VEŽBA 6 – Tehnike optimizacije upotrebom asemblerskog jezika

6.1 Uvod

Tokom prethodne vežbe pokazano je na koji način tehnike optimizacije C koda omogućavaju kompajleru generisanje efikasnog asemblerskog koda i iskorišćenje hardverskih proširenja digitalnog signal procesora (kao što su: paralelni prenos podataka, hardverske petlje, namenski adresni generatori i sl..). Ipak, kod složenih aplikacije i strogih ograničenja po pitanju resursa, i pored primenjenih optimizacionih tehnika, kod koji generiše programski prevodilac može biti neefikasan.

U tom slučaju, sledeća faza razvoja aplikacija porazumeva modifikaciju koda primenom optimizacionih tehnika u asembleru. Nakon profilisanja, vremenski kritični zadaci, funkcije obrade i programske petlje se realizuju u asemblerskom jeziku, u skladu sa pozivnom konvencijom. Asemblerske rutine se realizuju na osnovu koda koji je prilagođen aritmetici ciljne platforme, odnosno ne dovode do smanjenja tačnosti u odnosu na Model 2.

Tokom izrade ove vežbe pokazaće se:

- kako se kritični delovi aplikacije, napisane u programskom jeziku C, realizuju u asembleru,
- kako se koristi postojanje različitih magistrala procesora i dvostrukih MAC i ALU jedinica za paralelno izvršavanje instrukcija,
- kako se pozivaju asemblerske funkcije iz C koda (konvencija poziva),
- kako se optimizuje kod napisan u asembleru.

Oslanjajući se na znanja stečena u okviru vežbi 2 i 4, realizovaće se deo aplikacije upotrebom asemblerkog jezika, upoznaće se sa metodama unapređenja efikasnosti asemblerskog koda, a procenom utrošenih resursa uvideće se i odnos efikasnosti koda pisanog u C-u i asemblerskom jeziku.

6.2 Optimizacija koda upotrebom asemblerskog jezika

U ovoj fazi razvoja se pretpostavlja da je realizovan programski kod koji se uspešno izvršava na simulatoru, ali da je broj procesorskih ciklusa koji je potreban za obradu podataka i nakon primenjih tehnika optimizacije prevelik. Isto tako, pretpostavlja se da je kod uspešno optimizovan u smislu memorije podataka.

Nakon profilisanja koda je potrebno uočiti najzahtevnije delove obrade, i implementirati ih upotrebom asemblerskog jezika za ciljnu platformu. Dva moguća pristupa su: implementacija pojedinih delova koda upotrebom ASM iskaza, i implementacija čitavih funkcija upotrebom asemblerskog jezika i njihovo pozivanje iz programskog jezika C.

6.2.1 ASM iskaz

ASM iskaz omogućava korišćenje asemblerskih instrukcija unutar C koda. Korišćenjem ASM iskaza omogućava se implementacija dela C koda upotrebom asemblerskih instrukcija, pri čemu se delu koda pisanom u asembleru prosleđuju C-ovski operandi. Ubacivanje asemblerskog koda vrši se na kritičnim delovima programa radi poboljšanja performansi prevedenog koda. Drugim rečima, ASM iskaz omogućava mešanje asemblerskog i C koda. Dat je primer asemblerskog iskaza i rezultat prevođenja priloženog C koda.

Primer:

primer.c:

```
#include <stdfix.h>
accum foo(fract a, fract b)
{
    accum tmp;

    asm("%0 = %1 * (unsigned)%2" :
        "=a"(tmp) : "x"(a), "y"(b));

    return tmp;
}
```

primer.s:

```
    .public _foo
    .code_ovly


_foo:
    x0 = a0h
    y0 = a1h
#asm begin
    a0 = x0 * (unsigned)y0
#asm end
    Ret
```

6.2.1.1 Format ASM iskaza

ASM predstavlja niz karaktera (*string*) koji se sastoji od asemblerskih instrukcija u kojima se umesto operanda nalazi *%broj*. Operandi su numerisani sa leva na desno počevši od nule:

```
      asm("AnyReg(%0,%4)" : "=U"(out1), "=U"(out2) : "U"(in1), "U"(in2),
"U"(in3));
```

Prvi izlazni operand (*out1*) i treći ulazni (*in3*) se koriste unutar asemblerske instrukcije. CCC2 prosto zamenjuje ime registra umesto *%broj* unutar ASM stringa i takvog ga stavlja u prevedenu asemblersku datoteku. Višestruke instrukcije unutar jednog ASM stringa treba razdvajati koristeći „\n\t“ – nova linija pa novi tabulator:

```
      asm("AnyReg(%0,%2)\n\tAnyReg(%1,%3)" : "=U"(out1), "=U"(out2) :
"U"(in1),"U"(in2));
```

Ili

```
      asm("AnyReg(%0,%2)

      AnyReg(%1,%3)" : "=U"(out1), "=U"(out2) : "U"(in1), "U"(in2));
```

6.2.1.2 Ulazni i izlazni operandi ASM iskaza

Svaki operand je opisan stringom iza kojeg sledi C izraz. Broj operanada ASM iskaza je ograničen na 10. C izraz predstavlja spregu između asemblerskih operanada (registri) i C operanada. C izraz može da predstavlja odredište vrednosti koja je dobijena izvršavanjem asemblerske instrukcije (u slučaju izlaznih operanada), izraz koji predstavlja vrednost za inicijalizaciju registra u asembleru (u slučaju ulaznog operanda) ili oba.

6.2.1.3 Određivanje tipa operanda

Tip operanda koji se prosleđuje ASM iskazu određuje se korišćenjem sledećih ograničavača:

- “a“ – akumulator (a0, a1, a2, a3)
- „b“ – akumulator (b0, b1, b2, b3)
- „x“ – registri za podatke (x0, x1, x2, x3)
- „y“ – registri za podatke (y0, y1, y2, y3)
- „i“ – indeks registri (i0, i1, i2, i3, i4, i5, i6)
- „U“ – bilo koji registar – kompajler će uzeti prvi slobodan

Svi ulazni i izlazni operandi su dodeljeni različitim registrima. Izuzetak je slučaj ulazno-izlaznog operanda kojem će biti dodeljen jedan registar. Pored ograničavača operandima ASM iskaza se dodaju i modifikatori:

- “” – bez modifikatora – operand će biti ulazni.
- „=“ – operand je deklarisan kao izlazni
- “+” – operand će biti deklarisan kao ulazno-izlazni
- „&“ – operand je deklarisan kao *earlyclobber*. Pretpostavka da se ulazni operandi referenciraju pre izlaznih može biti pogrešna ako se u ASM stringu nalazi više od jedne asemblerske instrukcije. U ovom slučaju „&“ modifikator se može koristiti za označavanje izlaznih operanada, koji ne bi trebalo da koriste isti registar kao ulazni operand, ili da specifikuje ulazni operand koji ne bi trebalo da koristi isti registar kao izlazni operand.

Primer:

```
      asm("AnyReg(%0,%2)/n/tAnyReg(%1,%3)" : "=U"(out1), "=U"(out2) :
"U"(in1),"U"(in2));
```

Kod dobijen prevođenjem:

```
      AnyReg(a0,a1)
      AnyReg(a1,a0)
```

Out1 će da preuzme vrednost *in1* i *out2* će da preuzme vrednost *in2*. Operandima 0 i 3 mogu biti dodeljeni isti registri zato što je jedan ulazni a drugi izlazni. U tom slučaju će generisani asemblerski kod izgledati pogrešno. Problem se rešava korišćenjem & modifikatora:

Primer:

```
      asm("AnyReg(%0,%2)/n/tAnyReg(%1,%3)" : "=&U"(out1), "=U"(out2) :
"U"(in1),"U"(in2));
```

ili

```
      asm("AnyReg(%0,%2)/n/tAnyReg(%1,%3)" : "=U"(out1), "=U"(out2) :
"U"(in1), "&U"(in2));
```

Kod dobijen prevođenjem:

```
      AnyReg(a0,a1)
      AnyReg(a1,a2)
```

6.2.1.4 Clobber lista

Predstavlja listu registara koji se implicitno koriste i čiji se sadržaj menja u okviru jednog ASM iskaza. Sledeći primer generiše kod koji nije validan iz razloga što

kompajler ne zna da se a0 i a1 koriste i da se njihov sadržaj menja, tako da će vrednost 1 završiti u oba registra umesto da a0 dobije vrednost 1, a a1 vrednost 2.

Primer:

```
asm("AnyReg(a0,%0)\n\tAnyReg(a1,%1)" : : "a"(1), "a"(2));
```

Kod dobijen prevođenjem:

```
AnyReg(a0,a1)
AnyReg(a1,a0)
```

Korišćenjem *clobber* liste kompajler zna da se a0 i a1 implicitno koriste i sačuvaće njihovu konsistenciju. Dodatno, kompajler nikada neće dodeliti registre a0 i a1 nekom drugom operandu:

Primer:

```
asm("AnyReg(a0,%0)\n\tAnyReg(a1,%1)" : : "a"(1), "a"(2) : "a0", "a1");
```

Registri koji se mogu naći u clobber listi ASM iskaza su:

- "a0", "a1", "a2", "a3", "b0", "b1", "b2", "b3"
- "x0", "x1", "x2", "x3", "y0", "y1", "y2", "y3"
- "i0", "i1", "i2", "i3", "i4", "i5", "i6"

Primer:

```
int func (int* src, int* dest, int count)
{
      int result;
      asm("
            \tx0=%1; a0=+%3
            \ti0=%2
            \tcall _asmfunc
            \t%0=+a0"
            : "=a"(result)
            : "y"(src), "y"(dest), "a"(count)
            : "a0", "i0", "i3", "x0", "y2");
            return result;
}
```

6.2.2 Poziv asemblerskih funkcija iz programskog jezika C

Ukoliko određene funkcije nije moguće optimizovati u okviru programskog jezika C, tako da zadovoljavaju uslove koji se tiču potrošnje resursa, moguće je implementirati ih u potpunosti upotrebom asemblerskog jezika.

Tok prevođenja koda prikazan u poglavlju 4 pokazuje da se C kod prvo prevodi u asemblerski kod, na osnovu koga se dalje generišu objektne datoteke sa mašinskim kodom a nakon čega sledi povezivanje objektnih datoteka. Tok prevođenja za aplikacije pisane u asemblerskom kodu jednak je ovom toku bez prvog koraka (kompajlera). Iz prikazanog se može zaključiti da se na jednostavan način može pisati deo aplikacije koristeći asemblerski jezik, a deo koristeći C jezik, i u fazi povezivanja sastaviti te delove u jednu aplikaciju. Razvojno okruženje CLIDE pravi razliku između ručno pisanih asemblerskih datoteka i asemblerskih datoteka generisanih od strane kompajlera. Pravilo je da ručno pisane datoteke imaju nastavak „.a“, dok generisane asemblerske datoteke imaju nastavak „.s“. Potrebno je voditi računa o ovim nastavcima kako bi se izbegli problemi u toku prevođenja.

Kako bi se omogućio poziv asemblerskih funkcija iz dela programa pisanog u programskom jeziku C, i obrnuto, potrebno je voditi računa o nekoliko stvari.

6.2.2.1 Simboli generisani od strane kompajlera

Svi simboli generisani od strane kompajlera sadrže prefiks „_“. Svim simbolima definisanim u C domenu se pristupa iz asemblerskog jezika tako što se na naziv simbola doda pomenuti prefiks. Ovo pravilo važi i za promenljive i za funkcije.

Simboli definisani u programskom jeziku C:

```
__memX int x = 2;

void processingC()
{
    x++;
}
```

Pristup istim simbolima iz ASM jezika:

```
    .extern _processingC
    .extern _x

_mainASM:
    a0=(0)
    xmem[_x]=a0;
    call _processingC
    ret
```

Pored dodavanja prefiksa potrebno je voditi računa i o području vidljivosti simbola. Da bi neki simbol definisan u C domenu bio vidljiv unutar asemblerskog koda, neophodno je da taj simbol bude globalan. U asemblerskom kodu neophodno je deklarisati isti simbol kao eksterni koristeći ključnu reč *.extern.*

Da bi neki simbol bio vidljiv iz programskog jezika C, potrebno je u okviru datoteke gde je simbol definisan proglasit isti za globalan koristeći ključnu reč *.public*. Neophodno je, pored toga, napisati i deklaraciju tog simbola u okviru datoteke sa C kodom (ili zaglavlju uključenom od strane C datoteke).

Prilikom pristupa promenljivama potrebno je voditi računa o memorijskom prostoru u kome se ta promenljiva nalazi.

Simboli definisani u asemblerskom jeziku:

```
    .public _processingASM
    .public _y

    .xdata_ovly

_y    .dw (0)

    .code_ovly
_processingASM:
    a0=xmem[_y]
    a0 = a0 >> 1
    xmem[_y]=a0
    ret
```

Pristup simbolima iz programskog jezika C:

```
extern void processingASM();
extern __memX int y;
void main()
{
    y++;
    processingASM();
}
```

6.2.2.2 Pozivna konvencija kod CCC2 programskog prevodioca

Do sada je prikazano na koji način je moguće definisati funkciju upotrebom asemblerskog jezika i potom je pozvati u okviru funkcije pisane upotrebom programskog jezika C. Prikazane funkcije su bile bez parametara i povratne vrednosti. U slučaju funkcija koje sadrže parametre i/ili povratnu vrednost, način na koji se prosleđivanje parametara vrši propisan je skupom pravila koji se naziva pozivna konvencija.

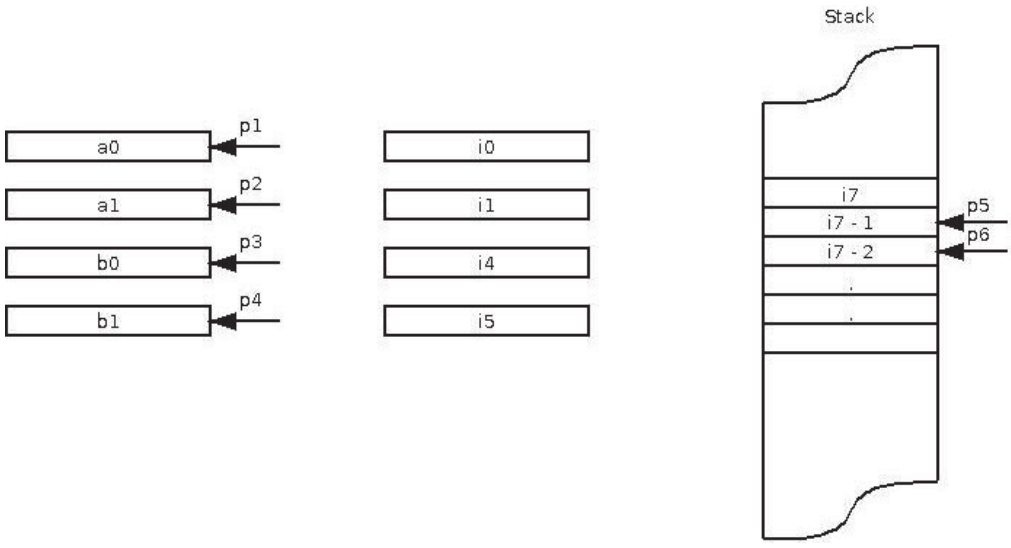
Prilikom prevođenja izvornog koda upotrebom CCC2 pozivna konvencija opisana je sledećim pravilima:

1. Argumenti se broje sa leva na desno.
2. Ako je parametar pokazivač, koristi se indeksni registar (i0, i1, i4, i5 tim redosledom).
3. Ako je parametar bilo kog drugog osnovnog tipa osim pokazivača, koriste se akumulatori (a0, a1, b0, b1 tim redosledom).
4. Registri za podatke se nikada ne koriste za prosleđivanje parametara.
5. Strukture se prosleđuju preko steka.
6. Indeksni registar i7 je pokazivač na stek, za koji se podrazumeva da se nalazi u memorijskoj zoni X. Stek raste naviše (nakon dodavanja elementa na stek, pokazivač se uvećava).
7. Ako funkcija vraća vrednost ona će se nalaziti u akumulatoru a0.
8. Ukoliko se prosleđuje više od 4 pokazivača u funkciju, prva četiri se prosleđuju korišćenjem indeksnih registara, dok se ostali prosleđuju preko steka.

9. Ukoliko se prosleđuje više od 4 parametra koja nisu pokazivači u funkciju, prva četiri se prosleđuju korišćenjem akumulatora, dok se ostali prosleđuju preko steka.
10. Prarametri se preko steka prosleđuju u obrnutom redosledu (tako da se prvi parametar nalazi na vrhu steka).
11. Registri a2, a3, b2, b3, x2, x3, y2, y3, i2, i3, i6 su rezervisani za korišćenje unutar funkcije koja vrši poziv. Ako se bilo koji od njih koristi unutar funkcije koja je pozvana neophodno ih je na početku funkcije sačuvati na steku i na kraju vratiti nazad njihove vrednosti. U slučaju da je pozivana funkcija pisana upotrebom programskog jezika C, CCC2 će generisati kod koji to obavlja, dok u slučaju ASM funkcija neophodno je to uraditi ručno.

Skup parametara funkcije definisane u asemblerskom jeziku određen je njenom deklaracijom u programskom jeziku C. Sledi primer deklaracije funkcije i ilustracija načina prosleđivanja parametara na osnovu pozivne konvencije.

```
void foo(int p1, int p2, int p3, int p4, int p5, int p6)
```



Slika 6.1 - Ilustracija prosleđivanja parametara kod funkcije foo

6.2.2.3 Prljanje registara – clobber lista kod asemblerskih funkcija

U poslednjem pravilu pozivne konvencije spomenuti su registri čiju je vrednost potrebno sačuvati na početku funkcije ukoliko se koriste. Preostali registri su a0, a1, b0, b1, i0, i1, i4, i5, koji se koriste za prosleđivanje parametara, i registri za podatake x0, x1, y0 i y1. Njihovo korišćenje je slobodno unutar pozivane funkcije.

Kako bi to bilo moguće, neophodno je da funkcija koja vrši poziv sačuva vrednost tih registara na steku kako bi nakon završetka pozivane funkcije te vrednosti mogle da se vrate.

U slučaju da je pozivana funkcija pisana upotrebom asemblerskog jezika, i da je u trenutku pisanja koda poznato da neće koristiti neke od ovih registara, moguće je specifikovati kompajleru koji je to skup registara koji je neophodno čuvati pre poziva. Ova operacija se vrši koristeći pragma iskaz:

- #pragma ASM_FUNC_CLOBBERED_REGS(<string of register names>)

Pragma iskaz se piše iznad deklaracije funkcije u programskom jeziku C, važi samo za funkcije čije je telo implementirano upotrebom asemblerskog jezika. Lista registara može da sadrži samo navedene registre koje kompajler podrazumevano čuva. Ukoliko je pragma iskaz izostavljen, podrazumevana je lista svih pomenutih registara. Dat je primer asemblerske funkcije i njoj odgovarajućeg pragma iskaza.

```
_boo:
    x0 = xmem[i0];
i0 += 1
    y0 = xmem[i0]
    a0 = x0 * y0
    xmem[i1] = a0
    ret
```

```
#pragma ASM_FUNC_CLOBBERED_REGS("a0,x0,y0,i0")
void boo(__Xmem _Fract* in, __Xmem _Fract*
out);
```

Parametri funkcije nalaze se u registrima i0 i i1 po konvenciji. Registri koji se koriste unutar funkcije su a0, x0, y0 i i0. Obratiti pažnju da se registar i1 ne nalazi u *clobber* listi. Razlog je to što se vrednost registra i1 ne menja u toku izvršenja funkcije.

6.3 Tehnike optimizacije asemblerskog koda za procesore CS48x

Efikasnost rutina napisanih u asembleru, u odnosu na kod generisan kompajlerom, se primarno zasniva na boljoj upotrebi paralelizma koje procesor omogućava, odnosno na paralelnom izvršavanju više operacija procesora unutar jedne instrukcione reči.

6.3.1 Paralelno izvršavanje više operacija unutar jedne instrukcione reči

Dužina programske reči procesora CS48x je 32 bita, a asembler dozvoljava jednu 32-bitnu programsku reč u jednoj liniji. Međutim, dok pojedine operacije koriste svih 32 bita programske reči (*full word instructions*), neke koriste svega 16 pa i 8 bita.

Ukoliko postoje dva paralelna prenosa podataka u instrukciji, svaki pojedinačni prenos koristi po 8 bita instrukcione reči (*short word instructions*). Svaki paralelni prenos (sa zauzećem gornjih 16 bita instrukcione reči) može biti iskombinovan sa bilo kojom aritmetičko-logičkom operacijom (koja zauzima donjih 16 bita), čime se formira kompletna instrukciona reč.

Prema zauzeću instrukcione reči, operacije se mogu razvrstati u sledeće kategorije:

1. Operacije koje zauzimaju celu instrukcionu reč
    a. Operacije kontrole toka izvršenja programa
    b. 64-bitni periferalni prenos (u ovu grupu spadaju i primitivne instrukcije koje mogu poslužiti za aproksimativno deljenje, kvadriranje i korenovanje)
    c. Memorijski prenos
    d. Upis konstante
    e. Operacije za manipulacije nad bitima
2. Operacije koje zauzimaju 16 gornjih bita instrukcione reči
    a. Samostalni višenamenski prenosi (prenosi između registara podataka i memorije X i Y). Podržavaju indeksno i neposredno (6-bitno) adresiranje. Postoje restrikcije ali svaki od njih se može naći u paraleli sa aritmetičkom operacijom
    b. 64-bitni višenamenski prenosi
    c. Ažuriranje indeksnih registara
1. Operacije koje zauzimaju po 8 gornjih bita instrukcione reči
    a. Paralelni višenamenski prenosi. Dozvoljavaju po jedan prenos sa X i jedan prenos sa memorijskom zonom Y u jednoj instrukciji. U istoj instrukciji može biti izvršeno i ažuriranje odgovarajućih indeksnih registara, a važi i to da mogu biti smeštene u paraleli sa MAC/ALU instrukcijom. Restrikcije vezane za protok podataka iz/u memorijsku zonu X ili Y su sledeće:
    b. Memorijska zona X može biti adresirana samo korišćenjem indeksnih registara i0 i i1
    c. Memorijska zona Y može biti adresirana samo korišćenjem indeksnih registara i4 i i5
    d. Memorijski prenos ka memorijskoj zoni X može biti obavljen jedino korišćenjem A akumulatora
    e. Memorijski prenos ka memorijskoj zoni Y može biti obavljen jedino korišćenjem B akumulatora
    f. Akumulatori (a0-a3, b0-b3) mogu učestvovati jedino kao izvorišni resursi.
    g. Registri za podatke (x0-x3, y0-y3) mogu učestvovati

Sa stanovišta formata asemblerskog programa, jedna linija asemblerskog koda predstavlja jednu instrukcionu reč. Pisanje instrukcione reči koja se sastoji iz više instrukcija vrši se u istoj liniji, pri čemu se instrukcije odvajaju znakom „;“. Labele, iako se nalaze u istoj liniji nisu uključene u instrukcionu reč, već samo simbolom predstavljaju adresu te reči. Pojedine aritmetičke operacije dozvoljavaju smeštanje rezultata u dva akumulatorska registra.

Primeri paralelizama unutar jedne instrukcione reči su dati u tabeli 6.1.

Tabela 6.1 – Paralelizmi

Instrukcija	Sintaksa	Polja CCR promenjena instrukcijom	Restrikcije
Paralelni prenos podataka između memorije i registara koji se može kombinovati sa aritmetičkim operacijama.	x0 = xmem[i0]; i0+=n; y0 = ymem[i4]; i4+=n x0 = xmem[i0]; i0+=n; ymem[i4]=b0 a0=x2*y0; x0=xmem[i0]; i0+=n; ymem[i4]=b0 xmem[i1]=a0; ymem[i4]=b3; i4+=1 x0=xmem[i0]; i0+=1; y0=ymem[i5]; i5+=n x3=xmem[i0]; i0+=1; ymem[i5]=b2; i5-=1	Nijedno	memorijska zona X mora biti adresirana korišćenjem registara i0 i i1 dok se za adresiranje Y memorije koriste i4 i i5. Podaci iz memorije X mogu se upisati samo u grupu X, dok se podaci iz Y memorije mogu upisati u Y grupu registara. U memoriju se podaci mogu upisivati samo poštujući sledeća pravila: u memoriju X iz A grupe akumulatora, u memoriju Y iz B grupe akumulatora.
Paralelni prenos podataka između registara i memorije i registara međusobno	ymem[i4]=b2; b2=a2	Nijedan	Ako je akumulator izvorni operand i obe instrukcije ne smeju da pripadaju istoj grupi. Ako je registar izvorni operand ograničenja nema.
Paralelni MAC	Aq=Ar ±= Xn*Xm;Bq=Br ±= Yn*Xm Aq=Ar ±= Xn*Xm;Bq=Br ±= -Yn*Xm Aq=Ar ±= Xn*Ym;Bq=Br ±= Yn*Ym Aq=Ar ±= Xn*Ym;Bq=Br ±= -	Nijedan	Date u opisu sintakse

	Yn*Ym Aq=Ar ±= Yn*Xm;Bq=Br ±= Xn*Xm Aq=Ar ±= Yn*Xm;Bq=Br ±= -Xn*Xm Aq=Ar ±= Yn*Ym;Bq=Br ±= Xn*Ym Aq=Ar ±= Yn*Ym;Bq=Br ±= -Xn*Ym Aq=Ar ±= -Xn*Xm;Bq=Br ±= Yn*Xm Aq=Ar ±= -Xn*Xm;Bq=Br ±= -Yn*Xm Aq=Ar ±= -Xn*Ym;Bq=Br ±= Yn*Ym Aq=Ar ±= -Xn*Ym;Bq=Br ±= -Yn*Ym Aq=Ar ±= -Yn*Xm;Bq=Br ±= Xn*Xm Aq=Ar ±= -Yn*Xm;Bq=Br ±= -Xn*Xm Aq=Ar ±= -Yn*Ym;Bq=Br ±= Xn*Ym Aq=Ar ±= -Yn*Ym;Bq=Br ±= -Xn*Ym		
Paralelni MAC II	Aq=Ap ±= Xn*Yn;Bq=Bp ±= Xn*Xm Aq=Ap ±= Xn*Yn;Bq=Bp ±= Xn*Ym Aq=Ap ±= Xn*Xm;Bq=Bp ±= Xn*Yn Aq=Ap ±= Xn*Ym;Bq=Bp ±= Xn*Yn Aq=Ap ±= Yn*Xn;Bq=Bp ±= Yn*Xm Aq=Ap ±= Yn*Xn;Bq=Bp ±= Yn*Ym Aq=Ap ±= Yn*Xm;Bq=Bp ±= Yn*Xn Aq=Ap ±= Yn*Ym;Bq=Bp ±= Yn*Xn Aq=Ap = -XY;Bq=Bp = -XX Aq=Ap = -XY;Bq=Bp = -XY Aq=Ap = -XX;Bq=Bp = -XY Aq=Ap = -YX;Bq=Bp = -YX Aq=Ap = -YX;Bq=Bp = -YY Aq=Ap = -YY;Bq=Bp = -YX	Nijedan	Date u opisu sintakse
Paralelni kvadrat broja	Aq = Ar ±= -Xn*Xn;Bq = Br ±= -Yn*Ym	Nijedan	Odredište: a0,b0 a1,b1 a2,b2

			a3,b3 a1a0,b1b0 a3a1,b3b1 a0a2,b0b2 a2a3,b2b3
Paralelni MAC sa dodavanjem	Aq = Ar = A0±Xn*Xm;Bq = Br = B0±Yn*Xm Aq = Ar = A0±Xn*Ym;Bq = Br = B0±Yn*Ym	Nijedan	Odredište: a0,b0 a1,b1 a2,b2 a3,b3 a1a0,b1b0 a3a1,b3b1 a0a2,b0b2 a2a3,b2b3
Paralelno množenje sa jedinicom uz opciono akumuliranje	Aq = Ap ±= ±Xn;Bq = Bp ±= ±Yn Aq = Ap ±= ±Yn;Bq = Bp ±= ±Xn	Nijedan	Odredište: a0,b0 a1,b1 a2,b2 a3,b3 a1a0,b1b0 a3a1,b3b1 a0a2,b0b2 a2a3,b2b3
Paralelno dodavanje sa opcionim pomeranjem	Ap=An ± Am;Bp=Bn ± Bm Ap=An ± Bm;Bp=Bn ± Am Ap=(An * 2) ± Am;Bp=(Bn * 2) ± Bm Ap=(An * 2) ± Bm;Bp=(Bn * 2) ± Am	A0, AS, B0, BS	Odredišni operand može da bude samo jedan od osam akumulatora.
Paralelni prenos podataka iz akumulatora u akumulator koristeći ALU[*1]	An =+ Am; Bn =+ Bm An =+ Bm; Bn =+ Am	A0, AS, B0, BS	Odredišni operand može da bude samo jedan od osam akumulatora.
Paralelni komplement jedinice	An =~ Am; Bn =~ Bm An =~ Bm; Bn =~ Am	A0, AS, B0, BS	Odredišni operand može da bude samo jedan od osam akumulatora.
Paralelno negiranje sadržaja akumulatora (komplement dvojke)	An =- Am; Bn =- Bm An =- Bm; Bn =- Am	A0, AS, B0, BS	Odredišni operand može da bude samo jedan od osam akumulatora.
Paralelna apsolutna vrednost akumulatora	An = |Am|; Bn = |Bm| An = |Bm|; Bn = |Am|	A0, AS, B0, BS	Odredišni operand može da bude samo jedan od osam akumulatora.
Paralelno OR, XOR i AND	An = An | Am;Bn = Bn | Bm An = An | Bm;Bn = Bn | Am	A0, AS, B0, BS	Odredišni operand može da bude samo

	An = An ^ Am;Bn = Bn ^ Bm An = An ^ Bm;Bn = Bn ^ Am An = An & Am;Bn = Bn & Bm An = An & Bm;Bn = Bn & Am		jedan od osam akumulatora.
Paralelno anuliranje sadržaja registra	An = 0; Bn = 0	A0, AS, B0, BS	Odredišni operand može da bude samo jedan od osam akumulatora.
Paralelno pomeranje	An = An << 1; Bn = Bn << 1 An = An << 4;Bn = Bn << 4 An = An << 8;Bn = Bn << 8 An = An >> 1;Bn = Bn >> 1	A0, AS, B0, BS	Odredišni operand može da bude samo jedan od osam akumulatora.
Paralelno ispitivanje vrednosti	An & Am;Bn & Bm An & Bm;Bn & Am	A0, AS, B0, BS	Odredišni operand može da bude samo jedan od osam akumulatora.
Paralelno upoređivanje akumulatora	An - Am;Bn - Bm An - Bm;Bn – Am |An| - |Am|;|Bn| - |Bm| |An| - |Bm|;|Bn| - |Am|	A0, AS, B0, BS	Odredišni operand može da bude samo jedan od osam akumulatora.

6.3.2 Primeri optimizacije koda

U ovom odeljku dati su primeri optimizacije asemblerskog koda. Prikazano je na koji način je moguće promenom redosleda instrukcija ili dodavanjem pojedinih instrukcija omogućiti bolje iskorišćenje paralelizma unutar instrukcione reči. Uz svaki primer dati su i rezultati u vidu broja potrebnih ciklusa za izvršenje optimizovane i neoptimizovane verzije koda.

6.3.2.1 Primer 1 – Kopiranje nizova

Prvi primer predstavlja kopiranje sadržaja jednog memorijskog niza u drugi kada se ne nalaze u istoj memorijskoj zoni. U prvom slučaju broj instrukcijskih ciklusa za izvršenje kopiranja jeste 2+128*2 = 258.

U drugom slučaju je prikazano na koji način se mogu iskoristiti paralelno čitanje i pisanje podataka u memoriju. Unutar petlje u jednom ciklusu se vrši: smeštanje vrednosti registra *x3* u akumulator kroz MAC jedinicu, čitanje nove vrednosti ulaznog niza u *x3*, uvećanje pokazivača, upis prethodne vrednosti *b3* u odredišni niz i uvećanje odredišnog pokazivača. Za ovakav pristup potrebno je 6+126*1 = 132 ciklusa. Može se zaključiti da se sa povećanjem veličine niza povećava i faktor optimizacije.

Neoptimizovan kod:

```
      i0 = (X_BX_Buffer1)
      i4 = (X_BY_Buffer2)
      do (128), >
            b3 = xmem[i0]; i0+=1
%:          ymem[i4] = b3; i4+=1
```

Kod nakon optimizacije:

```
      i0 = (X_BX_Buffer1)
      i4 = (X_BY_Buffer2)
      b3 = xmem[i0]; i0+=1
      x3 = xmem[i0]; i0+=1
      do (128 - 2), >
%:          b3 =+ x3; x3 = xmem[i0]; i0+=1; ymem[i4] = b3; i4+=1
      ymem[i4] = b3; i4+=1
      ymem[i4] = x3; i4+=1
```

6.3.2.2 Primer 2 – Aritmetičke operacije nad elementima dva niza

Na datom primeru prikazano je skaliranje dva odbiraka dva niza (kada se jedan nalazi u memorijskoj zoni X, a drugi u Y) množenjem sa jednom skalarnom konstantom. U prvom slučaju vrši se smeštanje vrednosti konstante u registar *y0* i postavljanje indeksnih registara *i0* i *i4* na početak nizova koje je potrebno skalirati. Nakon toga, u petlji se redom učitavaju odbirci jednog i drugog niza, množe sa konstantom i rezultat množenja se smešta u memorijske nizove. Broj potrebnih ciklusa za izvršenje ovog koda iznosi 3+128*6 = 771. Rešenje nakon optimizacije koristi po dva indeksna registra za svaki memorijski niz, jedan za čitanje i jedan za pisanje. Ovakav pristup omogućava da se u jednom ciklusu vrši množenje i učitavanje odbiraka za sledeću iteraciju, a u narednom smeštanje izračunatih odbiraka u memoriju. Optimizovan kod troši 7+128*2 = 263 ciklusa, međutim neophodna su mu dva indeksna registra više.

Neoptimizovan kod:

```
      ufixed16(y0) = (0x4000) #<--| y0 = 0x40000000 = 0.5
      i0 = (X_BX_Buffer1)
      i4 = (X_BY_Buffer2)
      do(128),>Loop
            x1 = xmem[i0]
            y1 = ymem[i4]
            a0 = x1 * y0
            b0 = y1 * y0
            xmem[i0] = a0; i0+=1
%Loop:      ymem[i4] = b0; i4+=1
```

Kod nakon optimizacije

```
      ufixed16(y0) = (0x4000) #<--| y0 = 0x40000000 = 0.5
      i0 = (X_BX_Buffer1) #<--| Pokazivač pisanja u X_BX_Buffer1
      i4 = (X_BY_Buffer2) #<--| Pokazivač pisanja u X_BY_Buffer2
      i1 = i0 #<--| Pokazivač čitanja iz X_BX_Buffer1
      i5 = i4 #<--| Pokazivač čitanja iz X_BY_Buffer2
      x1 = xmem[i1]; i1+=1 #<--| učitati ulazne odbirke za prvu
iteraciju
      y1 = ymem[i5]; i5+=1 #<--|
      do(128),>Loop
            a0 = x1 * y0; b0 = y1 * y0; x1 = xmem[i1]; i1+=1; y1 =
ymem[i5]; i5+=1
%Loop:      xmem[i0] = a0; i0+=1; ymem[i4] = b0; i4+=1
```

6.3.2.3 Primer 3 – Omogućavanje paralelizma smanjenjem broja iteracija petlje (1)

U ovom primeru prikazano je na koji način je moguće povećati broj kandidata za paralelizaciju instrukcija smanjenjem broja iteracija petlje. U kodu nakon optimizacije u jednoj iteraciji petlje obrađuju se po dva elementa niza. Samim tim ima dvostruko više instrukcija koje je moguće staviti u paralelu. U prvom slučaju broj potrebnih ciklusa iznosi 3+128*3 = 387. Optimizovani kod troši 6+64*5 = 326 ciklusa.

Neoptimizovan kod:

```
      i0 = (X_BX_Buffer1)
      i4 = (X_BY_Buffer2)
      uhalfword(a1) = (0x1)
      do(128), >Loop
            a0 = x0 * y0; x0 = xmem[i0]
            a0 = a0 + a1; y0 = ymem[i4]; i4+=1
%Loop:      xmem[i0] = a0; i0+=1
```

Kod nakon optimizacije

```
      i0 = (X_BX_Buffer1)
      i4 = (X_BY_Buffer2)
      i1 = i0
      uhalfword(a1) = (0x1)
      x0 = xmem[i0]; i0+=1
      y0 = ymem[i4]; i4+=1
      do(64), >Loop
            a0 = x0 * y0; x0 = xmem[i0]; i0+=1
            a0 = a0 + a1; y0 = ymem[i4]; i4+=1
            xmem[i1] = a0; i1+=1; a0 = x0 * y0
            a0 = a0 + a1; x0 = xmem[i0]; i0+=1
%Loop:      y0 = ymem[i4]; i4+=1; xmem[i1] = a0; i1+=1
```

6.3.2.4 Primer 4 – Omogućavanje paralelizma smanjenjem broja iteracija petlje (2)

Još jedan način optimizacije koda datog u prethodnom primeru prikazan je na primeru u nastavku. Indeks pisanja u *X_BX_Buffer1* postavlja se na jednu lokaciju pre početka niza. Zatim se izvrši čitanje prvog odbirka niza *X_BY_Buffer2* unapred. U registar u kome će se čuvati rezultat za upis postavlja se vrednost koja se trenutno nalazi na lokaciji pre početka niza *X_BX_Buffer1*, kako prilikom prvog upisa ne bi došlo do promene ove vrednosti. U prvoj iteraciji se samo izračunava vrednost koju je potrebno upisati u memoriju, a upisuje se vrednost unapred smeštena u *a2* na lokaciju na koju pokazuje indeksni registar pisanja. U svakoj narednoj iteraciji računa se trenutna vrednost za upis, a upisuje u memoriju prethodna. Na kraju, nakon izvršene petlje potrebno je upisati i poslednju izračunatu vrednost u memoriju. Ovakvo rešenje zahteva 9+128*2=265 ciklusa.

```
      i0 = (X_BX_Buffer1)
      i4 = (X_BY_Buffer2)
      i1 = (X_BX_Buffer1)
      x0 = xmem[i0]; i0+=1
      i1-=1
      y0 = ymem[i4]; i4+=1
      a2 = xmem[i1]
      uhalfword(a1) = (0x1)
      do(128),>Loop
            a0 = x0 * y0; x0 = xmem[i0]; i0+=1; y0 = ymem[i4]; i4+=1
%Loop:      a2 = a0 + a1; xmem[i1] = a2; i1+=1
      xmem[i1] = a2; i1+=1
```

6.4 Zadaci za samostalnu izradu

6.4.1 Zadatak 1: Dodavanje ASM iskaza u C kod

U ovom zadatku potrebno je izmeniti kod aplikacije realizovane u prethodnoj vežbi (*multitapEcho_model3*) dodavanjem ASM iskaza. Primeri korišćenja asemblerskog iskaza dati su u okviru projekta *ccc2Lessons,* datoteka *lesson06ASMStatement*.

6.4.1.1 Postavka zadatka:

1. Otvoriti projekat *multitapEcho_model3* u okviru CLIDE razvojnog okruženja.
2. Deo koda koji se nalazi unutar ugnježdene petlje unutar *multitap_echo* funkcije zameniti ugrađenim asemblerskim iskazom (*inline ASM*) koji obavlja istu funkcionalnost.
3. Prevesti i izvršiti aplikaciju. Ispitati ispravnost poređenjem izlaza sa izlazima iz aplikacije realizovane u prethodnoj vežbi.

6.4.2 Zadatak 2: Izrada sistema za dodavanje višestrukog eho efekta upotrebom asemblerskog jezika

U ovom zadatku potrebno je proširiti aplikaciju realizovanu u prethodnoj vežbi (*multitapEcho_model3*), tako što će se funkcija obrade (*multitap_echo*) realizovati upotrebom asemblerskog jezika. Potom je potrebno optimizovati asemblersku funkciju korišćenjem mogućnosti paralelnog izvršenja instrukcija.

6.4.2.1 Postavka zadatka:

1. Otvoriti projekat *multitapEcho_model3* u okviru CLIDE razvojnog okruženja.
2. U *inc* direktorijum dodati datoteku *multitapEcho.h.*
3. U okviru *multitapEcho.h* napisati deklaraciju funkcije *multitap_echo.*
4. U *src* direktorijum dodati datoteku *multitapEcho.a.*
5. U okviru *multitapEcho.a* definisati funkciju _multitap_echo. Voditi računa da ispred naziva labele stoji simbol „**_**“. *CCC2* automatski dodaje ovaj simbol ispred naziva generisanih labela.
6. Definisani simbol proglasiti za globalni (.*public _multitap_echo*) kako bi bio vidljiv iz ostalih datoteka.
7. Sve globalne simbole definisane u *main.c* a korišćene u funkciji obrade deklarisati kao eksterne unutar *multitapEcho.a.* Svim simbolima definisanim u programskom jeziku C pridružiti simbol „_“ kada se koriste u asemblerskom jeziku.
8. Implementirati funkciju *_multitap_echo* na osnovu C koda. Prosleđeni argumenti funkciji prilikom poziva nalaziće se unutar odgovarajućih registara definisanih pozivnom konvencijom.
9. Unutar *main.c* obrisati funkciju *multitap_echo* i uključiti zaglavlje *multitapEcho.h*

10. Prevesti i izvršiti aplikaciju. Ispitati ispravnost poređenjem izlaza sa izlazima iz aplikacije realizovane u prethodnoj vežbi.
11. Ponoviti ceo postupak za funkciju *multitap_echo_init.*