UNIVERZITET U NOVOM SADU
FAKULTET TEHNIČKIH NAUKA
Odsek za elektrotehniku i računarstvo
Institut za računarstvo i automatiku
Katedra za računarsku tehniku i računarske komunikacije

Jedno rešenje kros asemblera – deo za jedinicu za generisanje adresa

- Diplomski rad iz predmeta -
- Programski prevodioci -

Mentor:
Prof. Dr Miroslav Popović

Kandidat:
Vladimir Velisavljev, E8507

Novi Sad, novembar 2006.

1

1 Uvod

U ovom radu je realizovan prednji deo krosasemblera sa procesora DSP Motorola 56000 na procesor Cirrus Logic Coyote, koji se odnosi na jedinicu za generisanje adresa. Krosasembler je realizovan razvojem prednjeg dela i ponovnim korišćenjem zadnjeg dela, koji je realizovan na katedri za računarsku tehniku i računarske komunikacije. Prednji deo se realizuje korišćenjem programskih alata FLEX i YACC i dopisivanjem rutina u programskom jeziku C. Za potrebe verifikacija dobijenih rešenja realizovano je okruženje za automatsko testiranje sa testnim slučajevima za pojedinačne instrukcije i aplikacije za procenu performansi izlaznog koda.

Pojavom novih DSP procesora neophodno je razviti čitav niz aplikacija koje će se izvršavati na njima. Uzevši u obzir sličnost novih arhitekura sa starijim, nameće se mogućnost iskorišćenja aplikacija razvijanih na staroj platformi. Ručno prilagođavanje asemblerskog koda naporan je i mukotrpan proces, a upotreba viših programskih jezika i odgovarajućih prevodioca ne daje zadovoljavajuće rezultate sa aspekta optimizacije brzine izvršavanja.

Prirodno rešenje problema je razvijanje krosasemblera, programskog prevodioca koji prevodi asemblerski jezik starije platforme na novu. Ovim rešenjem postiže se veliko iskorišćenje već postojećih optimizacija kao i mogućnost implementiranja novih optimizacija specifične procesoru novije arhitekture.

Postojanje C kompajlera prilagođenog novijoj arhitekturi čini ovaj problem jednostavnijim. Realizaciju rada čini izmena prednjeg dela C kompajlera. Ovu mogućnost otvara moderan način implementiranja kompajlera, čiji su prednji i zadnji deo modelovani tako da budu nezavisni i izmenjivi. Takođe, neophodno je rešiti čitav niz nekompatibilnosti dve DSP arhitekture.

Rad je sačinjen od 6 poglavlja.

U drugom poglavlju opisane su teorijske osnove rada, kroz objašnjenje moderne implementacije kompajlera. Dat je kratak pregled projektovanja kompajlera u osnovnim crtama.

U trećem poglavlju dat je pregled arhitektura oba procesora, kao i specifičnosti u kojima se ove arhitekture razlikuju.

U četvrtom poglavlju je opisana sama realizacija prednjeg dela krosasemblera.

U petom poglavlju dat je pregled svega što je urađeno u ovom radu.

Sadržaj šestog poglavlja je lista literature koja je korištena za izradu rada.

2 Teorijske osnove rada

2.1 Osnove programskih prevodioca

Zadatak programskog prevodioca (*eng. compiler*) je da prevodi programe napisane na višem programskom jeziku na mašinski jezik. U opštem slučaju jezik A je nezavisan od mašine M i manipuliše tipovima i strukturama podataka, koje mašina nije u stanju da prepozna, dok sa druge strane mašina rukovodi registrima i memorijom, resursima koji su nedostupni višim programskim jezicima. Samim tim, program napisan u programskom jeziku A ne bi mogao da radi na mašini M, bez prevođenja na jezik mašine.

Uloga programskog prevodioca je da pročita ulazni program, napisan na izvornom jeziku, da prepozna značenje i prevede u ekvivalentni program na ciljnom programskom jeziku. Da bi ovo prevođenje bilo uspešno, potrebno je poznavati sintaksu i semantiku oba jezika. Takođe je potrebno svesti strukture podataka početnog jezika na strukture podataka jezika objekta, a da se pri tom ne promeni njihovo značenje. Mora se doneti odluka o tome kako će se izrazi, iskazi, i ostale strukture podataka polaznog jezika prevesti na objekte odredišnog jezika.

Preslikavanje polaznog jezika na mašinski jezik se odvija u nekoliko faza. Pre svega je važno odabrati način predstavljanja ciljnog jezika za objekte polaznog jezika. Zatim treba odlučiti koji će se nizovi instrukcija generisati u odredišnom jeziku, potom izračunati vrednosti izraza, odabrati šema predstavljanja organizacije procedura u jeziku objekta, kao i odabrati niz mašinskih instrukcija koje odgovaraju iskazima osnovnog jezika. Pored svega nabrojanog, neophodno je poznavati i okruženje u kojem će se prevedeni program koristiti, tj. okruženje u kojem će se pozivati procedure koda jezika objekta.

Prevodioc mora prepoznavati strukture procedura polaznog jezika. Prepoznavanje je postignuto izolovanjem svakog od njenih sastavnih delova. Izdvojeni delovi transformišu se u semantički ekvivalentne delove mašinskog jezika, koji se na kraju sklapaju u jednu logičku celinu radi formiranja procedura u mašinskom jeziku.

Prepoznavanje strukture se naziva sintaksnom analizom (*eng. parsing*), prevođenje strukture na mašinski jezik se naziva generisanjem koda. Pored ove dve faze, postoje još tri funkcije, koje zajedno čine prevodioc, a to su:

- ulazna sprežna komponenta
- izlazna sprežna komponenta
- optimizacija koda.

Ulazna sprežna komponenta se još naziva i leksička analiza i služi da prevede procedure osnovnog jezika sa osnovnog oblika, koji je diktiran tipom osnovnog jezika u interni oblik, a pogodan je za obradu od strane prevodioca. Izlazna sprežna komponenta se naziva asembliranje. Ova komponenta transformiše interni oblik procedura pogodnije za prevodioc u oblik koji je pogodniji za rad punjača. Optimizacija se odnosi na primenu

algoritama transformacija nad procedurama radi dobijanja mašinskog jezika sa što manjim brojem instrukcija u fazi generisanja izlaznog programa.

Gore nabrojanih pet funkcija se može posmatrati kao pet faza procesa prevođenja. Kod velikih prevodioca ove faze su uglavnom jasno izražene, dok kod manjih to često nije slučaj, već su neke od njih spojene.

Procedure polaznog jezika se dovode na ulaz prevodioca u obliku nestrukturiranih nizova znakova. Osnovni elementi procedura su identifikatori, konstante, operatori i znakovi punktuacije. Identifikatori i konstante imaju promenljivu dužinu. Operatori su sastavljeni od dva znaka, zbog ograničenog skupa znakova.

Faza sintaksne analize prepoznaje strukturu procedura polaznog jezika, ali ne utvrđuje strukturu pomoću koje će se formirati identifikatori.

Faza leksičke analize zamenjuje svaki osnovni element jednim simbolom *(eng. token)* i izbacuje znakove razmaka. Polazna procedura se transformiše iz niza znakova u niz simbola, po jedan za svaki element. Simbol (t,i) se sastoji iz dve informacije, i ima isti oblik i veličinu:

- t je tip objekta koji je predstavljen (identifikator, operator, ...)
- i definiše koji je to objekat datog tipa.

Niz simbola koji izlazi iz leksičke analize ne sadrži nikakvu strukturu, već samo predstavlja niz simbola koji odgovaraju osnovnom elementu u proceduri osnovnog jezika, u redosledu u kojem se pojavio.

Generisanje izlaznog programa iziskuje identifikovanje članova, izraza, iskaza, i ostalih sintaksnih elementa koji se pojavljuju u procedurama osnovnog jezika. Faza sintaksne analize transformiše niz simbola u odgovarajući prelazni oblik, koji je neophodan u daljim fazama prevođenja. Šema faza rada prevodioca data je na slici 2.1.

Slika 2.1 Šema faza kompajlera

2.2 Leksička analiza

Zadatak faze leksičke analize je analiza polaznog programa i formiranje sintaksnih klasa. Ona služi za prepoznavanje osnovnih elemenata ili simbola polaznog jezika.

Leksičku analizu čine tri osnovna zadatka:

1. podela niza znakova osnovnog jezika u kojem je program napisan, u osnovne elemente ili simbole (token) jezika,
2. formiranje tabele literala i tabele identifikatora
3. formiranje tabele uniformnih simbola

Pretraživanje polaznog programa je sekvencijalno. Osnovni elementi se označavaju znakovima praznine, operatorima i specijalnim simbolima, čime se oni prepoznaju kao identifikatori, literali i terminalni simboli. Osnovni elementi se smeštaju u tabele, koje se u kasnijim fazama proširuju dodatnim informacijama, kao što su preciznost, tip podatka, dužina, klasa smeštanja u memoriju,... Ovi podaci su potrebni tokom celog procesa prevođenja, pa je potrebno omogućiti pristup ovim informacijama tokom ostalih faza prevođenja.

U fazi leksičke analize koristi se i tabela terminala, koja se sastoji od po jednog elementa za svaki treminalni simbol.

U tabeli literala se nalaze literali. Literali su specijalna vrsta simbola koji se odnosi na konstante čija je vrednost identična vrednosti samog literala. Adresa literala označava lokaciju literala u vreme izvršenja, a unosi se u narednim fazama prevođenja.

Tabela identifikatora se formira leksičkom analizom, i sadrži opis svih identifikatora koji se nalaze u početnom programu. U kasnijim fazama se unose dodatne informacije.

Tabela simbola se takođe formira tokom leksičke analize, i služi za predstvaljanje niza znakova početnog jezika kao niza simbola.

Algoritam leksičke analize kao prvi zadatak ima da izvrši analizu ulaznih nizova znakova, i njihovo transformisanje u simbole. Leksička analiza prepoznaje tri tipa simbola:

1. terminalne simbole
2. moguće identifikatore
3. literale

Leksička analiza upoređuje sve reči sa poljima u tabeli terminalnih simbola. Ukoliko se utvrdi da je simbol terminalni, leksički analizator formira tabelu simbola ukazujući na dati simbol. Ukoliko ne dođe do poklapanja dati simbol je ili mogući identifikator, ili literal. U slučaju mogućeg identifikatora simbol se upisuje u tabelu identifikatora, ukoliko se već ne nalazi u njoj. Brojevi, nizovi znakova pod navodnicima i ostali samodefinišući podaci se tretiraju kao literali, i upisuju se u tabelu literala. U slučaju da reči ne potpadaju ni pod jednu od ove tri grupe, izdaje se poruka o grešci. Kao primer algoritma leksičke analize dat je sledeći program:

```
main(){
   int a,b;
   scanf(" %d", &a);
   b = 2 + a;
   printf(" %d", b);
}
```

Tabela Identifikatora	
IND	IDENTIFIKATOR
1	main
2	scanf
3	X
4	Y
5	printf

Tabela 2.1 Tabela identifikatora

Tabela Literala	
IND	LITERAL
1	" % d"
2	2

Tabela 2.2 Tabela literala

Tabela Terminala	
IND	TERMINAL
1	{
2	}
3	,
4	;
5	int
6	*
7	(
8	)
9	=
10	&

Tabela 2.3 Tabela terminala

Tabela Uniformnih Simbola		
TIP	IND	SIMBOL
IND	1	main
TRM	7	(
TRM	8	)
TRM	1	{
TRM	5	int
IDN	2	a
TRM	3	,
IDN	3	b
TRM	4	;
IDN	4	scanf
TRM	7	(
LIT	1	" %d"
TRM	3	,
TRM	10	&
IDN	2	a
TRM	8	)
TRM	4	;
IDN	3	b
TRM	9	=
LIT	2	2
TRM	6	+
IDN	2	a
TRM	4	;
IDN	5	printf
TRM	7	(
LIT	1	" %d"
TRM	3	,
IDN	3	b
TRM	8	)
TRM	4	;
TRM	2	}

Tabela 2.4 Tabela uniformnih simbola

2.3 Sintaksna analiza

Prepoznavanje rečenica (sintaksne konstrukcije) u nizu simbola nastalih u leksičkoj analizi i interpretacija istih naziva se sintaksna analiza. Svaka rečenica se sastoji iz niza simbola koji imaju određeno značenje. Proces transformacije simbola u sintaksne konstrukcije naziva se sintaksna analiza. Pošto se izvrši ocena sintakse, vrši se interpretacija značenja (semantika). Značenje se definiše za svaku sintaksnu konstrukciju, bilo u obliku direktnog koda, ili u obliku neke od prelaznih konstrukcija. Sintaksni oblik polaznog jezika se definiše prema pravilima koja se nazivaju redukcijama. Redukcije definišu osnovne sintaksne konstrukcije i odgovarajuće rutine prevođenja (rutine akcija) koje se izvršavaju po prepoznavanju konstrukcije.

Namena sintaksne analize je prepoznavanje glavnih konstrukcija jezika i pozivanje odgovarajućih rutina akcije koje generišu prelazni oblik ili matricu ovih konstrukcija.

Redukcije su direktno zavisne od polaznog jezika, i ukoliko se menja polazni jezik, i one same moraju da se menjaju. Uobičajen način za realizovanje sintaksnog analizatora je u obliku jedinstvenog koda.

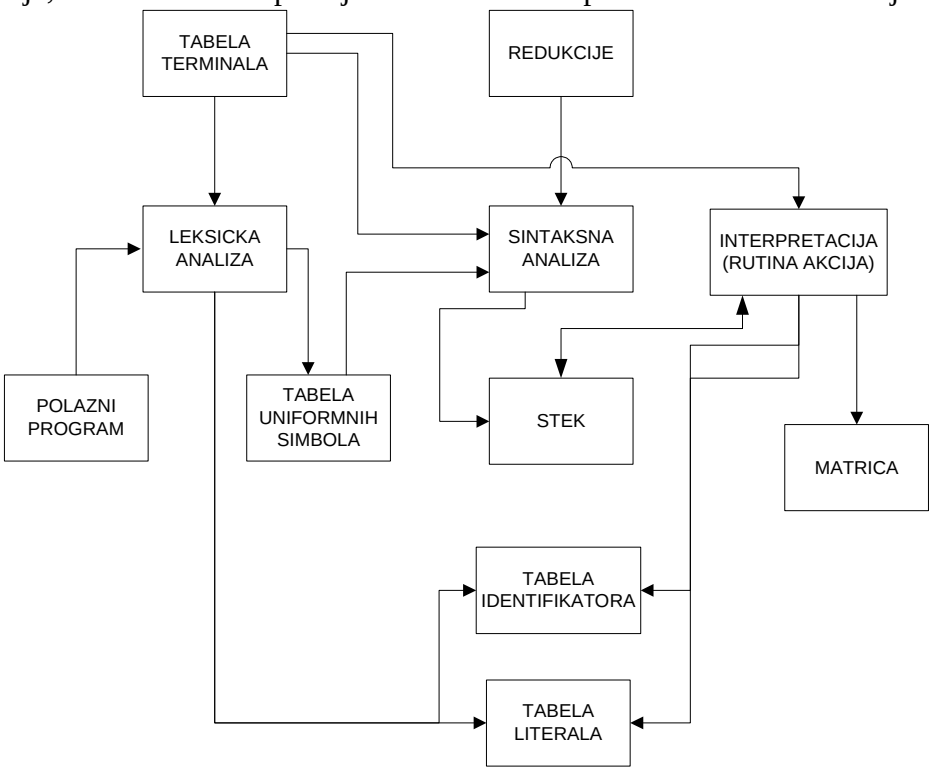
Ulazni podaci za sintaksni analizator su tabela uniformnih simbola, čiji svaki simbol ulazi u stek *(eng. stack)* i tabela redukcija.

Redukcije su sintaksna pravila polaznog jezika i nalaze se u tabeli redukcija. Redukcije upravljaju interpretacijama u fazi sintaksne analize. Opšti oblik redukcija je:

Labela:	Prethodni vrh steka	Rutina akcija	Novi vrh steka	Sledeća redukcija

Tabela 2.1 Opšti oblik redukcija

Labela se definiše proizvoljno, prethodni vrh steka je simbol koji se upoređuje sa tekućim vrhom steka. Ako se prethodni vrh steka poklapa sa tekućim vrhom steka, poziva se rutina akcije, a novi vrh steka postaje tekući vrh steka po završetku rutine akcije.



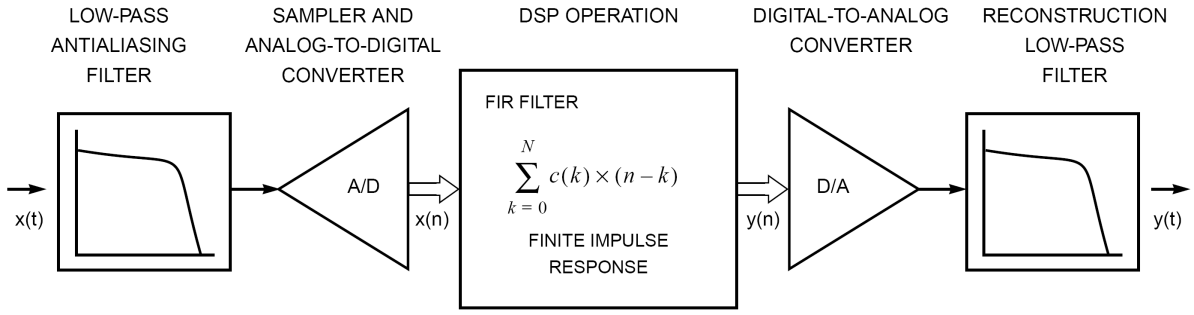
Slika 2.2 Leksička i sintaksna analiza

Algoritam sintaksne faze sastoji se iz ispitivanja redom redukcija, u cilju poklapanja sa poljem prethodnog vrha steka i tekućeg vrha steka sve dok se poklapanje ne pojavi. Kada se pojavi poklapanje poziva se rutina akcije. Po završetku rutine akcije, kontrola se vraća sintaksnom analizatoru, menja se vrh steka, da bi bio saglasan sa poljem novog vrha steka i postupak se ponavlja počevši od redukcje definisane u polju sledeće redukcije. Danas se najčešće koristi opšti sintaksni analizator tipa pomeri/redukuj *(eng. shift/reduce).*

3 Opis fizičke arhitekture i alata

3.1 Opis platforme Motorola 56K

Radi se o digitalnom signal procesoru koji je namenjen za obradu signala u realnom vremenu, kao što su uzorkovanje u tačnim vremenskim intervalima i digitalizacija. Primeri toga su: filtriranje signala, konvolucija (mešanje dva signala), korelacija (poređenje dva signala), čišćenje, pojačanje i transformacija signala.



Slika 3.1. Primer tipične upotrebe DSP procesora

DSP56000 procesor sadrži 3,75K reči u programskom ROMu, dva programirana programska ROMa. Jezgro procesora sadrži tri izvršne jedinice koje rade u paraleli: ALU (aritmetičko-logička jedinica), AGU (jedinica za generisanje adresa) i programski kontroler.

BRZINA – 10,25 miliona instrukcija u sekundi (MIPS), odnosno može da izvrši 1024 tačke kompleksne FFT transformacije za 3,23 ms.

PRECIZNOST – podaci širine 24 bit obezbeđuju 144 dB dinamičkog opsega, a među rezultat sadržan u 56 bit akumulatoru obezbeđuje preko 336 dB.

PARALELIZAM – sve izvršne jedinice (AGU, ALU i programski kontroler), memorija i periferija rade u paraleli. U paraleli može da bude 24 bit x 24 bit množenje, 56 bit dodavanje, dva memorijska prenosa podataka i dva povećanja adresnih pokazivača (linearno, moduo ili sa obratnim prenosom). Sve ovo može da se izvrši u jednom ciklusu rada procesora.

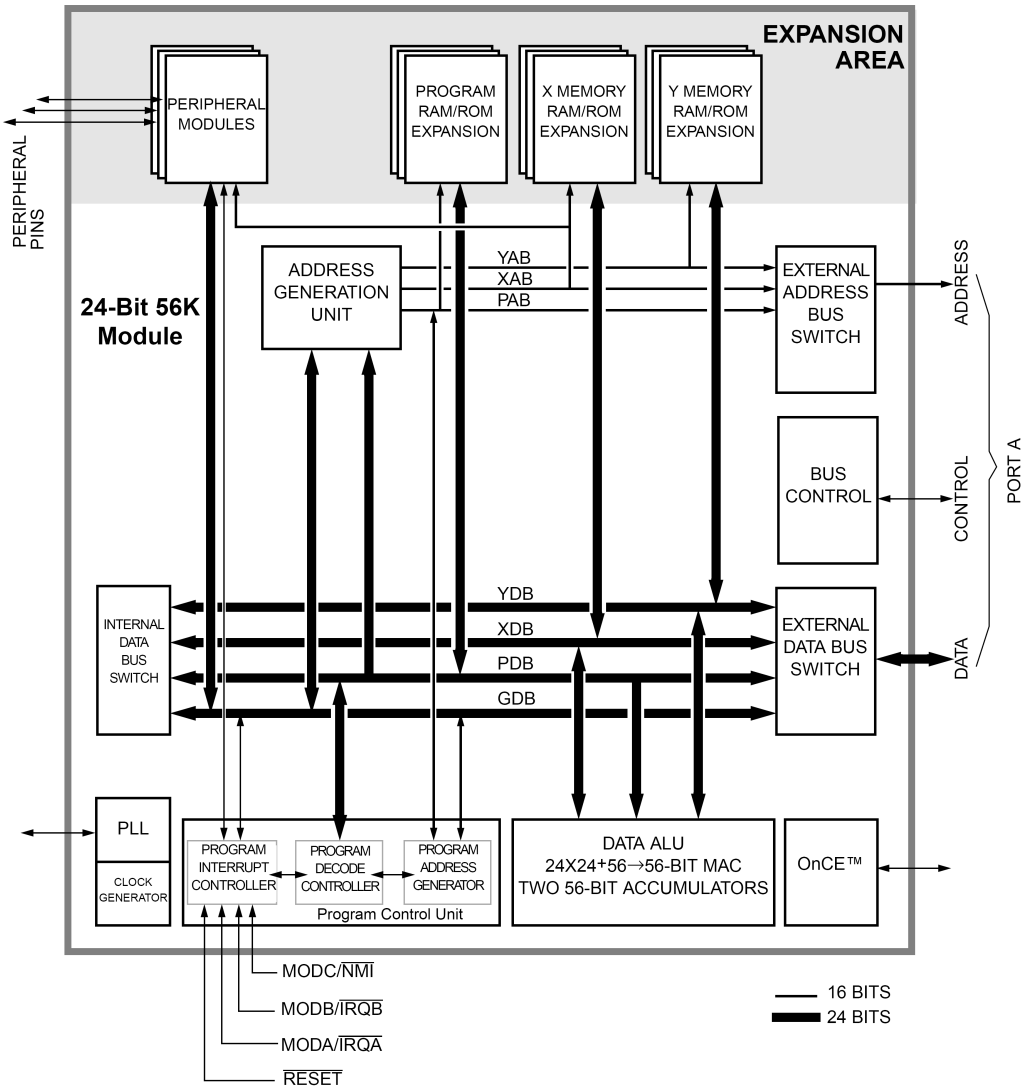
PROTOČNA OBRADA – tri dela obrade instrukcije, nevidljivih za programera.

INSTRUKCIJE – ima 62 instrukcije i sintaksu koja podržava paralelno izvršenje.

3.1.1 Tok podataka

Blok dijagram jezgra je prikazan na slici 4. Jezgro se sastoji od linija podataka, adresnih linija, ALU (*eng. ALU – Arithmetic Logic Unit*), adresnog generatora (*eng. AGU – Address Generation Unit*), X memorijske zone podataka, Y memorijske zone podataka,

programske memorijske zone, programski kontroler i ulaza-izlaza. Adresni generator sadrži osam adresnih 16-bitnih registara (R0-R7) koji služe za generisanje adresa, osam 16-bitnih registara odstojanja (N0-N7) koji služe za povećavanje adresnih pokazivača ili podataka, osam 16-bitnih registara modifikatora (M0-M7) koji služe za određivanje režima adresne aritmetike. U svakom od ovih registra se mogu čuvati i podaci. Tok podataka poseduje četiri 24-bitna registra i dva 56-bitna akumulatora. Svaki akumulator se sastoji iz tri dela, koja se mogu posmatrati kao zasebni registri: A2:A1:A0. A2 registar je 8-bitni, dok su A1 i A0 24-bitna, i svakom od ovih registara može se pristupati nezavisno. ALU je zadužena za sve logičke operacije nad akumulatorima.

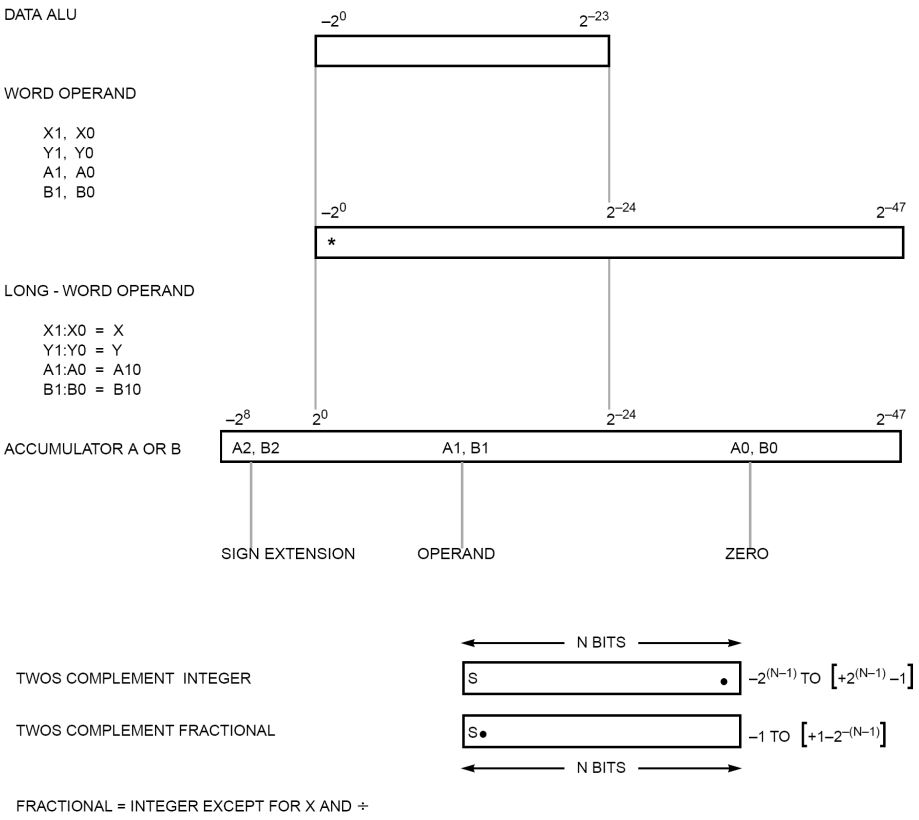


Slika 3.2 Tok podataka u procesoru Motorola 56000

3.1.2 Predstavljanje podataka

Za predstavljanje podataka u ovom procesoru korišćen je komplement dvojke u obliku razlomka. Registri se sastoje od 24 bita, dok akumulatori sadže 56 bita. Sve unutrašnje ALU operacije se obavljaju nad 56-bitnim podacima.

U 24-bitnim registrima levi bit predstavlja znak, iza kojeg se nalazi decimalna tačka, a iza njih se nalazi 23-bitni razlomljeni deo. Najveći pozitivan broj koji je moguće predstaviti je 0x7fffff (decimalno $1-2^{-31}$), dok je najveći negativan broj koji je moguće predstaviti unutar registra 0x800000 (decimalno -1,0).



Slika 3.3 Predstava broja u nepokretnom zarezu

Postoje dve mogućnosti predstavljanja vrednosti unutar akumulatora. U prvom slučaju akumulator se može posmatrati kao 48-bitna vrednost, kod koje je levi bit bit znaka, praćen decimalnom tačkom, iza koje se nalazi 47-bitni razlomljeni deo. U drugom slučaju akumulator se može posmatrati kao 56-bitna vrednost gde se levih osam bita posmatra kao celobrojna vrednost, praćeni decimalnom tačkom, iza koje ide 55-bitna razlomljena vrednost. Najveća pozitivna vrednost koju je moguće predstaviti unutar akumulatora je 0x7f ffffff ffffff (decimalno $256-2^{-63}$), dok je najveća negativna vrednost koju je moguće predstaviti unutar akumulatora 0x80 000000 000000 (decimalno -256,0).

Moguće je učitati 24-bitnu vrednost iz X ili Y memorije u akumulator, i tada se vrednost smešta u A1 ili B1 deo akumulatora, znak se proširuje na A2 ili B2 deo, dok se A0 ili B0 deo postavlja na nulu. U slučaju učitavanja 48-bitne vrednosti, podaci iz X memorije se smeštaju u A1 ili B1 deo akumulatora, iz Y memorije u A0 ili B0 deo, dok se znak vrednosti iz A1 ili B1 dela proširuje na A2 ili B2 deo akumulatora.

3.1.3 Generisanje adresa

Jedinica za generisanje adresa se sastoji od tri grupe registara: osam 16-bitnih adresnih registara; osam 16-bitnih registara odstojanja; osam 16-bitnih registara modifikatora. Adresni registri služe za smeštanje adresa, dok registri odstojanja služe za povećavanje adresa u adresnim registrima. Šema adresnog generatora data je na slici 5.

Adresni registri i registri odstojanja i modifikatora zajedno učestvuju u generisanju adrese prilikom indirektnog pristupa memorijskim lokacijama. Adresni režim koji se

primenjuje zavisi od sintakse instrukcije za pristup memoriji. Mogući su sledeći adresni režimi:

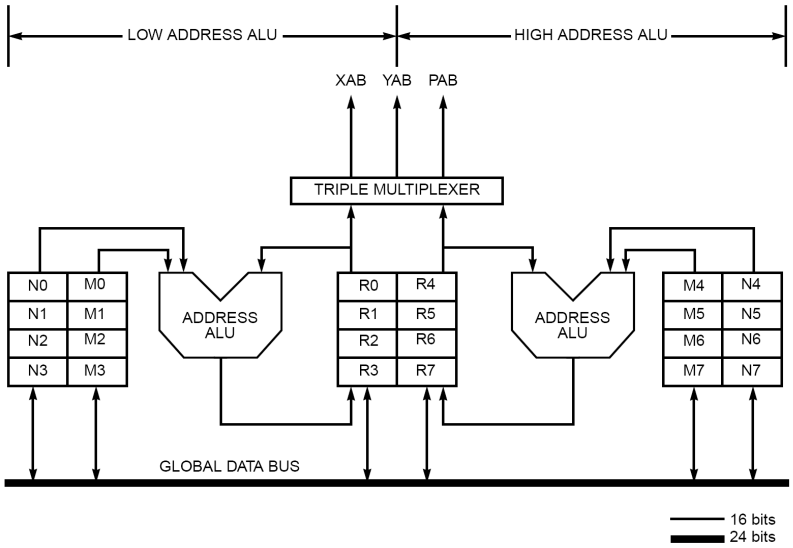
- bez izmene adresnog registra
- sa povećanjem vrednosti adresnog registra nakon pristupa memoriji za 1
- sa smanjenjem vrednosti adresnog registra nakon pristupa memoriji za 1
- sa povećanjem vrednosti adresnog registra nakon pristupa memoriji za sadržaj registra odstojanja
- sa smanjenjem vrednosti adresnog registra nakon pristupa memoriji za sadržaj registra odstojanja
- pristup memorijskoj lokaciji čija je adresa zbir adresnog i registra odstojanja bez izmene adresnog registra
- sa smanjenjem vrednosti adresnog registra pre pristupa memoriji za 1

Vrednost modifikator registra određuje jedan od režima adresiranja:

- linearni
- moduo
- i režim sa obratnim prenosom

Ovi adresni režimi pružaju mogućnost jednostavne uoptrebe raznih struktura podataka kao što su FIFO, kružni baferi, stekovi, bit reverzni FFT baferi.

Postojanjem dve ALU u okviru AGU bloka postiže se paralelno izvršavanje. U jednom instrukcijskom ciklusu moguće je generisati dve adrese. Na primer za X, Y ili P memorijsku zonu. Registri su podeljeni u dve grupe R0-R3 ide u kombinaciji sa registrima N0-N3 i M0-M3, a druga grupa R4-R7 ide u kombinaciji sa registrima N4-N7 i M4-M7.



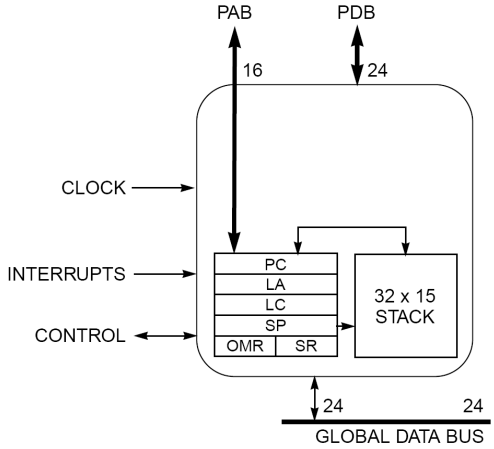
Slika 3.4 Adresni generator

Prilikom operacija upisa u memoriju i čitanja iz memorije, postoji mogućnost paralelnog izvršenja mašinskih instrukcija. Instrukcije u paraleli izvršavaju se istovremeno. Jedna instrukcijska reč kodira se sa jednom ili dve 24-bitne memorijske reči. U jednom skupu paralelnih instrukcija može da se spoji najviše 3 mašinskih instrukcija. Istovremeno može da

se obavi jedna operacija sa memorijskom zonom X i jedna operacija sa memorijskom zonom Y i jedna ALU operacija sa specijalnim rasporedom indeksa resursa.

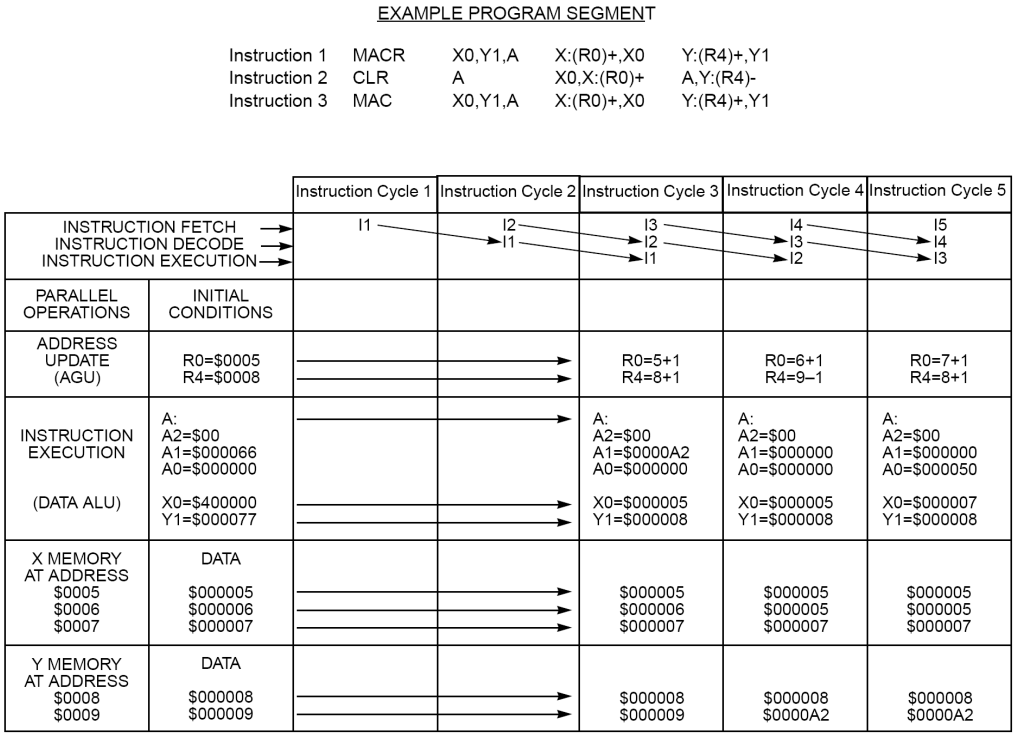
3.1.4 Blok za kontrolu toka programa

Blok za kontrolu toka programa sastoji se od šest registara i registra adrese sistemskog steka (SS). Registri su: programski brojač (*eng. Program Counter – PC*), status registar (*eng. Status Register – SR*), sistemski stek, programski kontrolni registri: registar adrese petlje (*eng. Loop address – LA*) registar brojača prolaza kroz petlju (*eng. Loop Counter – LC*) i registar pokazivača na stek (*eng. Stack Pointer – SP*).



Slika 3.5 Programski kontroler

Programski brojač je 16-bitni pokazivač čija je namena da pokazuje na narednu instrukciju, koju treba dobaviti iz programske memorije. Stek podprograma se koristi za smeštanje povratne vrednosti programskog brojača pri pozivu podprograma. Dugačak je 32 bita, i sastoji se od 16 lokacija. Implementiran je kao kružni bafer. Petlje takođe koriste ovaj stek da sačuvaju stanje petlje pri pozivu nove petlje.

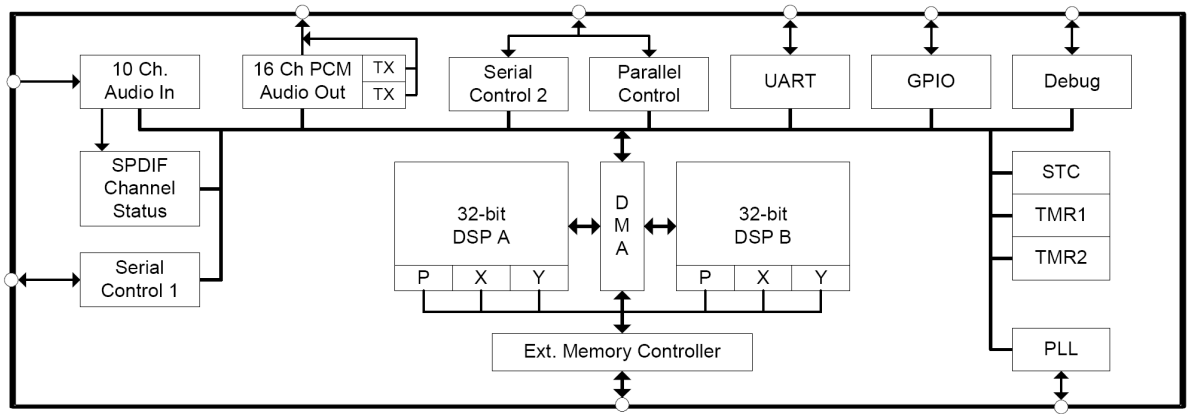


Slika 3.6 Protočna struktura izvršavanja mašinskih instrukcija

Dubina protočne obrade je tri instrukcije. Izvršenje mašinske instrukcije se odvija u tri faze. Prva faza je pribavljanje instrukcije (*eng. fetch*), druga je dekodiranje instrukcije (*eng. decode*) i treća faza je izvršenje instrukcije (*eng. execute*).

3.2 Opis platforme Coyote Cirrus Logic

Radi se o procesorskom jezgru, koji u sebi integriše dva 32-bitna procesora klase DSP, kao i DMA *(eng. Direct Memory Access)* kontroler, uz puni opseg audio periferija. CS495XX poseduje odvojene memorijske zone za podatke X i Y, kao i posebnu programsku memorijsku zonu P. Svako od jezgara je 32-bitni DSP projektovan za rad u nepokretnom zarezu, koji su u stanju da u jednom ciklusu izvrše dva nezavisna pristupa memoriji *(eng. Memory Access Control)*. Svako jezgro poseduje osam 72-bitnih akumulatora, kao i osam 32-bitnih registara, 4 X-data i 4 Y-data registra, i 12 indeksnih registara, koji služe za pristup memoriji. Takođe postoji objedinjeni pristup memorijskim lokacijama X i Y, zona XY ili L zona, širine reči 64-bita.

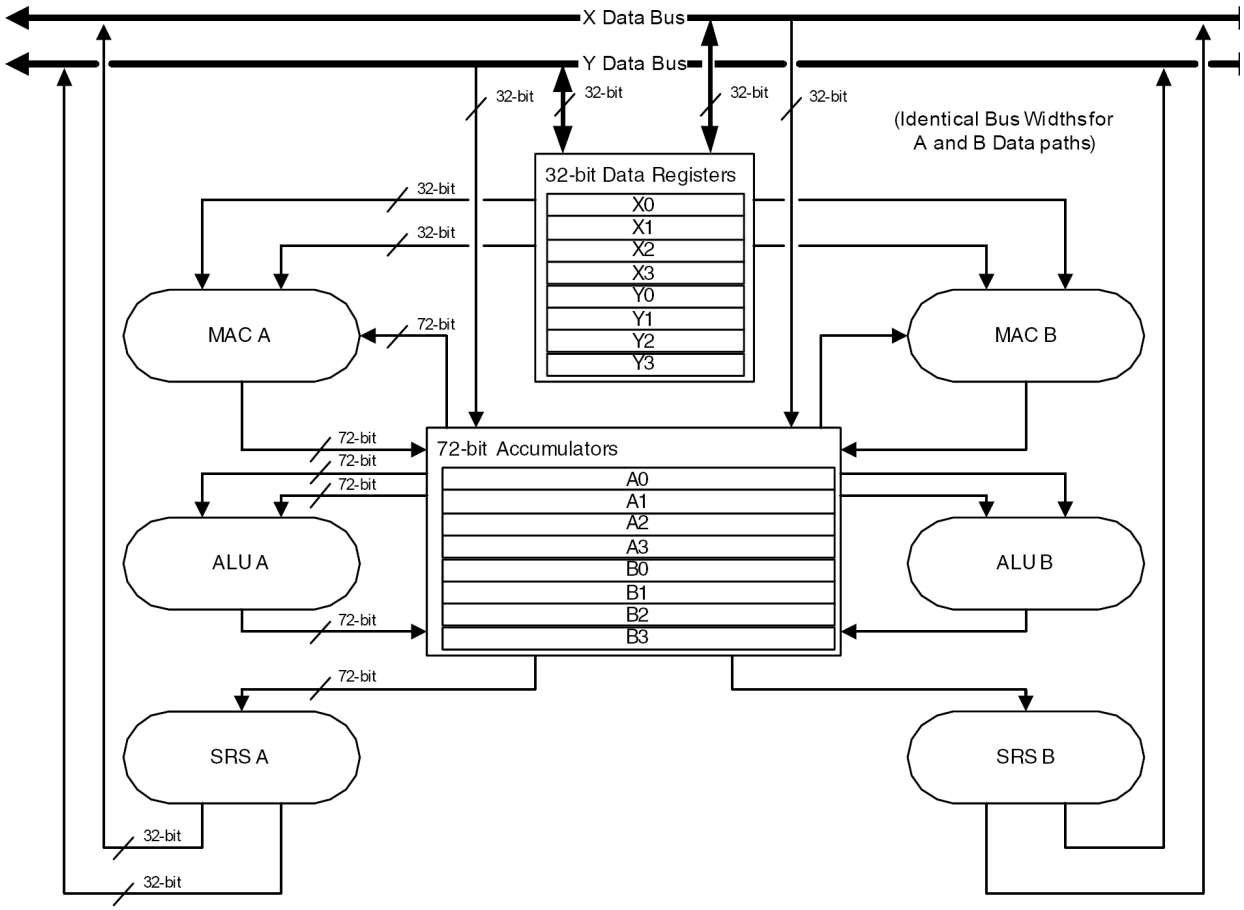


Slika 3.7 Struktura Coyote CS495XX procesora

Oba DSP jezgra su vezana za prilagodljiv DMA kontroler. DMA kontroler služi za prenos podataka između periferija, kao što su DAI i DAO *(eng. Digital Audio Input i Digital Audio Output)*, spoljne memorije ili bilo koje DSP memorije, sve bez posredovanja samih DSP jezgara. Ovo dovodi do rasterećenja DSP jezgra i ostavlja više MIPS-a za obradu audio signala.

3.2.1 Tok podataka

Blok dijagram jezgra je prikazan na slici 3.8. Jezgro se sastoji od kontrolne jedinice, paralelnih adresnih generatora (*eng. AGU – Address Generation Units*) i paralelnih putanja podataka (A i B). Adresni generatori sadrže osam 16-bitnih registara koji služe za generisanje adresa, kao i osam 16-bitnih registara koji u sprezi sa indeksnim registrima služe za odabiranje režima adresiranja. Svaki tok podataka poseduje 4 32-bitna registra i 4 72-bitna akumulatora, tako da sam procesor poseduje osam registara i osam akumulatora. Svaki akumulator se sastoji iz tri dela, koja se mogu posmatrati kao zasebni registri: **Guard**, **High**, **Low**. **Guard** registar je 8-bitni, dok su **High** i **Low** oba 32-bitna, i svakom od ovih registara se može pristupati nezavisno. Svaka putanja podataka takođe sadrži po jednu MAC (*eng. Multiply and Accumulate*) jedinicu, jednu SRS (*eng. Shift Round Saturate*) jedinicu, i jednu aritmetičko-logičku jedinicu (*eng. ALU – Arithmetical-Logic Unit*). ALU je zadužena za sve logičke operacije nad akumulatorim dok se SRS brine o prekoračenjima opsega pri raznim izračunavanjima.

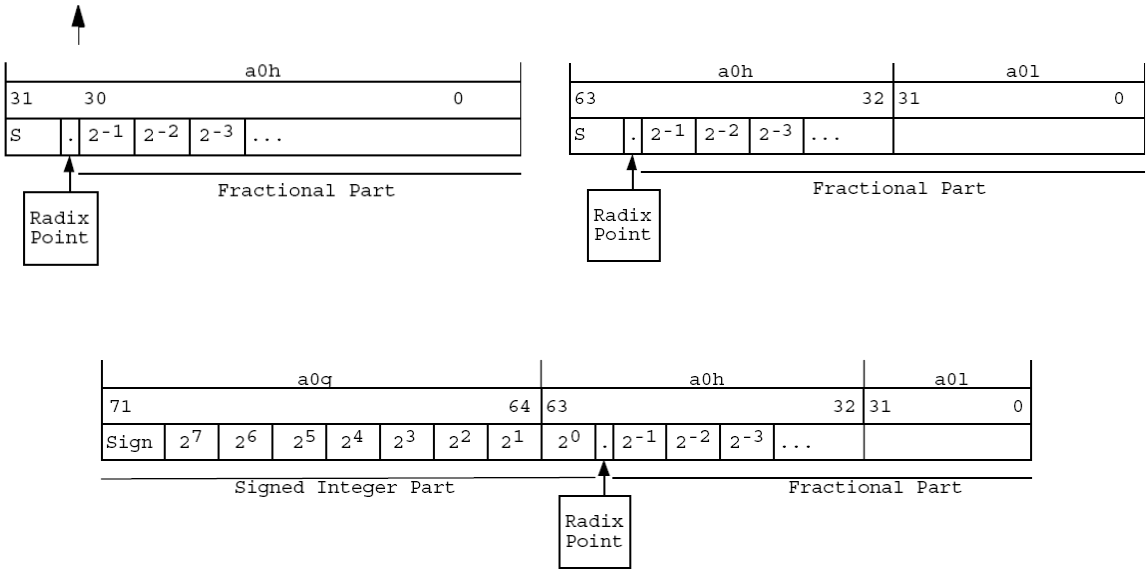


Slika 3.8 Tok podataka u procesoru CS495XX

3.2.2 Predstavljanje podataka

Za predstavljanje podataka u ovom procesoru korišćen je komplement dvojke u obliku razlomka. Registri se sastoje od 32 bita, dok akumulatori sadže 72 bita, koji mogu da se pročitaju kroz SRS kao 32-bitni ili 64-bitni podaci. Sve interne ALU operacije se izvršavaju nad 72-bitnim podacima.

U 32-bitnim registrima levi bit predstavlja znak, iza kojeg se nalazi decimalna tačka, a iza njih se nalazi 31-bitni razlomljeni deo. Najveći pozitivan broj koji je moguće predstaviti je 0x7fff ffff (decimalno $1-2^{-31}$), dok je najveći negativan broj koji je moguće predstaviti unutar registra 0x8000 0000 (decimalno –1,0).



Slika 3.9 Predstava broja u pokretnom zarezu

Postoje dve mogućnosti kod predstavljanja vrednosti unutar akumulatora. U prvom slučaju akumulator se može posmatrati kao 64-bitna vrednost, kod koje je levi bit bit znaka, praćen decimalnom tačkom, iza koje se nalazi 63-bitni razlomljeni deo. U drugom slučaju akumulator se može posmatrati kao 72-bitna vrednost gde se levih osam bita posmatra kao celobrojna vrednost, praćeni decimalnom tačkom, iza koje ide 63-bitna razlomljena vrednost. Najveća pozitivna vredonst koju je moguće predstaviti unutar akumulatora je 0x7f ffff ffff ffff ffff (decimalno $256-2^{-63}$), dok je najveća negativna vrednost koju je moguće predstaviti unutar akumulatora 0x80 0000 0000 0000 0000 (decimalno -256,0).

Moguće je učitati 32-bitnu vrednost iz X ili Y memorije u akumulator, i tada se vrednost smešta u **High** deo akumulatora, znak se proširuje na **Gurad** deo, dok se **Low** deo postavlja na nulu. U slučaju učitavanja 64-bitne vrednosti, podaci iz X memorije se smeštaju u **High** deo akumulatora, iz Y memorije u **Low** deo, dok se znak vrednosti u **High** delu proširuje na **Guard** deo akumulatora.

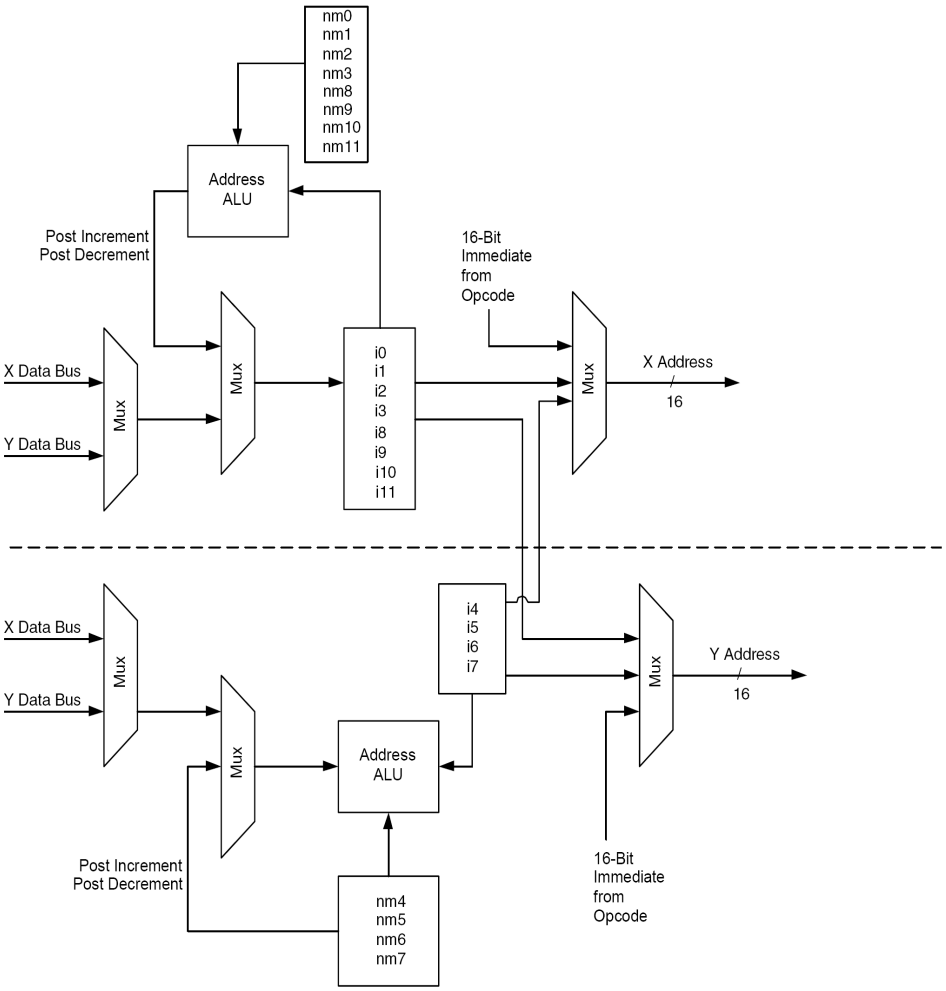
Čitanje podataka iz akumulatora se izvodi kroz SRS jedinicu, koja postoji za oba toka podataka, A i B. SRS jedinica obavlja zadatke pomeranja *(eng. shift)*, zaokruživanja *(eng. round)* i zasićenja *(eng. saturate)* u tom redosledu. Do pomeranja, zaokruživanja i zasićenja dolazi jedino u slučaju da se vrednost iz akumulatora čita u memoriju ili u registre, dok vrednosti unutar akumulatora ostaju nepromenjene. Pomeranje, zasićenje i zaokruživanje se reguliše vrednošću specijalnog regirstra CCR.

3.2.3 Generisanje adresa

Jedinica za generisanje adresa se sastoji od dva skupa registara. Jednu grupu čini 12 16-bitnih indeksnih registara, dok drugu grupu čini 12 16-bitnih moduo registara. Indeksni registri služe za smeštanje adresa, dok moduo registri služe za definisanje različitih režima adresiranja. Šema adresnog generatora je data na slici 3.10.

Visok nivo paralelnog izvršenja mašinskih instrukcija dobija se adresiranjem memorije indeksnim registrima *i0* i *i1* kod operacija sa X memorijskom zonom i registara *i4* i

i5 kod operacija nad Y memorijskom zonom. Za potrebe operativnog sistema koriste se indeksni registri *i8, i9, i10* i *i11*.



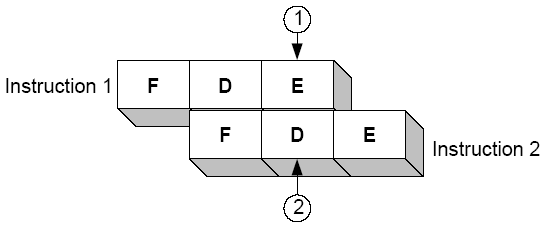
Slika 3.10 Adresni generator

Prilikom operacija upisa i čitanja u i iz memorije, postoji mogućnost paralelnog izvršenja mašinskih instrukcija. Paralelizacija obuhvata sve operacije sem postavljanja sistemskih registara, upisa konstanti i neposredni upis vrednosti. Instrukcije koje su paralelizovane izvršavaju se istovremeno. Ovakvo izvršenje predstavlja jednu 32-bitnu procesorsku reč. U jednom skupu paralelnih instrukcija može da se spoji najviše 6 mašinskih instrukcija. Istovremeno može da se obavi jedno čitanje iz memorijske zone X i jedno čitanje iz memorijske zone Y, dve MAC operacije sa specijalnim rasporedom indeksa resursa i dva inkrementa različitih indeksnih registara.

3.2.4 Kontrola toka programa

Kontrola toka programa se sastoji od programskog brojača *(eng. Program Counter-PC)*, dva sistemska steka, i dva kontrolna registra: MR i CCR. MR je moduo registar *(eng. Moduo Register)*, a CCR je registar koda uslova *(eng. Code Condition Register)*.

Programski brojač je 16-bitni pokazivač čija je namena da pokazuje na narednu instrukciju, koju treba dobaviti iz programske memorije. Stek podprograma se koristi za smeštanje povratne vrednosti programskog brojača pri pozivu podprograma. Dugačak je 16 bita, i sastoji se od 16 lokacija. Implementiran je kao kružni bafer. Stek petlje se koristi da se sačuva stanje petlje pri pozivu nove petlje. Dugačak je 48 bita, i implementiran je kao kružni bafer od 8 lokacija.



Slika 3.11 Protočna struktura izvršavanja mašinskih instrukcija

Dubina protočne strukture izvršenja je tri instrukcije. Izvršenje mašinske intrukcije čitanje odvija se u tri faze. Prva faza je pribavljanje instrukcije (*eng. fetch*), druga je dekodiranje instrukcije (*eng. decode*) i treća faza je izvršenje instrukcije (*eng. execute*).

3.3 Alati programskih prevodioca

Kako bi se olakšala implementacija prednjeg dela programskih prevodioca, razvijen je skup alata koji služi za pisanje sintaksnih i leksičkih analizatora. Ovi alati su najčešće osmišljeni tako da se međusobno nadovezuju. Primer ovakvih alata su generatori leksičkih i sinstaksnih analizatora.

3.3.1 Generator leksičkog analizatora

Jedan popularan alat za pisanje leksičkog analizatora je FLEX (*eng. Fast Lexical Analyzer Generator*). FLEX generiše automat sa konačnim brojem stanja, koji prepoznaje osnovne simbole prevođenog jezika i prosleđuje ih u vidu tokena sintaksnom analizatoru.

FLEX automat definiše se dozvoljenim skupom pravila sadržanih u ulaznoj datoteci. Struktura ulazne datoteke opisana je sledećom formom:

```
{definicije}
%%
{pravila}
%%
{korisničke rutine}
```

Pravila i definicije zadaju se uz pomoć sledećih operatora:

```
x         znak "x"
"x"       znak "x", čak i kad je x operator
\x        znak "x", čak i kad je x operator
[xy]      znak x ili y
[x-z]     znak x, y ili z
[^x]      bilo koji znak osim x
.         bilo koji znak osim nove linije
^x        x na početku linije
<y>x      x kada je automat u početnom stanju y
x$        x na kraju linije
x?        opciono x
x*        0,1,2, ... pojava x
x+        1,2,3, ... pojava x
x|y       x ili y
(x)       x
x/y       x pre y
{xx}      zamena xx njenom prethodnom definicijom
x{m,n}    od m do n pojava x
```

Primeri nekih definicija:

```
DEC_DIGIT       [0-9]
BIN_DIGIT       [0-1]
HEX_DIGIT       [0-9a-fA-F]
DEC_NUMBER      {DEC_DIGIT}+
HEX_NUMBER      "$"{HEX_DIGIT}+
```

Primeri nekih pravila:

```
{DEC_NUMBER}                          { return (NUMBER); };
{HEX_NUMBER}                          { return (HEX_TK); };
{DEC_DIGIT}*"."{DEC_DIGIT}+           return FLOAT_TK;
{DEC_DIGIT}*"."{DEC_DIGIT}+{EXPONENT} return FLOAT_TK;
{DEC_DIGIT}+{EXPONENT}                return FLOAT_TK;
```

Nakon obrade ulazne datoteke, FLEX generiše automat u obliku programa u jeziku C, koji se zatim može prevesti u izvršnu datoteku na željenoj arhitekturi.

3.3.2 Generator sintaksnog analizatora

Jedan od alata za generisanje sinstaksnog analizatora koji se prirodno nadovezuje na FLEX je YACC (*eng. Yet Another Compiler Compiler*). YACC generiše sistem konačnog automata sa stekom u obliku programa u jeziku C. Automat se generiše na osnovu sintaksnih pravila zadatih u ulaznoj datoteci. Ulazna datoteka ima sledeću formu:

```
Uvodne deklaracije
%%
Sintaksna pravila
%%
Dodatne rutine
```

Uvodne deklaracije i dodatne rutine pišu se u programskom jeziku C i prenose se neizmenjene u generisani program. Pravila za prepoznavanje sintakse zadaju se na sledeći način:

```
naziv_pravila :     pravilo1
                    akcija1
                    | pravilo2
                    akcija2
```

Konkretno pravilo definiše se koristeći osnovne simbole jezika dobijene iz leksičkog analizatora i već ranije definisanih sintaksnih pravila rekurzivno:

```
<simbol1/pravilo1> <simbol2/pravilo2>  ... <simboln/pravilon>
```

Zadatak leksičkog analizatora je da iz dobijenih simbola prepoznaje sintaksna pravila i izvršava zadatu akciju. Akcija se takođe piše u programskom jeziku C i najčešće realizuje konkretno povezivanje sa sledećom komponentom programskog prevodioca. Ova veza ostvaruje se preko ulančavanja određene, sada već apstraktne strukture u odgovarajuće liste.

Operacije koje izvodi sintaksni analizator nad ulaznim simbolima leksičkog analizatora su sledeće: pomeri (*eng. shift*), redukuj (*eng. reduce*), prihvati (*eng. accept*) i greška (*eng. error*).

Operacija pomeri prihvata nove simbole i proverava da li postoji sintaksno pravilo u koje se oni uklapaju. Ukoliko se na ulazu pojavi neočekivani simbol, koji se ne uklapa ni u jedno pravilo, izvršava se operacija greška.

Operacija redukuj izvršava se ukoliko se pronađe prvo potpuno poklapanje sa pravilom, kad se izvršava konkretna akcija zadata za to pravilo.

Operacija prihvati izvršava se ukoliko je ispravno prepoznata cela ulazna datoteka zadatim pravilima i proglašava sintaksnu analizu uspešno završenom.

Operacijom greška može se implementirati mehanizam za opis konkretne nastale greške krajnjem korisniku programskog prevodioca, u zavisnosti od mesta na kom se greška pojavila.

Prilikom pisanja pravila, treba voditi računa da ne postoje preklapanja. Generator sintaksnog analizatora vrši proveru ispravnosti gramatike. Može doći do dve vrste sukoba:

- Pomeri/redukuj (*eng. shift/reduce*)
- Redukuj/redukuj (*eng. reduce/reduce*)

Pomeri/redukuj se javlja kada sintaksni analizator ne zna da li da nastavi sa prihvatanjem simbola ili da redukuje pravilo, dok se redukuj/redukuj javlja ukoliko postoje pravila definisana istim nizom simbola. U ovakvim situacijama neophodno je definisati sintaksna pravila na drugačiji način.

4 Realizacija

Zadatak realizacije prednjeg dela krosasemblera je izrada sintaksnog i leksičkog analizatora upotrebom alata za njihovo generisanje, FLEX i YACC.

4.1 Leksički analizator

Početak realizacije rešenja je definisanje izraza leksičkog analizatora ulaznog formata asemblera DSP procesora Motorola 56000. Zadatak leksičkog analizatora je da prepozna pojedinačne simbole ulazne datoteke i prosledi ih u odgovarajućem redosledu sintaksnom analizatoru. Cilj je prepoznati sledeće skupove simbola asemblerskog jezika procesora Motorola 56000:

- heksadecimalni, decimalni i brojevi u pokretnom zarezu
- matematički operatori
- ugrađene asemblerske funkcije
- direktive za dodelu struktura podataka
- registri
- skup instrukcija

Prepoznavanje svakog simbola opisano je odgovarajućim operatorima u ulaznoj dadoteci FLEX alata.

Neposredne brojne vrednosti definisane su sledećim izrazima. Prvo su zadate pomoćne definicije, iz kojih su kasnije izvedeni složeniji izrazi.

```
DEC_DIGIT                                   [0-9]
BIN_DIGIT                                   [0-1]
HEX_DIGIT                                   [0-9a-fA-F]
SIGN                                        [+-]
EXPONENT                                    [eE][+-]?{DEC_DIGIT}+
DEC_NUMBER                                  {DEC_DIGIT}+
HEX_NUMBER                                  "$"{HEX_DIGIT}+
BIN_NUMBER                                  "%"{BIN_DIGIT}+

{DEC_NUMBER}                                return NUMBER;
{HEX_NUMBER}                                return HEX_TK;
{DEC_DIGIT}*"."{DEC_DIGIT}+                 return FLOAT_TK;
{DEC_DIGIT}*"."{DEC_DIGIT}+{EXPONENT}       return FLOAT_TK;
{DEC_DIGIT}+{EXPONENT}                      return FLOAT_TK;
```

Matematički operatori se zamenjuju odgovarajućim simbolima:

```
"+"             return PLUS;
"-"             return MINUS;
```

```
"`"             return COMPLE;
"!"             return NOTTK;
"*"             return ASTER;
{SLASHTK}       return SLASH;
"%"             return MOD;


"<"             return LT;
"<="            return LE;
">"             return GT;
">="            return GE;
"=="            return ASG;
"!="            return NOTEQU;

"&"             return BAND;
"|"             return BOR;
"^"             return UPARR;

"&&"            return AND;
"||"            return OR;

"<<"            return LSHIFT;
">>"            return RSHIFT;

","             return ',';
"("             return '(';
")"             return ')';
":"             return ':';
```

Prepoznavanje registara obavljaju sledeći izrazi:

```
x[0-1]|y[0-1]     { strcpy(regname,yytext); return INPUT_REG_X_Y; }

a|b               { strcpy(regname,yytext); return ACCU_A_B; }
a[0-2]|b[0-2]     { strcpy(regname,yytext); return ACCU_REG_A_B; }
a10|b10           { strcpy(regname,yytext); return ACCU_REG_A10_B10; }
ab|ba             { strcpy(regname,yytext); return ACCU_REG_AB_BA; }

r[0-7]            { strcpy(regname,yytext); return ADDR_REG; }
n[0-7]            { strcpy(regname,yytext); return OFFSET_REG; }
m[0-7]            { strcpy(regname,yytext); return MODIF_REG; }
```

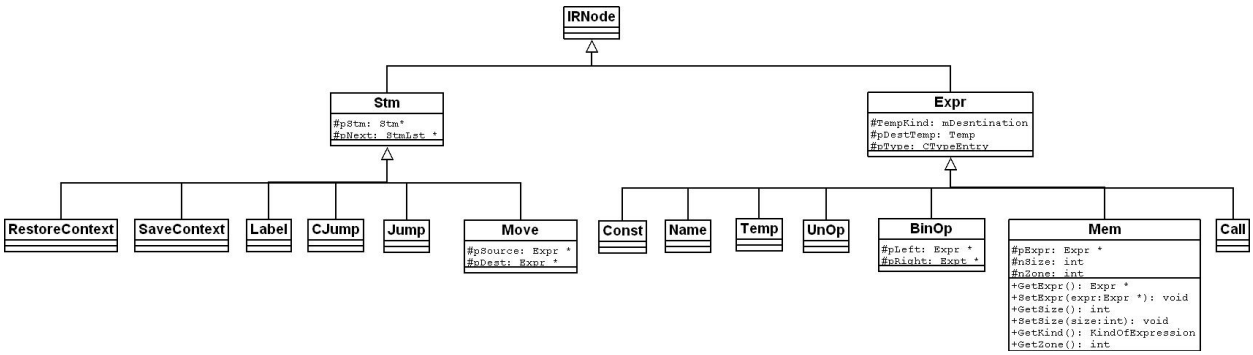
Prepoznavanje ugrađenih asemblerskih funkcija, direktiva i skup instrukcija izdvojeno je u posebnim ručno napisanim rutinama, kako bi se uprostio sam automat sintaksnog analizatora. Njihovo prepoznavanje obavlja se kada generisani automat ne prepozna niz ulaznih znakova. Ukoliko ni ove rutine ne prepoznaju niz znakova iz jednog od ova tri skupa, ovaj niz znakova se prosleđuje kao simbol identifikatora.

Niz znakova nakon znaka „;“ smatraju se komentarom ulaznog programa i dalje se odbacuju.

4.2 Sintaksni analizator

Zadatak realizacije sintaksnog analizatora je definisanje pravila prepoznavanja iskaza asemblera procesora Motorola 56000 kao i prevođenje iskaza u oblik stabla među-reprezentacije (*eng. Intermediate Representation Tree*).

Osnovna klasa kojom se modeluje stablo među-reprezentacije je *IRNode*. Nju nasleđuju klase *Stm* i *Expr*. Klasa *Stm* modeluje iskaz (*eng. Statement*), a klasa *Expr* modeluje izraz (*eng. Expression*) u jeziku među-reprezentacije. Na slici 4.1 dat je pregled klasa jezika međureprezentacije. Stablo među-reprezentacije je ulaz za sledeću fazu programskog prevođenja.



Slika 4.1 Model klasa jezika međureprezentacije kompajlera

Klase *Label*, *CJump*, *Jump* i *Move* naslednice su klase *Stm*. Klasom *Label* obeležava se polozaj labele u programu, klasama *CJump* i *Jump* modeluju se uslovni skok i skok kontrole toka izvršenja programa. Klasa *Move* predstavlja osnovni iskaz dodele.

Klase *Const*, *Name*, *Temp*, *UnOp*, *BinOp, Mem i Call* naslednice su klase *Expr*. Klasa *Const* modeluje neposrednu vrednost u izrazu; klasa *Name* modeluje promenljivu; klasa *Mem* modeluje pristup memorijskoj lokaciji; klasa *Temp* modeluje privremenu promenljivu; klase *UnOp* i *BinOp* modeluju unarne i binarne operacije; klasa *Call* modeluje poziv podrutine.

Realizacija sintaksnog analizatora počinje definisanjem sintaksnih pravila. Posmatrajući iz problema realizacije sintaksnog analizatora, asemblerska datoteka sastoji se iz niza linija. Sintaksno pravilo za prepoznavanje linije opisano je rekurzivnom definicijom na sledeći način:

```
input:  /*empty*/
     | input line
```

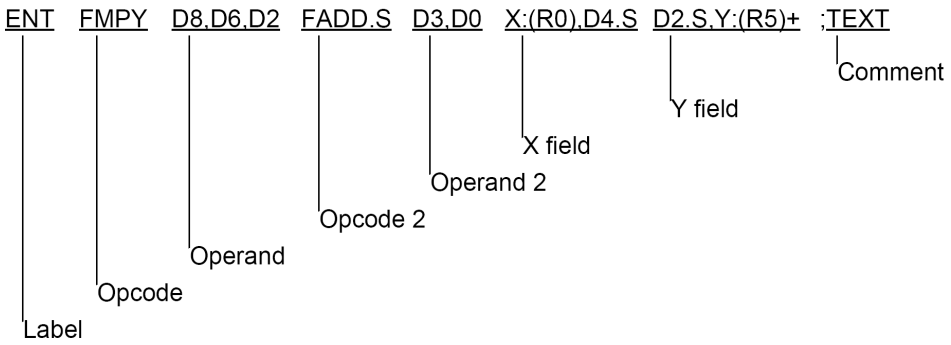
Prva linija obezbeđuje da sintaksni analizator prihvata i praznu ulaznu datoteku, dok je u drugoj liniji rekurzivnim mehanizmom obezbeđena sekvencijalna analiza cele datoteke.

Asemblerska datoteka sastoji se od asemblerskih direktiva i asemblerskih iskaza što je opisano pravilom *line*:

```
line:
   label '\n'
   | label  ' ' instruction '\n'
   | ' ' instruction '\n'
   | symbol_def '\n'
```

```
  | data_def_storage_allocation '\n'
  | assembly_control '\n'
  | '\n'
  | ' ' '\n'
```

Ovim pravilom obezbeđeno je prepoznavanje direktiva za kontrolu rada asemblera, dodele memorijskih struktura, definicije simbola i skupa instrukcija. Zadatak ovog rada obuhvata realizaciju dela sintaksnog analizatora iskaza sa pristupom memorijskim lokacijama preko opisa ponašanja jedinice za generisanje adresa. Format iskaza dat je na slici 4.2.



Slika 4.2: Format iskaza asemblerskog jezika procesora Motorola 56000

Iskaz započinje opcionom labelom. Labele su pomoćni simboli koji se prilikom asembliranja prevode u memorijske lokacije. Odgovarajuća predstava labele ulančava se u stablo među-reprezentacije.

Simbol instrukcije (*eng. opcode*) zajedno sa operandima čini instrukciju čije se ponašanje opisuje odgovarajućim podstablom među- reprezentacije.

X i Y polje opisuju memorijske prenose u ili iz odgovarajućih memorijskih zona koji se obavljaju istovremeno sa izvršenjem instrukcije.

Pravilo *instruction* podeljeno je na sledeće grupe instrukcija:

```
instruction:
            arith_instr             // arithmetic instructions rules
            | log_instr             // logical instructions rules
            | bit_manip_instr       // bit manipulation instruction rules
            | loop_instr            // loop instructions rules
            | move_instr            // move instructions rules
            | prog_control_instr    // flow control instructions rules
            | par_move_instr        // instructions with parallel move rules
            | par_move_instr   parallel_move
            | par_move_instr   parallel_move   parallel_move
```

Instrukcije koje uključuju u rad jedinicu za generisanje adresa spadaju u grupu instrukcija za pristup memorijskim lokacijama. Ove instrukcije mogu se podeliti u više grupa:

- Neposredan pristup memorijskim lokacijama
- Posredan pristup memorijskim lokacijama
- Izmena sadržaja adresnih registara bez pristupa memorijskoj lokaciji

4.2.1 Neposredan pristum memorijskim lokacijama

Neposredan pristup memorijskoj lokaciji podrazumeva da je adresa memorijske lokacije unapred zadata u telu instrukcije absolutnom vrednošću ili labelom. Za ovakve instrukcije generiše se odgovarajući čvor među-reprezentacije za pristup memorijskoj lokaciji.

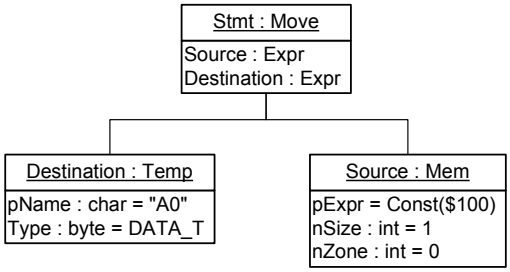
```
| ' ' zone ':' exp ',' reg_all
| ' ' reg_all ',' zone ':' exp
```

Pravilo *zone* obuhvata sve memorijske zone, pravilo *exp* izračunava vrednost izraza koji u ovom slučaju predstavlja memorijsku lokaciju, dok pravilo *reg_all* obuhvata sve moguće kombinacije registara.

Primeri ovakvih instrukcija su:

```
MOVE X:$100,A0
MOVE A0,X:$100
MOVE X:$100+5,A0
MOVE X:TEMP,A0
```

Prvi navedeni primer ima logičko značenje punjenja akumulatora A0 vrednošću memorijske lokacije x:$100 i u jeziku među-reprezentacije predstavljen je sledećom strukturom objekata na slici 4.3:



Slika 4.3: Razvijeno stablo među-reprezentacije

4.2.2 Posredan pristup memorijskim lokacijama

Posredan pristup memorijskim lokacijama pomoću adresnih registara obuhvata više različitih adresnih režima:

- bez izmene adresnog registra
- sa povećanjem vrednosti adresnog registra nakon pristupa memoriji za 1
- sa smanjenjem vrednosti adresnog registra nakon pristupa memoriji za 1
- sa povećanjem vrednosti adresnog registra nakon pristupa memoriji za sadržaj registra odstojanja
- sa smanjenjem vrednosti adresnog registra nakon pristupa memoriji za sadržaj registra odstojanja
- pristup memorijskoj lokaciji čija je adresa zbir adresnog i registra odstojanja bez izmene adresnog registra
- sa smanjenjem vrednosti adresnog registra pre pristupa memoriji za 1

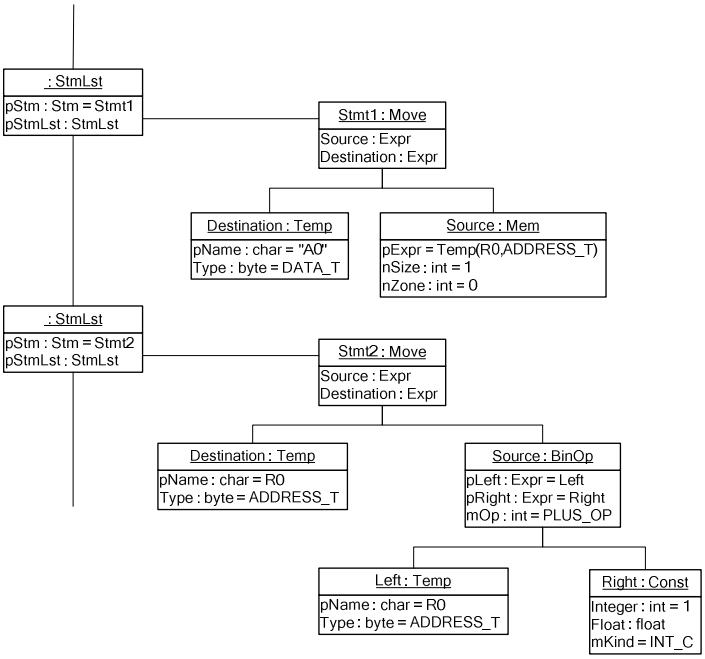
Ovi režimi pokriveni su sledećim sintaksnim pravilima i primerima obuhvaćenih instrukcijskih reči:

```
| ' ' zone ':' '(' reg_abxy ')' sign_eff   eff_reg_abxy ',' reg_all
| ' ' reg_all ',' zone ':' '(' reg_abxy ')' sign_eff   eff_reg_abxy

X:(R0),A0
X:(R0)+,A0
X:(R0)-,A0
X:(R0)+N0,A0
X:(R0)-N0,A0
A0,X:(R0)
A0,X:(R0)+
A0,X:(R0)-
A0,X:(R0)+N0
A0,X:(R0)-N0
```

Kao što se vidi iz primera, ovim pravilima pokriveno je više različitih adresnih režima pristupa memorijskim lokacijama. Osim za prvi slučaj u kojem se ne javlja promena sadržaja indeksnog registra, svi ostali slučajevi generišu više od jednog iskaza stabla među-koda. Jedan iskaz je potreban za konkretan pristup memorijskoj lokaciji, a ostali iskazi služe za izmenu adresnog registra na odgovarajući način.

Drugi primer ima značenje punjenja akumulatora A0 vrednošću memorijske lokacije čija je adresa zapisana u adresnom registru R0 i nalazi se u X memorijskoj zoni. Nakon pristupa memorijskoj lokaciji, vrednost adresnog registra se povećava za 1. Modeluje se sledećim iskazima jezika među-reprezentacije prikazanim na slici 4.4:



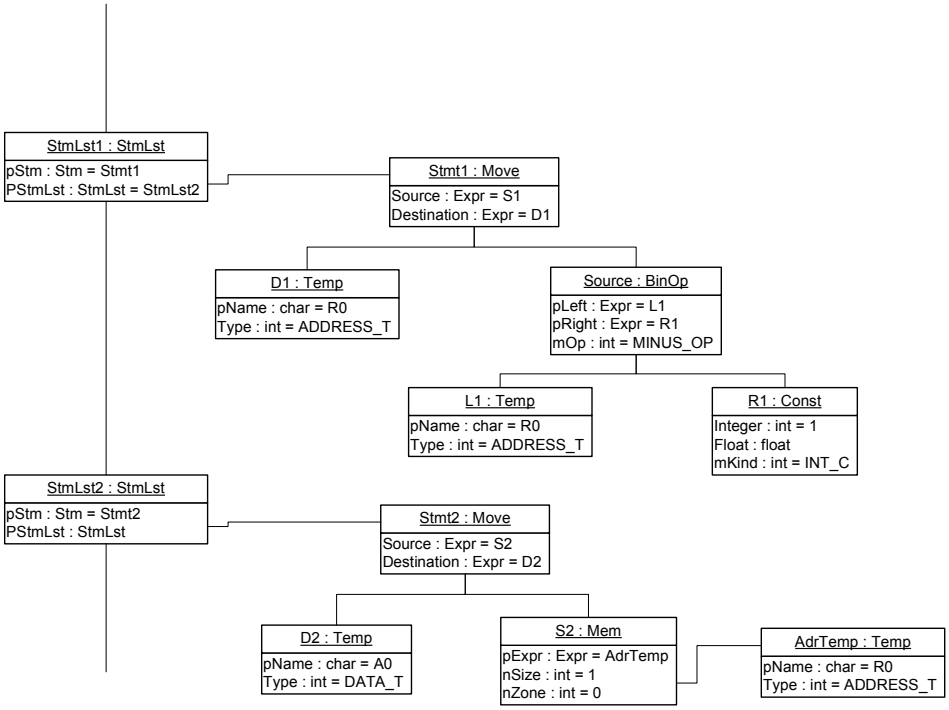
Slika 4.3: Razvijeno stablo među-reprezentacije

```
| ' ' zone ':' MINUS '(' reg_abxy ')' ',' reg_all
```

```
| ' ' reg_all ',' zone ':' MINUS '(' reg_abxy ')'

X:-(R0),A0
A0,X:-(R0)
```

Prvi primer ima značenje smanjivanja vrednosti adresnog registra za 1 i zatim punjenje akumulatora A0 vrednošću sadržaja memorijske lokacije iz X memorijske zone i izmenjene adrese R0. Ove operacije prikazane su na slici 4.4.

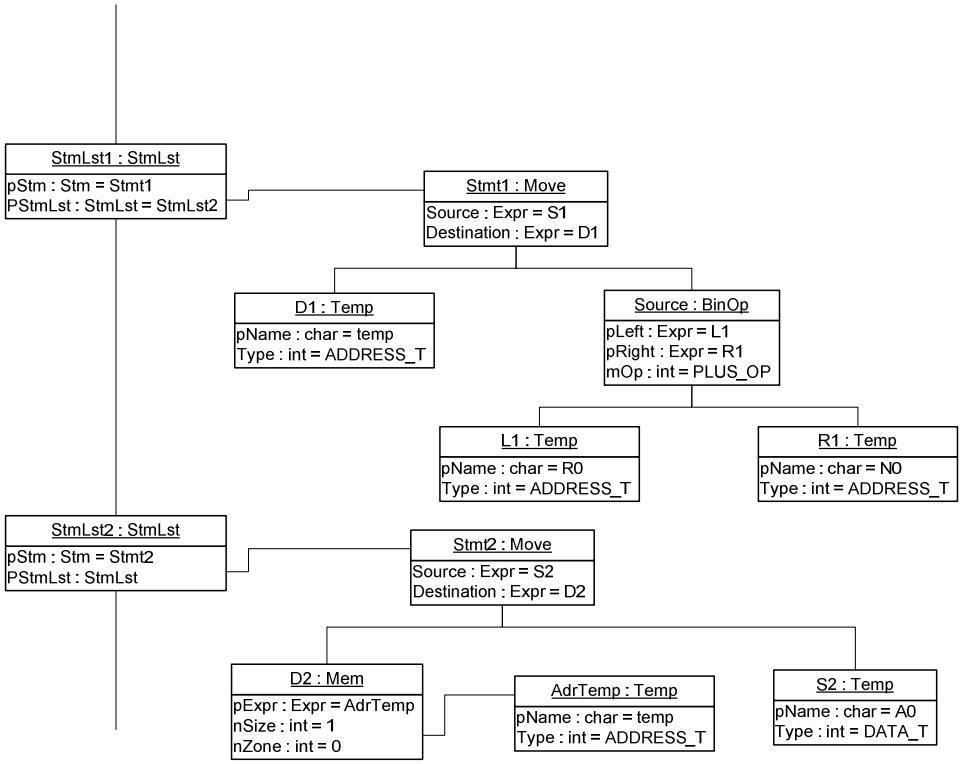


Slika 4.4: Razvijeno stablo među-reprezentacije

```
| ' ' zone ':' '(' reg_abxy PLUS reg_abxy ')' ',' reg_all
| ' ' reg_all ',' zone ':' '(' reg_abxy PLUS reg_abxy ')'

X:(R0+N0),A0
A0,X:(R0+N0)
```

Drugi primer opisuje iskaz punjenja memorijske lokacije iz X memorijske zone, čija je adresa zbir vrednosti adresnog registra R0 i indeksnog registra N0. Pri tom se vrednost adresnog registra ne menja. U jeziku među-reprezentacije ova operacija opisana je na slici 4.5.



Slika 4.5: Razvijeno stablo među-reprezentacije

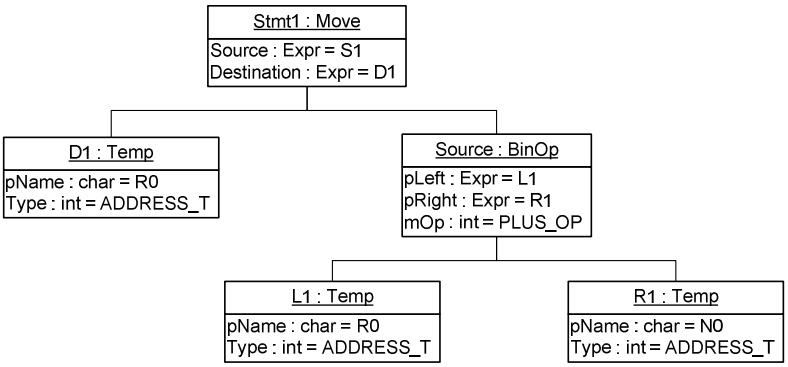
Ovim pravilima opisani su svi indirektni režimi adresiranja. Preostalim pravilom opisana je operacija izmene sadržaja adresnog registra bez pristupa memorijskoj lokaciji.

4.2.3 Izmena sadržaja adresnih registara bez pristupa memorijskoj lokaciji

```
| ' ' '(' reg_abxy ')' sign_eff  move_reg_abxy

(R0)+
(R0)-
(R0)+N0
(R0)-N0
```

Primerima su pokriveni slučajevi povećavanja, odnosno smanjivanja vrednosti adresnog registra za 1, odnosno za sadržaj indeksnog registra. U jeziku među-reprezentacije treći primer opisan je na slici 4.6.



Slika 4.6: Razvijeno stablo među-reprezentacije

Ovim pravilima pokriveni su svi slučajevi upotrebe jedinice za generisanje adresa u linearnom režimu. Ukoliko je sadržajem modifikator registra rad jedinice za generisanje adresa postavljen u moduo režim ili režim sa obratnim prenosom, javlja se niz novih problema koje je potrebno rešiti kako bi programi bili ispravno izvršavani na novoj platformi. Ovi problemi javljaju se iz nepotpune usaglašenosti jedinica za generisanje adresa polazne i ciljne platforme.

Sledećom tabelom dat je prikaz osnovnih razlika u arhitekturi jedinica za generisanje adresa:

Motorola 56000 Address Generation Unit	Coyote Address Generation Unit
Registri odstojanja 16 bita	Registri odstojanja 12 bita
Moduo modifikator registri 16 bita	Moduo modifikator registri 4 bits
Vrednost moduo registra u opsegu 0…32768	Vrednost moduo registra je n, gde je željeni moduo = 2^{n+1}
Aritmetika sa obratnim prenosom sa operatorom oduzimanja	Nije podržana

Tabela 4.1. Neusaglašenost jedinica za generisanje adresa

Rešenja neusaglašenosti polaznog i ciljnog procesora nazivaju se emulacijama. Neophodno je bilo emulirati ponašanje procesora Motorola 56000 na procesoru Coyote.

Prva dva problema proističu iz činjenice da se vrednosti odstojanja i moduo vrednosti na ciljnoj platformi zapisuju u jedan zajednički registar širine 16 bita, dok polazna platforma ima odvojene registre. Problem je rešen upotrebom akumulatora i aritmetičko-logičke jedinice za dobijanje ispravne adrese.

Činjenicom da se u moduo registar ciljne platforme ne mogu upisati sve vrednosti koje se mogu upisati u moduo registar polazne platforme, dobijamo moduo adresne režime koji nisu podržani. Ovaj problem rešen je ručnom proverom i eventualnom korekcijom vrednosti adresnog registra. S obzirom da u ovom slučaju vrednost moduo registra ima funkciju veličine struktura *buffer* i kružni *buffer*, praktično je bilo potrebno uvesti kontrolu položaja adresnog registra unutar ove strukture.

Problem ne postojanja aritmetike sa obratnim prenosom sa operatorom oduzimanja rešen je negacijom vrednosti registra odstojanja i primenom aritmetike sa operatorom sabiranja.

5 Testiranje i verifikacija

Problem testiranja i verifikacije programskog prevodioca veoma je složen. Mali testovi imaju za zadatak otkrivanja grešaka u radu programskog prevodioca, dok je verifikacija izvršena prevođenjem i izvršavanjem zadatog skupa aplikacija na ciljnoj platformi.

5.1 Testiranje kros asemblera DejaGNU testovima

DejaGNU je okruženje za testiranje programa. Svrha je da obezbedi jedinstven ulaz za sve testove. DejaGNU se sastoji iz skupa biblioteka napisanih u Tcl skript jeziku, napravljenih da podrže pisanje test alata. DejaGNU je napisan u Expect jeziku, koji opet koristi Tcl kao svoju osnovu.

DejaGNU nudi sledeće prednosti:

- Prilagodljivost i usaglašenost - omogućuju pisanje testova za bilo koji program,
- Obezbeđuje viši nivo abstrakcije - omogućuje pisanje testova koji su prenosivi na bilo koju platofrmu, na kojoj je potrebno izvršiti testiranje datog programa,
- Svi testovi imaju isti izlazni format - lakše integrisanje testova u dalji proces izrade programskih rešenja

Pošto je DejaGNU, kao što mu samo ime kaže, namenjen GNU/GPL programskim sistemima, da bi bilo moguće korišćenje DejaGNU testova na Windows platformama, neophodno je koristiti Cygwin radno okruženje. Cygwin radno okruženje je POSIX sistem za Windows platforme, koji obuhvata skup neophodnih GNU/GPL programa, i biblioteku koja omogućuje POSIX sistemske pozive na Windows platformama.

DejaGNU kao rezultat testiranja može da izda sledeće poruke:

- PASS – test je uspešno završen, tj. potvrđeno je da je tvrdnja istinita,
- FAIL – test je proizveo grešku koju je bilo potrebno proizvesti, tj. potvrđeno je da je tvrdnja neistinita,
- UNRESOLVED – test je proizveo neodređeni rezultat. Obično, ovo znači da se test izvršio, ali na neočekivan način. Ova situacija zahteva od programera da ručno pregleda kod i pronađe uzrok greške,
- XFAIL – DejaGNU ne podržava postojanje očekivanih neuspeha. Tako da u tom slučaju treba takođe da bude vraćena vrednost PASS,
- UNTESTED - ovo stanje označava da test nije izvršen
- UNSUPPORTED – ne postoji podrška za određeni test
- ERROR – označava da se desila značajna greška u izvršavanju testa, kao što je nepostojanje određene datoteke ili gubitak komunikacije
- WARNING – označava mogući problem prilikom testa. Obično se odnosi na greške koje su otklonjive

- NOTE – informacija o testu

Za potrebe testiranja kros asemblera, napisan je skup testova u asemblerskom jeziku procesora Motorola 56000. Za sve ove testove, očekivani izlaz je PASS. Izvršavanje testova u DejaGNU okruženju podrazumeva prevođenje kros asemblerom i izvršavanje na simulatoru ciljne platforme.

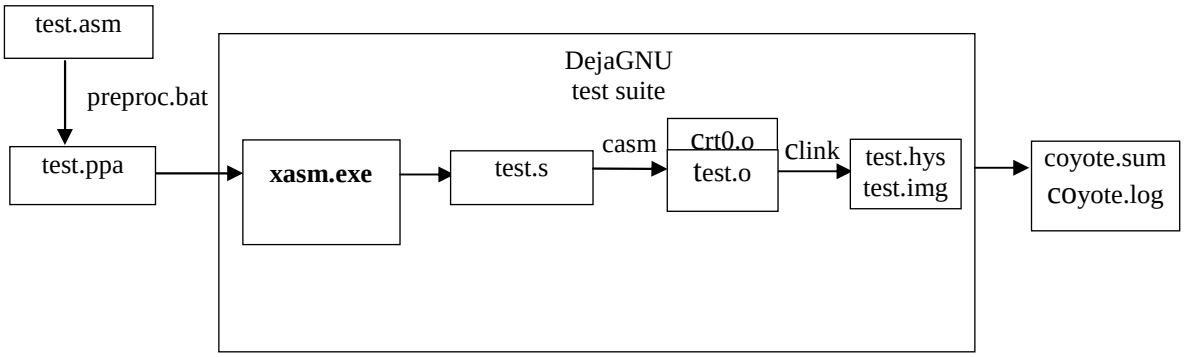
Svaki test ima za zadatak proveru prevođenja jedne instrukcije polazne platforme. Napisane su sledeće grupe testova:

- Aritmetičke instrukcije – 69 testova
- Logičke instrukcije – 12 testova
- Instrukcije za manipulaciju na nivou bita – 12 testova
- Petlje – 4 testa
- Instrukcije za pristup memorijskim lokacijama – 75 testova
- Instrukcije za kontrolu toka izvršenja programa – 70 testova

Ukupno je napisano 242 kratka testa. Raspodela testova po grupama zavisila je od prisutnosti određene instrukcije u verifikacionim aplikacijama.

Osim provere ispravnosti rada kros asemblera, testovi su imali i ulogu merenja performansi. Kao faktor određivanja performanse rada, uzet je odnos broja polazne i dobijene veličine mašinskog koda svakog testa.

Proces testiranja prikazan je na slici 5.1



Slika 5.1 DejaGNU testno okruženje

Tipičan test primer dat je u produžetku:

```
  include   'labels'
  move      #3,a
  move      #3,b
  abs a
  cmp a,b
  jeq _pass
  jsr       _abort
_pass
  jsr       _exit0
```

Navedeni primer testira instrukciju *abs*. Uključena datoteka *labels* sadrži deklaraciju podrutina *_abort* i *_exit0*. Testiranje se obavlja na sledeći način. Registar *a* sadrži izračunatu vrednost, dok se u registru *b* nalazi očekivana vrednost. Nakon poređenja ove dve vrednosti,

tok izvršenja testa ide ka *_abort*, odnosno *_exit0* grani u zavisnosti od rezultata poređenja. Ukoliko je rezultat očekivan, izvršava se grana *_exit0*.

Rezultat kros asembliranja ovog testa dat je u produžetku:

```
# Function : _main

       .code at (0x1000)

_main                          #cycle 0 | rules:       307
       fixed16(a1) = (768)     #cycle 1 | rules:       130
       a1 = |a1|               #cycle 2 | rules:       123
       fixed16(a0) = (768)     #cycle 3 | rules:       130
Label_1073741824               #cycle 0 | rules:       15
       a0 = a0 - a1            #cycle 1 | rules:       15
       if(a==0) jmp _pass      #cycle 2 | rules:       15
Label_1073741823               #cycle 0 | rules:       15
tmplabel_3                     #cycle 0 | rules:       307
Label_1073741825               #cycle 0 | rules:       6
       call _abort             #cycle 1 | rules:       6
tmplabel_4                     #cycle 0 | rules:       6
_pass                          #cycle 0 | rules:       307
Label_1073741826               #cycle 0 | rules:       6
       call _exit0             #cycle 1 | rules:       6
tmplabel_5                     #cycle 0 | rules:       6
Label_0                        #cycle 0 | rules:       307
__main_END                     #cycle 0 | rules:       307
                               #cycle 0 | rules:       34
       ret                     #cycle 1 | rules:       34
```

Podrutine *_exit0* i *_abort* definisane su na sledeći način:

```
# Function : _exit0
  .public _exit0
_exit0:
  halt 1
# Function : _abort

  .public _abort
_abort:
  halt -1
```

Ovaj test vratiće rezultat PASSED.

Rezultati testiranja prikazani su u tabeli 5.1. Značenje kolona je sledeće:

- Group instruction – grupa instrukcija
- Number of rules – broj implementiranih sintaksnih pravila
- Number of tests – broj predviđenih testova
- Implemented – broj implementiranih testova
- Tested – broj uspešnih testova
- 56k code size – veličina memorijske slike testa na platformi Motorola 56k
- Coyote code size – veličina memorijske slike prevedenog testa
- Coyote/56k ratio – odnos veličina memorijskih slika

- Number of appearance – broj pojavljivanja u aplikacijama
- App. Ratio – udeo u ukupnom broju pojavljivanja svih instrukcija

5.2 Verifikacija kros asemblera aplikacijama

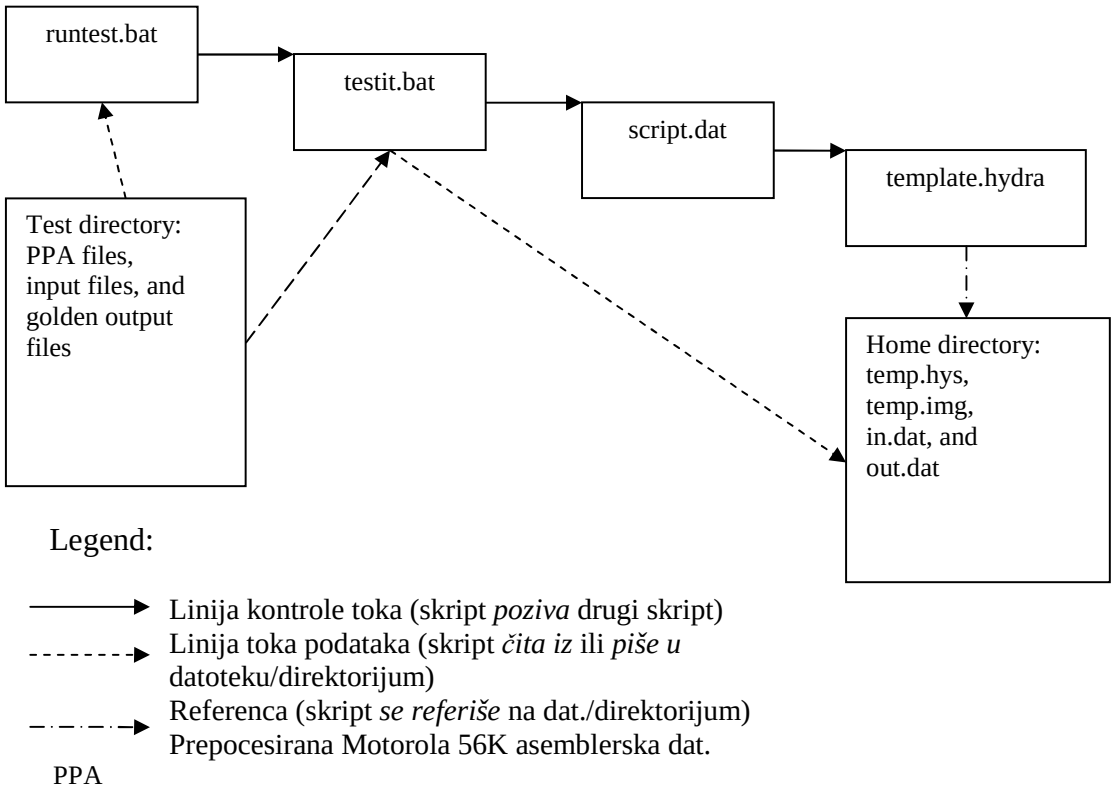
Za potrebe verifikacije kros asemblera, sastavljen je skup aplikacija pisanih za Motorola 56000 procesor. Svaka aplikacija pokrenuta je na simulatoru, a zatim i na razvojnoj ploči ove platforme. Na ovaj način dobijeni su izlazi iz aplikacija za unapred definisane ulaze, koji služe za verifikaciju aplikacija na ciljnoj platformi.

U pogledu načina testiranja razlikuju se dve grupe aplikacija:

- ulazni i izlazni podaci nalaze se u memorijskoj mapi
- ulazni podaci čitaju se iz datoteke, a izlazni se zapisuju u datoteku

Za prvu grupu aplikacija iskorišteno je DejaGNU testno okruženje na već opisan način. Svakoj prevedenoj aplikaciji pridružena je podrutina čiji je zadatak verifikacija dobijenih rezultata. Verifikacija se sastoji od poređenja svih vrednosti rezultata izvršenja na polaznoj i ciljnoj platformi. Rezultat testiranja aplikacije vraća rezultat PASSED ukoliko verifikaciona podrutina ne pronađe odstupanje rezultata sa ciljne platforme.

Usled specifičnosti aplikacija iz druge grupe, za njihovu verifikaciju nije bilo moguće iskoristiti DejaGNU testno okruženje. Način testiranja ove grupe aplikacija prikazan je na slici 5.2.



Slika 5.2 Testno okruženje

Automatizacija testiranja ostvarena je kroz 4 hijerarhijski povezane skript datoteke:

1. runtest.bat
2. testit.bat
3. script.dat
4. template.hydra

Prvi skript *runtest.bat* poziva sledeći *testit.bat* za svaku aplikaciju koja se nalazi u datom direktorijumu za testiranje. U ovom direktorijumu nalaze se aplikacije u asembleru procesora Motorola 56000, ulazne test datoteke za svaku aplikaciju, kao i izlazne test datoteke dobijene izvršenjem svake aplikacije na platformi procesora Motorola 56000.

Skript *testit.bat* poziva kros asembliranje pojedinačne aplikacije, kopira dobijene memorijske slike prevedene aplikacije, zajedno sa testnim datotekama u radni direktorijum. Zatim poziva simulator procesora Coyote nad prevedenom aplikacijom i ulaznom datotekom. Nakon izvršenja simulatora, dobijena izlazna datoteka poredi se sa datotekom dobijenom izvršenjem aplikacije na polaznoj platformi.

Skript *script.dat* sadrži niz komandi simulatora procesora Coyote. Rad simulatora započinje učitavanjem odgovarajućeg projekta, a zatim i njegovim pokretanjem. Definisan je i maksimalan broj ciklusa izvršenja, nakon čijeg isteka se prekida njegov rad. Na ovaj način eliminisana je mogućnost tzv. mrtve petlje, kada se tok izvršenja programa vrti kroz istu granu bez mogućnosti izlaska iz nje.

Template.hydra je tekstualna datoteka u kojoj se nalazi opis ciljne platforme, naziv datoteke sa memorijskom slikom i mapom prevedene aplikacije, kao i nazivi ulazne i izlazne datoteke.

Rezultat svake od ovih faza testiranja aplikacije zapisuje se u log datoteku radi kasnije analize.

Jedan primer verifikacije aplikacija je množenje dve matrice. Aplikacija *Matmul1.asm* napisana u Motorola 56000 asemblerskom jeziku data je u nastavku:

```
          org     x:0
mat_a
          dc      .4
          dc      .3
          dc      -.6
          org     y:0
mat_b
          dc      -.4
          dc      .5
          dc      .6
          dc      .35
          dc      -.3
          dc      -.35
          dc      .15
          dc      .4
          dc      .45
mat_x     ds      3
          org     p:$100
matmul    move              #mat_a,r0
          move              #mat_b,r4
          move              #2,m0
          move              #mat_x,r1
          move              x:(r0)+,x0  y:(r4)+,y0
          mpy     x0,y0,a   x:(r0)+,x0  y:(r4)+,y0
          mac     x0,y0,a   x:(r0)+,x0  y:(r4)+,y0
          macr    x0,y0,a   x:(r0)+,x0  y:(r4)+,y0
          move                          a,y:(r1)+
          mpy     x0,y0,a   x:(r0)+,x0  y:(r4)+,y0
          mac     x0,y0,a   x:(r0)+,x0  y:(r4)+,y0
          macr    x0,y0,a   x:(r0)+,x0  y:(r4)+,y0
          move                          a,y:(r1)+
          mpy     x0,y0,a   x:(r0)+,x0  y:(r4)+,y0
          mac     x0,y0,a   x:(r0)+,x0  y:(r4)+,y0
          macr    x0,y0,a   x:(r0)+,x0  y:(r4)+,y0
          move                          a,y:(r1)+
          end
```

Ulazni podaci, matrice a i b, zadati su strukturama *mat_a* i *mat_b*. Rezultat množenja je matrica x, *mat_x*. Ova aplikacija je ujedno i primer potrebe emulacije moduo adresnog režima, gde je vrednost modifikator registra različita od 2^n-1. Na primeru se vidi paralelizacija instrukcija množenja i memorijskog prenosa.

Prevedena aplikacija kros asemblerom bez emulacije nepodržanog moduo režima izgleda ovako:

```
# Function : _main
```

```
x_start:    .equ (0x0)
y_start:    .equ (0x0)
xy_start:   .equ (0x0)

   .xdata at (xy_start + 0)
mat_a:      .dw (0.400000)
   .dw (0.300000)
endbuff_5:  .dw (-0.600000)

   .ydata at (xy_start + 0)
mat_b:      .dw (-0.400000)
   .dw (0.500000)
   .dw (0.600000)
   .dw (0.350000)
   .dw (-0.300000)
   .dw (-0.350000)
   .dw (0.150000)
   .dw (0.400000)
endbuff_16: .dw (0.450000)
mat_x:      .bss (0x2)
endbuff_19: .bss (0x1)

   .code at (0x1000)

_main
   i0 = (0) + (mat_a)
   i5 = (0) + (mat_b)
   i4 = (0) + (mat_x)
   x0 = xmem[i0];  i0 += 1;    y0 = ymem[i5];    i5 += 1
   x0 = xmem[i0];  i0 += 1;    y0 = ymem[i5];    i5 += 1;    a0 =  x0 * y0
   x0 = xmem[i0];  i0 += 1;    y0 = ymem[i5];    i5 += 1;    a0 += x0 * y0
   x0 = xmem[i0];  i0 += 1;    y0 = ymem[i5];    i5 += 1;    a0 += x0 * y0
                               ymem[i4] = a0;    i4 += 1;    a0 =  x0 * y0
   x0 = xmem[i0];  i0 += 1;    y0 = ymem[i5];    i5 += 1
   x0 = xmem[i0];  i0 += 1;    y0 = ymem[i5];    i5 += 1;    a0 += x0 * y0
   x0 = xmem[i0];  i0 += 1;    y0 = ymem[i5];    i5 += 1;    a0 += x0 * y0
                               ymem[i4] = a0;    i4 += 1;    a0 =  x0 * y0
   x0 = xmem[i0];  i0 += 1;    y0 = ymem[i5];    i5 += 1
   x0 = xmem[i0];              y0 = ymem[i5];    i5 += 1;    a0 += x0 * y0
                               y0 = ymem[i5];                a0 += x0 * y0
                               ymem[i4] = a0
__main_END
   ret

   .public _main
```

Uočava se zadržani nivo paralelizacije instrukcija množenja i memorijskih prenosa. Razlika se ipak krije u potrebnom broju mašinskih ciklusa za izvršenje aplikacije. Na procesoru Motorola 56000 potrebne su dve memorijske reči za zapis jedne linije asemblerskog jezika sa paralelnom obradom, dok je na procesoru Cirrus Logic Coyote potrebna samo jedna memorijska reč. Međutim, navedeni primer bez emulacije ne odražava funkcionalnost polaznog programa. Funkcionalno ispravno prevedena aplikacija data je u produžetku:

```
# Function : _main
x_start:    .equ (0x0)
y_start:    .equ (0x0)
xy_start:   .equ (0x0)

   .xdata at (xy_start + 0)
mat_a:      .dw (0.400000)
   .dw (0.300000)
endbuff_5:  .dw (-0.600000)

   .ydata at (xy_start + 0)
mat_b:      .dw (-0.400000)
   .dw (0.500000)
   .dw (0.600000)
   .dw (0.350000)
   .dw (-0.300000)
   .dw (-0.350000)
   .dw (0.150000)
   .dw (0.400000)
endbuff_16: .dw (0.450000)
mat_x:      .bss (0x2)
endbuff_19: .bss (0x1)

   .code at (0x1000)

_main
   halfword(a0) = (endbuff_5)
   i0 = (0) + (mat_a)
   nop
   nm0 = (0xe000)
   i5 = (0) + (mat_b)
   b1 = i0
   i4 = (0) + (mat_x)
   uhalfword(b0) = (0)
   x0 = xmem[i0];              a0 = a0 - b1
Label_1073741824
                  i0 += 1;    a0 = a0 - b0
   if(a>0) jmp tmplabel_2
   i0 = (0) + (mat_a)
tmplabel_2
   halfword(b1) = (endbuff_5)
   b3 = i0
   uhalfword(b2) = (0)
   y0 = ymem[i5]; i5 += 1;    b1 = b1 - b3
   x0 = xmem[i0];             b0 =x0 * y0
Label_1073741826
                  i0 += 1;    b1 = b1 - b2
   if(b>0) jmp tmplabel_8
   i0 = (0) + (mat_a)
tmplabel_8
   halfword(a0) = (endbuff_5)
   b2 = i0
   uhalfword(b1) = (0)
   y0 = ymem[i5]; i5 += 1;    a0 = a0 - b2
   x0 = xmem[i0];             b0 += x0 * y0
Label_1073741828
                  i0 += 1;    a0 = a0 - b1
```

```
   if(a>0) jmp tmplabel_14
   i0 = (0) + (mat_a)
tmplabel_14
   halfword(a0) = (endbuff_5)
   b2 = i0
   uhalfword(b1) = (0)
   y0 = ymem[i5];  i5 += 1
                               b0 += x0 * y0
                               a0 = a0 - b2
   uhalfword(b0l) = (0)
   x0 = xmem[i0]
Label_1073741830
                   i0 += 1;    a0 = a0 - b1
   if(a>0) jmp tmplabel_20
   i0 = (0) + (mat_a)
tmplabel_20
   halfword(b1) = (endbuff_5)
   b3 = i0
   uhalfword(b2) = (0)
   y0 = ymem[i5];  i5 += 1
   ymem[i4] = b0;  i4 += 1;    b1 = b1 - b3
   x0 = xmem[i0];              b0 =x0 * y0
Label_1073741832
   i0 += 1;  b1 = b1 - b2
   if(b>0) jmp tmplabel_27
   i0 = (0) + (mat_a)
tmplabel_27
   halfword(b1) = (endbuff_5)
   b3 = i0
   uhalfword(b2) = (0)
   y0 = ymem[i5];  i5 += 1;    b1 = b1 - b3
   x0 = xmem[i0];              b0 += x0 * y0
Label_1073741834
                   i0 += 1;    b1 = b1 - b2
   if(b>0) jmp tmplabel_33
   i0 = (0) + (mat_a)
tmplabel_33
   halfword(b1) = (endbuff_5)
   b3 = i0
   uhalfword(b2) = (0)
   y0 = ymem[i5];  i5 += 1
                               b0 += x0 * y0
                               b1 = b1 - b3
   uhalfword(b0l) = (0)
   x0 = xmem[i0]
Label_1073741836
                   i0 += 1;    b1 = b1 - b2
   if(b>0) jmp tmplabel_39
   i0 = (0) + (mat_a)
tmplabel_39
   halfword(b1) = (endbuff_5)
   b3 = i0
   uhalfword(b2) = (0)
   y0 = ymem[i5];  i5 += 1
   ymem[i4] = b0;  i4 += 1;    b1 = b1 - b3
   x0 = xmem[i0];              b0 =x0 * y0
Label_1073741838
```

```
                  i0 += 1;    b1 = b1 - b2
  if(b>0) jmp tmplabel_46
  i0 = (0) + (mat_a)
tmplabel_46
  halfword(b1) = (endbuff_5)
  b3 = i0
  uhalfword(b2) = (0)
  y0 = ymem[i5]; i5 += 1;    b1 = b1 - b3
  x0 = xmem[i0];             b0 += x0 * y0
Label_1073741840
                  i0 += 1;    b1 = b1 - b2
  if(b>0) jmp tmplabel_52
  i0 = (0) + (mat_a)
tmplabel_52
  halfword(b1) = (endbuff_5)
  b3 = i0
  uhalfword(b2) = (0)
  y0 = ymem[i5]; i5 += 1;    b1 = b1 - b3
                             b1 = b1 - b2
  y0 = ymem[i5];             b0 += x0 * y0
  uhalfword(b0l) = (0)
  ymem[i4] = b0
__main_END

  ret

  .public _main
```

Može se uočiti da je postignut manji nivo paralelizacije, kao i znatno proširenje polazne aplikacije sa neophodnom emulacijom. Upotreba rutina emulacije u ovom primeru prvenstveno zavisi od vrednosti modifikator registra, a samim tim i veličina korišćenih struktura podataka. Iz tog razloga su sve aplikacije u kojima se javlja potreba za emulacijom prevedene i sa varijantom bez emulacija, kako bi se dobila prava slika o performansama samog prevodioca.

Tabela 5.2 sadrži rezultate verifikacije aplikacija. U tabeli se nalazi lista svih aplikacija sa sledećim kolonama:

- Application – naziv aplikacije ili pojedinačne datoteke
- Motorola 56k
  - Code size – veličina memorijske slike aplikacije
  - Code size ratio – udeo u ukupnoj veličini koda
  - Software simulation – uspešnost simulacije
  - Hardware execution – uspešnost izvršenja na ploči
- Cirrus Logic Coyote
  - Code size with emu – veličina memorijske slike prevedene aplikacije sa emulacijom
  - Code size – veličina memorijske slike prevedene aplikacije bez emulacije
  - Works with – lokacija ulaznih podataka (memorija ili datoteka)
  - Software simulation – uspešnost simulacije
- CDF with emu – odnos veličine polazne i prevedene aplikacije sa emulacijom

- CDF – odnos veličine polazne i prevedene aplikacije bez emulacije
- MIPS Motorola – broj ciklusa izvršenja na platformi Motorola
- MIPS Coyote with emulation – broj ciklusa izvršenja na platformi Coyote sa emulacijom
- MIPS ratio with emulation – odnos broja ciklusa izvršenja sa emulacijom
- MIPS ratio – odnos broja ciklusa izvršenja bez emulacije

6 Zaključak

U ovom radu realizovan je prednji deo krosasemblera sa procesora DSP Motorola 56000 na procesor Cirrus Logic Coyote, koji se odnosi na jedinicu za generisanje adresa. Krosasembler je realizovan razvojem prednjeg dela i ponovnim korišćenjem zadnjeg dela, koji je realizovan na katedri za računarsku tehniku i računarske komunikacije. Prednji deo je realizovan korišćenjem programskih alata FLEX i YACC i dopisivanjem rutina u programskom jeziku C. Za potrebe verifikacija dobijenih rešenja realizovano je okruženje za automatsko testiranje sa testnim slučajevima za pojedinačne instrukcije i aplikacijama za procenu performansi izlaznog koda.

Zadatak je uspešno realizovan o čemu govore rezultati testiranja i verifikacije ručno napisanih testova i gotovih aplikacija, koje predstavljaju rešenja realnih problema na DSP procesoru. Postignut je veliki uspeh u pogledu odnosa veličine koda polaznog i prevedenog asemblerskog koda.

Odnos od 1:1,75 postignut u prevođenju malih testova potiče iz slabe mogućnosti paralelizacije instrukcija kao i potrebe za emulacijom nekih nekompatibilnosti.

Odnos koda sa emulacijom postignut u prevođenju aplikacija iznosi 1:2,31, dok je odnos koda bez emulacije 1:0,91. Ove cifre govore koliko se polazna i ciljna platforma razlikuju. Poznavajući sve neusaglašenosti ovih platformi, malim izmenama na polaznom asemblerskom programu, mogu se postići još bolji rezultati u prevođenju. Odnos koda bez emulacije ispod 1 pokazuje tehnološku nadmoć nove arhitekture DSP procesora.

Postignuti rezultati ne predstavljaju apsolutni maksimum. Moderna modularna implementacija programskog prevodioca ostavlja prostora za primenu novih optimizacionih tehnika.

7 Literatura

[1] Prof. Dr Vladimir Kovačević, Prof. Dr Miroslav Popović: *Sistemska Programska Podrška u Realnom Vremenu*, Edicija tehničke nauke, Novi Sad, 2002.

[2] Prof. Dr Vladimir Kovačević: Logičko Projektovanje Računarskih Sistema i Projektovanje Digitalnih Sistema, Edicija tehničke nauke, Novi Sad, 2001.

[3] Andrew Appel, Jens Palsberg: *Modern Compiler Implementation in Java, Second Edition*, Cambridge Univeristy Press, Cambridge, 2002.

[4] Programming languages — C, INTERNATIONAL STANDARD ISO/IEC 9899:1999 (E)

[5] Cirrus Logic, Preliminary Product Information: *CS495XX Data Sheet*, USA, 2004.

[6] Cirrus Logic, Preliminary Product Information: *Coyote 32-bit DSP Instruction Set and Architecture Reference Manual*, USA, 2005.

[7] Motorola, DSP56000 24-BIT DIGITAL SIGNAL PROCESSOR FAMILY MANUAL, USA, 1994.