slici 7.7. Svaki kanal ima dva dela, *Ping* i *Pong*. Na primer, dok dekoder dekoduje i upisuje sadržaj u *Ping* deo levog i desnog kanala, sadržaj iz *Pong* delova se prosleđuje u IO memorijski niz na dalju obradu u ostalim nivoima obrade. Kada dekoder završi sa dekodovanjem i napuni *Ping* delove oba kanala, situacija se obrće i sada se dekodovani sadržaj upisuje *Pong*, dok se PCM-ovi iz *Ping* prosleđuju u IO memorijski niz na dalju obradu.

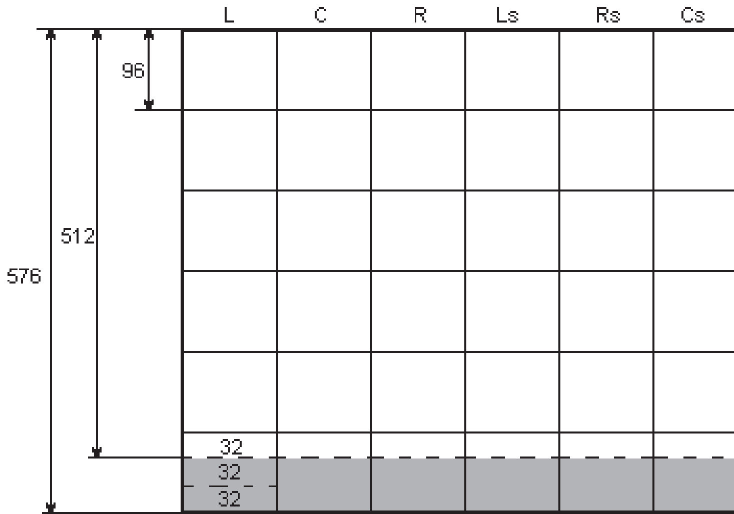


Slika 7.7 - Struktura Ping-Pong-a za dva kanala

Kao što je gore objašnjeno, dekoderski nivo generiše PCM odbirke koji se prosleđuju u IO memorijski niz. Operativni sistem zatim poziva module koji se nalaze u nivou za međuobradu i završnu obradu, respektivno. Ako se obrada vrši nad maksimalno 16 odbiraka, kolika je veličina jednog okvira, vrši se neposredna obrada nad podacima koji se nalaze u IO memorijskom nizu i tako obrađeni podaci prosleđuju na izlaz. U slučaju da se obrada radi na većem broju odbiraka od veličine jednog okvira, potrebno je koristiti interni memorijski niz kao i kod dekodera. On se puni iz ulaznog IO memorijskog niza u *block* rutini, a obrada nad prikupljenim podacima se izvršava u *background (decoder)* rutini. Paralelno sa punjenjem internog memorijskog niza, u *block* rutini se već obrađeni podaci iz internog memorijskog niza vraćaju u IO memorijski niz.

*7.2.6.2 Horizontalno-vertikalna šema skladištenja PCM odbiraka (*eng. *criss-cross)*

Osnovna karakteristika *Criss-Cross* šeme skladištenja odbiraka jeste velika ušteda memorijskog prostora. Jedan od primera koji ovo pokazuje je upoređivanje *Ping-Pong* i *Criss-Cross* metoda za obradu podataka u audio dekoderima, koji dekodovanjem minimalne jedinice generišu N=512 odbiraka po kanalu. U ovom slučaju *Ping-Pong* buffer će za jedan kanal zahtevati 2x512 memorijskih lokacija.



Slika 7.8 - Inicijalno stanje Criss – Cross bafera

Ukoliko dekoder podržava 6 kanala tada je ukupan broj lokacija za smeštanje i obradu odbiraka svih kanala jednak:

$$512*2*6 = 6144 \approx 6K$$

Sa druge strane, upotreba *Criss-Cross* koncepta će zahtevati duplo manje memorijskog prostora za istu konfiguraciju dekodera. Izgled ovog bafera za 6 kanalni dekoder dat je na slici 7.8.

Dimenzija *Criss-Cross* bafera se računa na osnovu izraza:

$$BS * nCh * x >= N,$$

gde je:

- BS – Brick Size (veličina jedinice obrade),
- nCh – Broj podržanih kanala,

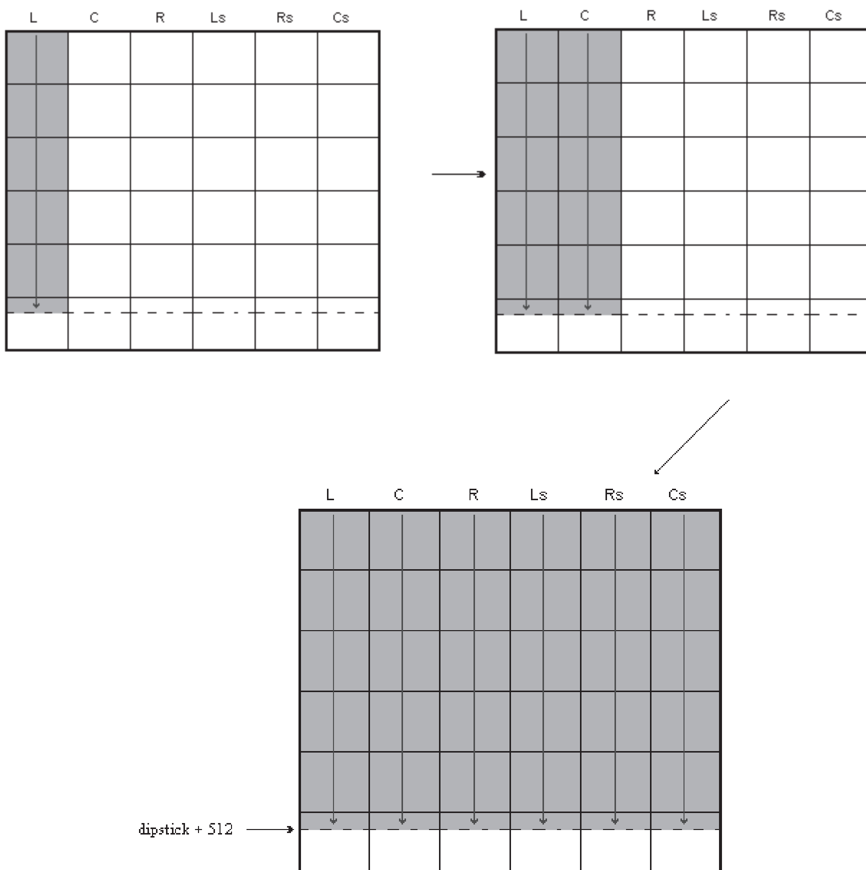
- x – umnožak koji obezbeđuje da broj N bude prvi veći ili jednak broj od broja odbiraka koje dekoder izbacuje po kanalu nakon dekodovanja jedne jedinice dekodovanja.

Količina memorije koja je potrebna za ovakvu strukturu je nCh*(BS * nCh * x).

Primer:

```
BS  = 16
nCh = 6
N = 512
                    BS * nCh * x = 16 * 6 * 6 = 576
```

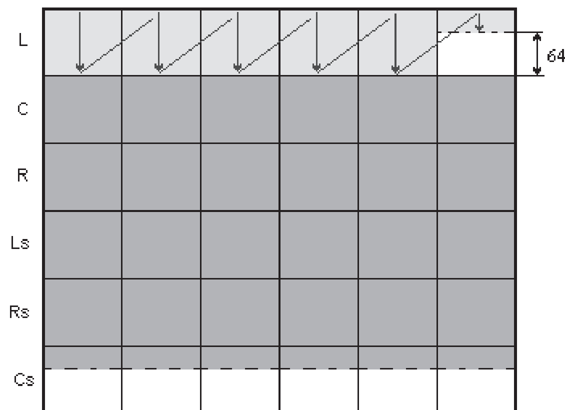
Pa je dimenzija *Criss-Cross* – bafera 576 po kanalu, a ukupni memorijski zahtevi 6*576 = 3456 ≈ **3.4K**



Slika 7.9 - Punjenje bafera

Princip rada sa ovom vrstom strukture je: smeštanje odbiraka počinje nad prvim kanalom *(L)* u smeru od gore ka dole i nastavlja se u istom maniru za sve kanale, slika 7.9. Kada se svi kanali upišu, uvećava se dubina upisa za 512, što predstavlja indikator da su svi kanali dekodovani, tj da je memorijski niz za smeštanje ulaznih odbiraka pun. Sada se prelazi na izbacivanje redom prvog, drugog, itd., odbirka svakog kanala istovremeno, kao što je pokazano na slici 7.10. Kada 96 odbiraka svakog kanala izađe iz bafera, oslobodio se prostor za punjenje bafera podacima prvog kanala sledeće jedinice dekodovanja. Slobodan prostor je dovoljan za smeštanje 576 odbiraka, tako da će nakon punjenja ostati 64 lokacije slobodne

Kada se ceo memorijski niz napuni u horizontalnom smeru, vrši se izbacivanje redom prvog, drugog, itd., odbirka svakog kanala paralelno; kada se izbaci 96 odbiraka svakog kanala, oslobodio se prostor za smeštanje podataka prvog kanala treće jedinice dekodovanja u vertikalnom smeru i nastavlja se proces baferovanja kao i kod prve jedinice dekodovanja.



Slika 7.10 - Punjenje bafera podacima druge jedinice dekodovanja u horizontalnim smeru.

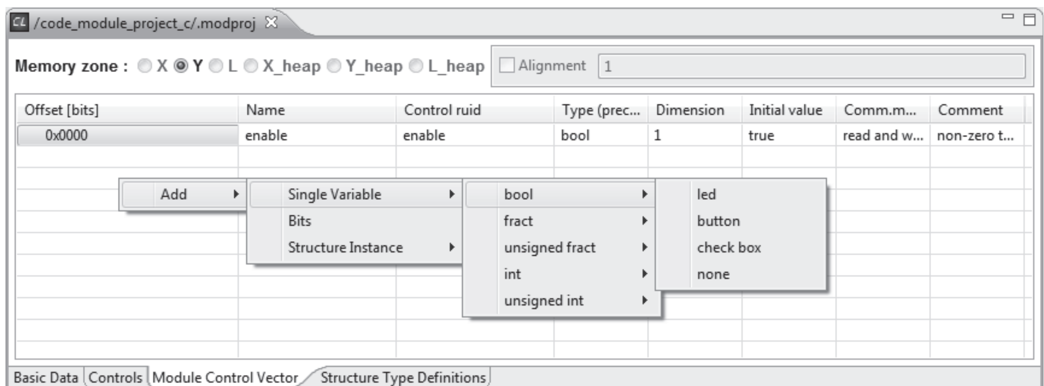
7.3 Grafičko uređivanje MCV tabele

U okviru integrisanog razvojnog okruženja CLIDE omogućeno je grafičko uređivanje MCV tabele modula. Otvaranje prozora za uređivanje vrši se otvaranjem *.modproj* datoteke i odabirom MCV kartice. MCV tabela je predstavljena u obliku tabele. Svaki red tabele predstavlja jedno polje u MCV strukturi. Prilikom prevođenja modula, na osnovu date tabele se generiše odgovarajuća C ili asemblerska struktura.

Dodavanje novih elemenata vrši se pritiskom desnog tastera miša i odabirom opcije *Add.* Kolone u tabeli sadrže sledeće osobine:

- *offset* – odstojanje polja od početka MCV tabele (automatski se izračunava),
- name – naziv polja, odgovara nazivu generisane promenljive,
- *ruid* – jednistveni identifikator grafičke kontrole povezane sa ovom MCV promenljivu,
- *type* – tip podataka pridružen promenljivoj,
- *dimension* – dimenzija promenljive, ukoliko je različita od 1 promenljiva predstavlja niz promenljivih označenog tipa,
- *init value* – inicijalna vrednost,
- *comm mode* – ovlašćenja OS nad promenljivom (čitanje, pisanje ili oba).

Brisanje elemenata MCV tabele se vrši pritiskom desnog tastera miša i odabirom opcije *Delete.*



Slika 7.11 – Uređivač MCV tabele (dodavanje nove promenljive)

Svakoj MCV promenljivoj je moguće pridružiti grafičku kontrolu. Ova kontrola služi za upravljanje MCV tabelom u toku kontrolisanog izvršavanja modula. Dodavanje kontrola se vrši odabirom kartice *Controls* u okviru uređivača .modproj datoteke. Svaku kontrolu je moguće povezati sa MCV promenljivom koristeći njen jedinstveni identifikator (*RUID*). Nakon pokretanja aplikacije u okviru CLIDE okruženja, svaka promena vrednosti ovih kontrola rezultuje upisom novih vrednosti u MCV tabelu modula koji se izvršava na uređaju ili u simulatoru.

Podešavanje kontrola vrši se u okviru *Property View* prozora. Ovaj prozor je kontekstualan, drugim rečima, prikazuje podešavanja vezana za trenutno označenu

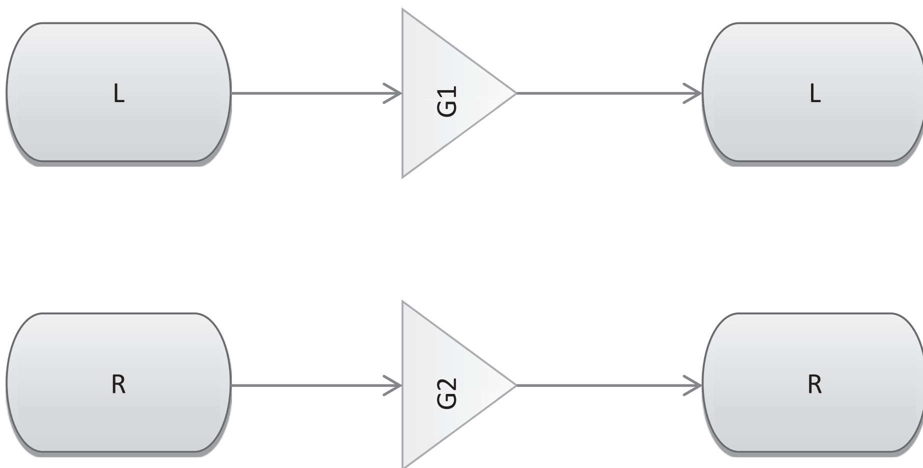
kontrolu. Podešavanja se odnose na naziv, tip podataka, ograničenja vrednosti, početnu vrednost i vizualne karakteristike kontrola.

Spisak svih postojećih kontrola i njihove osobine razlikuju se u različitim verzijama CLIDE okruženja, i moguće ih je pregledati u odeljku *Clide Help* -> *Firmware Developement Guide* -> *Control Panel.s*

7.4 Zadaci za samostalnu izradu

7.4.1 Zadatak 1: Korišćenje radnog okvira na sistemima baziranim na procesoru CS48x

U okviru prvog zadatka na priloženom primeru upoznaće se sa korišćenjem radnog okvira na sistemima baziranim na procesoru CS48x. Dati primer predstavlja sistem koji vrši pojačanje, odnosno slabljenje signala. Na ulazu u sistem očekivana su 2 kanala. Koeficijenti pojačanja prosleđeni su sistemu kroz MCV tabelu. Za dva koeficijenta pojačanja realizovane su dve kontrole tipa *„knob",* koje omogućavaju korisniku da menja pojačanje u toku izvršenja programa.



7.4.1.1 Postavka zadatka:

1. Uvući sve projekte iz direktorijuma *Zadatak1* u razvojno okruženje.
2. Proučiti kod unutar *PCM_Passthrough_asm* modula. Analizirati Brick i Background rutinu.
3. Pokrenuti priloženi projekat koristeći simulator.
    a. za pokretanje na simulatoru koristiti *simulator_app* projekat,
    b. pre pokretanja potrebno je podesiti ulaznu i izlaznu datoteku unutar *file_io* izvršnog okruženja.
4. Otvoriti prozor sa kontrolama *PCM_Passthrouth_asm* modula.
5. Menjati kontrole pojačanja u toku izvršenja programa.
6. Zaustaviti izvršenje i analizirati izlaznu datoteku koristeći alat *Audacity*.
7. Pokrenuti priloženi primer na razvojnoj ploči. Za pokretanje na ploči koristiti *CDB49X_DC48L20_app* projekat i *analog* izvršno okruženje.
8. Povezati *Line out* izlaz na računaru na audio ulaze na ploči i slušalice na audio izlaz na razvojnoj ploči.

9. Otvoriti prozor sa kontrolama *PCM_Passthrouth_asm* modula.
10. Menjati kontrolu pojačanja u toku izvršenja programa.

7.4.2 Zadatak 2: Ugrađivanje sistema za dodavanje višestrukog eho efekta u radni okvir

Cilj ovog zadatka je ugradnja realizovanog sistema za dodavanje višestrukog eho efekta u radni okvir na sistemima baziranim na procesoru CS48x, kako bi se omogućilo izvršavanje ove aplikacije na DSP uređaju.

7.4.2.1 Postavka zadatka:

1. Otvoriti projekat *multitapEcho_framework* u okviru CLIDE razvojnog okruženja.
2. Prekopirati datoteke *multitapEcho.a* iz prethodne vežbe u pomenuti projekat.
3. Definicije globalnih konstanti premestiti u mtechoGlobalConsts.h.
4. Definiciju strukture stanja smestiti u mtechoDataVars.a.
5. Izmeniti kod *multitap_echo* funkcije tako da se umesto *BLOCK_SIZE* konstante definisane od strane korisnika koristi *___X_VY_BLOCK_SIZE* .
6. U okviru *postKickStart* rutine pozvati funkciju za inicijalizaciju eho modula.
7. U okviru *brick* rutine pozvati *multitap_echo* funkciju. Kao uzlazni i izlazni niz potrebno je proslediti pokazivač na prvi kanal *(___X_VX_PPM_INPUT_CHANNELS[0]*).
8. Dodati promenljive za podešavanje pojačanja u MCV tabelu. Za svaku promenljivu napraviti i grafičku kontrolu tipa *knob.* Kao tip podataka kontrole odabrati *level_db.* Ograničiti opseg kontrole od -24dB do 0dB. Postavit inicijalne vrednosti kontrola u MCV tabeli na vrednosti koje su ranije bile definisane kao konstante za inicijalizaciju.
9. Pokrenuti priloženi projekat koristeći simulator
    a. za pokretanje na simulatoru koristiti *simulator_app* projekat,
    b. odabrati za ulaznu datoteku onu za koju postoje izgenerisani referentni izlazi iz prethodnog zadatka,
    c. **ne menjati** kontrole u toku izvršenja,
    d. prekinuti izvršenje i uporediti izlaznu datoteku sa referentnom.
11. Pokrenuti priloženi projekat na razvojnoj ploči. Za pokretanje na ploči koristiti C*DB49X_DC48L20_app* projekat, i *analog* izvršno okruženje.
12. Povezati *Line out* izlaz na računaru na audio ulaze na ploči i slušalice na audio izlaz na razvojnoj ploči.
13. Otvoriti prozor sa kontrolama *multitapEcho_framework* modula.
14. Menjati kontrole pojačanja u toku izvršenja programa.
15. Izvršiti procenu utroška resursa čitave aplikacije nakon ugradnje u programsko okruženje.