VEŽBA 5 – Profilisanje C koda i tehnike optimizacije

5.1 Uvod

DSP arhitekture spadaju u procesorske arhitekture sa ograničenim resursima. Savremeni sistemi za digitalnu obradu signala podrazumevaju obradu velikog broja podataka u ograničenim vremenskim intervalima, odnosno u realnom vremenu. Iz tog razloga je prilikom razvoja DSP aplikacija neophodno voditi računa o utrošenim resursima. Najbitnije informacije su potrošnja procesorskih ciklusa, odnosno da li je procesor u stanju da obradi podatke u zadatom vremenu, i zauzeće memorije, to jest da li procesor ima dovoljno memorije da se smesti programski kod i podatke.

Simulatori fizičke platforme obično omogućavaju izvršavanje i ispitivanje koda koji zahteva više resursa nego što fizička platforma omogućava. Kod koji se ispravno izvršava na simulatoru je važna tačka u razvoju DSP aplikacija. Kada ispitivanje rezultata nad skupom ispitnih vektora pokaže da modifikacije koda nisu unele grešku u izvršavanju algoritma, pristupa se analizi korišćenja resursa prilikom izvršavanja koda. Ova analiza se zove profilisanje koda i u zavisnosti od rezultata profilisanja, ciljne platforme i zahteva aplikacije, primenjuju se odgovarajuće optimizacione tehnike.

U okviru ove vežbe, pokazaće se na koji način se vrši profilisanje DSP aplikacije i koje mogućnosti za procenu efikasnosti koda nudi razvojno okruženje CLIDE. Obradiće se tehnike optimizacije procesorskih ciklusa i memorije, i biće prikazano kako se:

- pronalazi deo koda koji predstavlja zahtevnu obradu,
- primenjuju optimizacije specifične za arhitekturu CS48x,
- proverava da li je izmena doprinela efikasnijem izvršenju koda i u kojoj meri.

5.2 Profilisanje DSP aplikacija

Profilisanje koda daje informacije o utrošenim resursima, odnosno, govori da li se program može izvršavati na ciljnoj platformi, kao i da li brzina izvršavanja ispunjava zadata vremenska ograničenja. Pored potrošnje procesorskih ciklusa (eng. *Million Instructions Per Second* - MIPS) i zauzeća memorije (programske i memorije za podatke), postoje i dodatne informacije koje omogućavaju pisanje efikasnijeg koda, kao što je broj pristupa memoriji.

Profilisanje aplikacije predstavlja ispitivanje ponašanja programa korišćenjem informacija sakupljenih u toku samog izvršavanja programa. Analizom performansi utvrđuju se delovi programa koji su pogodni za optimizaciju brzine izvršavanja ili korišćenja memorije.

5.2.1 Određivanje broja utrošenih instrukcijskih ciklusa

Optimizacija brzine izvršavanja se vrši nad delovima programa koji se često izvršavaju (najčešće su to tela programskih petlji). Integrisano razvojno okruženje CLIDE omogućava profilisanje izvršavanja koda, a informacije o profilisanju su dostupne za aplikacije koje se sastoje od datoteka pisanih u C-u, asemblerskom jeziku ili u aplikacijama koje predstavljaju kombinaciju C i asemblerskih datoteka. Profilisanje koda se vrši u okviru simulatorskog okruženja. Funkcionalnost koja se odnosi na brojanje ciklusa, vremena izvršavanja i ostale neophodne informacije su integrisane u sam simulator.

Prilikom pokretanja profilisanja nije neophodno specijalno/ponovno prevođenje programa. Da bi profilisanje koda bilo omogućeno, neophodno je podesiti odgovarajuću *.groovy* datoteku koja se nalazi u simulatorskom projektu. Unutar pomenute datoteke potrebno je podesiti promenljivu *profiling_support* na vrednost *on*, kao što je dato u sledećem kodu:

```
//
// Profiling support
//
  'profiling_support':          "on",
```

Informacije o profilisanom kodu se mogu dobiti prilikom samog izvršavanja programa koristeći prikaz koji se zove *Profiling*. Svako zaustavljanje DSP procesa prouzrokovaće osvežavanje *Profiling* prikaza. Ukoliko *Profiling* prikaz nije otvoren, otvaranje se može uraditi koristeći *Windows* meni i *Show View* opciju. Podaci dobijeni profilisanjem koda mogu biti prikazani u formi tabele ili u formi stabla kod koga je prikazana struktura poziva funkcija. Izgled *Profiling* prikaza je dat na slici 5.1. Tabela sadrži redom naziv funkcije, datoteku sa izvornim kodom, liniju na kojoj

se funkcija nalazi, adresu funkcije u programskoj memoriji, broj ciklusa potrošen unutar tela funkcije (ne računajući pozive podrutina), prosečan broj instrukcije potrošen unutar tela funkcije (odgovara vrednosti iz prethodne kolone podeljenoj sa brojem pristupa funkciji), ukupan broj ciklusa potrošen unutar tela funkcije računajući i cikluse potrošene na izvršenje podrutina i u poslednjoj koloni broj poziva funkcije.

Profiling

Symbols	Source File	Line Num...	Address	Base Time	Average Base Time	Cumulative Time	Calls
root			0x0000	35	35.00	700	1
X_S_SRSUnit	..\..\lesson02SRSUnit.a	49	0x0001	57	57.00	57	1
X_S_ProgramFlow	..\..\lesson04ProgramFlo...	49	0x0039	137	137.00	145	1
X_S_SampleFunctionInterruptControll	..\..\lesson04ProgramFlo...	189	0x0064	4	4.00	4	1
X_S_SampleFunction	..\..\lesson04ProgramFlo...	156	0x0061	4	4.00	4	1
X_S_MemoryMoves	..\..\lesson05MemoryMo...	66	0x0067	48	48.00	48	1
X_S_MACInstructions	..\..\lesson06MACUnit.a	47	0x00de	15	15.00	15	1
X_S_Introduction	..\..\lesson01Introduction.a	95	0x00ec	2	2.00	2	1
X_S_ALUUnitInstructions	..\..\lesson07ALUUnit.a	47	0x00bc	35	35.00	35	1
X_S_AddressGenerationUnit	..\..\lesson03AGU.a	56	0x0095	231	231.00	231	1
__SegStart_CODE_PROM_LIB_BIQUAD	..\..\quad.a	32	0x1200	132	66.00	132	2

Slika 5.1– Profiling prikaz

Broj utrošenih instrukcijskih ciklusa može se dobaviti i unutar samog koda koji se izvršava na simulatoru. Dobavljanje se vrši pozivom funkcije:

- **`unsigned long long cl_get_cycle_count ()`**

Ova funkcija nalazi se unutar zaglavlja <*dsplib\timers.h*>. Određivanje broja ciklusa utrošenog na izvršavanje dela koda se vrši tako što se dobavi trenutna vrednost utrošenih ciklusa pre bloka koda od interesa pozivom pomenute funkcije, dobavi se vrednost trenutne vrednosti utrošenih ciklusa nakon bloka koda od interesa i te dve vrednosti se oduzmu.

```
      unsigned long long count1, count2, spent_cycles;
      count1 = cl_get_cycle_count() ; // Get current cycle count

      // Execute some processing
      for(i = 0; i<BRICK_SIZE; i++)
      {
            *ptr3 =  ((*ptr2)>>1) + ((*ptr1)>>1);
            ptr1++;ptr2++;ptr3++;
      }

      count2 = cl_get_cycle_count(); // Get current cycle count
      spent_cycles = count2 - count1; // Calculate cycle count spent
on processing
```

5.2.2 Analiza utrošene memorije

Analiza potrebne memorije predstavlja bitnu stavku prilikom izrade DSP aplikacije. Na osnovu analize utrošene memorije proverava se da li napisana aplikacije može da se uklopi u memorijska ograničenja fizičke arhitekture. Procena se vrši tako što se izmeri ukupna količina zauzete memorije za sve tri memorijske zone (X, Y i programska memorija), i uporedi se sa veličinom odabrane memorijske mape za ciljnu arhitekturu.

Merenje zauzeća memorije vrši se analizom generisane .MAP ili .lst datoteke nakon prevođenja. Datoteka .MAP se nakon prevođenja projekta nalazi unutar izlazne datoteke. Ova datoteka sadrži informacije o zauzetoj memoriji od strane cele aplikacije, kao i nazive, adrese i inicijalne vrednosti svih simbola.

Na početku datoteke data je lista zauzete memorije po segmentima:

```
Address class CODE        Address:[0000...FFFF], length 10000
        Address:[0000...0000], length 0001, <RESERVED for crt0_GEN_0002, class
CODE>
        Address:[0001...0228], length 0228, <RESERVED for main_GEN_0008, class
CODE_OVLY>
         ...
        Address:[023D...023F], length 0003, <RESERVED for fwrite_x_GEN_0000, class
CODE_OVLY>
        Address:[0240...FFFF], length FDC0, <FREE>

Address class X  Address:[0000...FFFF], length 10000
        Address:[0000...0000], length 0001, <RESERVED for crt0_GEN_0000, class X>
          ...
        Address:[01BF...09BE], length 0800, <RESERVED for simulator_stack_GEN_0000,
class X>
        Address:[09BF...FFFF], length F641, <FREE>

Address class X_AVAIL    Address:[0000...FBFF], length FC00, subclass of X
        Address:[0001...0100], length 0100, <RESERVED for main_GEN_0003, class
X_OVLY>
        ...
        Address:[01BE...01BE], length 0001, <RESERVED for main_GEN_0000, class
X_OVLY>
        Address:[09BF...FBFF], length F241, <FREE>
```

Ukoliko je neki segment memorije podskup drugog, u njegovom prikazu stoji ključna reč *subclass* (npr. *Address class X_AVAIL ..., subclass of X*). Prilikom računanja ukupne potrošnje memorije ovi segmenti se ne računaju (već su uračunati u nadskupove). Ispod ovog odeljka dat je pregled memorije po objektnim datotekama,

pa se može odrediti tačna potrošnja memorije jednog od strane jedne objektne datoteke.

```
----------Module:  main(Segment name)
          main_GEN_0000, address 01BE ,length 0001, class    X_OVLY
          main_GEN_0001, address 0141 ,length 003F, class    X_OVLY
          main_GEN_0002, address 0101 ,length 0040, class    X_OVLY
          main_GEN_0003, address 0001 ,length 0100, class    X_OVLY
          main_GEN_0004, address 019E ,length 0018, class    X_OVLY
          main_GEN_0005, address 01B6 ,length 0004, class    X_OVLY
          main_GEN_0006, address 0180 ,length 001E, class    X_OVLY
          main_GEN_0007, address 01BA ,length 0004, class    X_OVLY
          main_GEN_0008, address 0001 ,length 0228, class  CODE_OVLY
```

Iz datog primera se vidi da je ukupna potrošnja main modula:

Memorijska zona X	Memorijska zona Y	Programska memorija
446 reči	0 reči	552 reči

Pored utrošene memorije potrebno je voditi računa i o najvećoj zauzetoj adresi (adresa poslednjeg segmenta u memorijskoj zoni + njegova veličina). Ukoliko se koriste kvalifikatori za poravnanje ili eksplicitno zauzimanje memorije na određenoj adresi, može se desiti da između segmenata postoji nezauzeta memorija.

Datoteka .lst sadrži samo analizu utroška memorije po objektnim datotekama, u malo drugačijem formatu. Ova datoteka sadrži i prikaz tačnog rasporeda vrednosti po adresama u memoriji, kao prikaz rasporeda asemblerskih instrukcija u programskoj memoriji. Ove informacije se mogu iskoristiti za procenu utroška memorije od strane zasebne funkcije unutar objektne datoteke.

5.3 Tehnike optimizacije C koda za prevođenje CCC2 prevodiocem

U ovom odeljku dat je pregled tehnika optimizacije specifičnih za arhitekture kojima je namenjen CCC2 kompajler. Ove optimizacije su zasnovane na mogućnostima, proširenjima i ograničenjima pomenutih (i sličnih) DSP arhitektura.

5.3.1 Optimizacija programskih petlji

Podrška hardverske petlje uz indeksne generatore je jedna od najvažnijih osobina DSP procesora. Prednost hardverskih petlji jeste ušteda procesorskih ciklusa koji se troše na instrukcije za uvećanje indeksa, proveru uslova i instrukcije skoka. Opis hardverskih petlji dat je u poglavlju 2. Da bi CCC2 kompajler generisao kod koji se oslanja na hardversku petlju, potrebno je da petlja napisana u okviru C koda bude u formi *for* petlje. Pored toga potrebno je da petlja ispunjava sledeće uslove:

- Uslov kraja petlje mora biti prisutan.
- Sve vrednosti u okviru uslova, izuzev indeksa, moraju biti konstantne.
- Mora biti prisutna modifikacija indeksa petlje.
- Modifikacija indeksa mora biti konstanta.

Drugim rečima, CCC2 će uspešno iskoristiti hardversku petlju samo u slučaju kada može unapred da izračuna broj iteracija. Ta informacija mora biti sadržana unutar izraza *for* petlje. Ovo je praktično sadržano u prva tri uslova hardverske petlje. Takođe, računanje broja iteracija je jednoznačno određeno ukoliko se indeks ne modifikuje unutar tela petlje. Ova provera postoji unutar CCC2 kompajlera, što objašnjava četvrti zahtev.

Na sledećim primerima prikazane su petlje koje ispunjavaju navedene uslove. Prevođenje svakog od navedenih primera upotrebom CCC2, rezultovaće asemblerskim kodom koji koristi hardversku petlju.

Primer 1:

```
int foo ()
{
      int i, sum = 0;
      for (i = 0 ; i < 10 ; i++)
            sum += array1[i] + array2[i];
      return sum;
}
```

Primer 2:

```
int foo ()
{
      int i, sum = 0;
      for (i = -5 ; i < 5 ; i+=2)
            sum += array1[i] + array2[i];
      return sum;
}
```

Primer 3:

```
#define win_size 512
int foo ()
{
      int i, sum = 0;
      for (i = 0 ; i < win_size ; i++)
            sum += array1[i] + array2[i];
      return sum;
}
```

Primer 4:

```
int foo ()
{
      int i, sum = 0;
      for (i = 1 ; i < 1024 ; i *= 2)
            sum += array1[i] + array2[i];
      return sum;
}
```

Naredni primeri ne ispunjavaju uslove za generisanje hardverskih petlji:

Primer 5 (zahtev 1):

```
int foo ()
{
      int i, sum = 0;
      for (i = 0 ; ; i++)
      {
            if (i < 10)
                  break;
            sum += array1[i] + array2[i];
      }
      return sum;
}
```

Primer 6 (zahtev 2):

```
int x = 1;
int foo ()
{
      int i, sum = 0;
      for (i = 0 ; i < x ; i++)
      {
            sum += array1[i] + array2[i];
      }
      return sum;
}
```

Primer 7 (zahtev 4):

```
int x = 1;
int foo ()
{
      int i, sum = 0;
      for (i = 0 ; i < 10 ; i+=x)
      {
            sum += array1[i] + array2[i];
      }
      return sum;
}
```

Kako bi se ilustrovala prednost korišćenja hardverske petlje, prikazani su rezultati prevođenja primera koji ispunjava, i primera koji ne ispunjava date uslove. Rezultat

prevođenja dat je u formi asemblerskog koda. Sa leve strane je prikazan rezultat prevođenja koda koji ispunjava uslove (Primer 1), a sa desne strane rezultat prevođenja koda koji ne ispunjava uslove hardverske petlje (Primer 7).

Primer1.s:

```
_foo:
    a0 = 0
    a1 = 0
    do (0xa), label_end_92
label_begin_92:
    i0 = a1
    i1 = a1
    i0 = i0 + (_array1 + 0)
    b0 = xmem[i0]
    i0 = i1 + (_array2 + 0)
    b1 = xmem[i0]
    b0 = b0 + b1
    a0 = a0 + b0
    uhalfword(b0) = (0x1)
label_end_92:
    a1 = a1 + b0
for_end_0:
    ret
```

Primer7.s:

```
_foo:
    a0 = 0
    a1 = 0
for_1:
    uhalfword(b0) = (0xa)
    a1 - b0
    if (a >= 0) jmp (for_end_1)
    i0 = a1
    i1 = a1
    i0 = i0 + (_array1 + 0)
    b0 = xmem[i0]
    i0 = i1 + (_array2 + 0)
    b1 = xmem[i0]
    b0 = b0 + b1
    a0 = a0 + b0
    b0 = xmem[_x + 0]
    a1 = a1 + b0
    jmp (for_1)
for_end_1:
    ret
```

5.3.2 Optimizacija uslovnih iskaza

Uslovni iskaz koji sadrži poređenje dve promenljive, kompajler gotovo uvek prevodi u instrukciju oduzimanja, koristeći aritmetičko logičku jedinicu i instrukciju grananja na osnovu registra stanja. Neefikasnost prilikom prevođenja potiče od toga što prilikom izvršenja operacije oduzimanja može doći do prekoračenja opsega tipa promenljivih koje se porede. Prekoračenje opsega se razrešava na različite načine, u zavisnosti od imlementacije samog programskog prevodioca i tipa podataka. Standard C jezika ne propisuje koja vrednost se nalazi unutar promenljive nakon što se desi prekoračenje. Kako bi se omogućilo da poređenje dva broja daje ispravan rezultat, neophodno je sprečiti prekoračenje usled oduzimanja. Sprečavanje prekoračenja opsega vrši se tako što CCC2 prevodilac generiše dodatne instrukcije koje služe za pripremu vrednosti koje se nalaze u registrima, a odnose se na proširenje znaka i/ili deljenje vrednosti sa 2.

Efikasnost poređenja se može poboljšati u slučajevima kada je tokom pisanja koda unapred poznato da do prekoračenja ne može doći. Poboljšanje se postiže tako što se umesto neposrednog poređenja između dve promenljive, koristi računanje razlike dve promenljive i poređenje razlike sa nulom.

Pisanje koda na ovakav način štedi dva ciklusa prilikom poređenja vrednosti u nepokretnom zarezu, odnosno, 4 ciklusa prilikom poređenja celobrojne vrednosti

Primer:

```
if (a < b) ...
```

Rezultat prevođenja:

```
a0 = a0 >> 1
a1 = a1 >> 1
a0 - a1
if (a >= 0) jmp (else_0)
```

Kod nakon optimizacije:

```
if (a - b < 0) ...
```

Rezultat prevođenja:

```
a0 = a0 - a1
if (a >= 0) jmp (else_1)
```

5.3.3 Optimizacija kontrole toka programa

Harvard arhitektura većine DSP procesora nije efikasna kada je u pitanju kontrola toka programa. Prilikom pisanja koda upotrebom asemblerskog jezika to se jasno vidi kroz ograničenja instrukcijskog skupa i broja instrukcija koji je potreban za realizaciju uslovnih iskaza. Međutim, prilikom pisanja programa upotrebom C programskog jezika to nije očigledno, pa lepo struktuiran i čitljiv kod u C-u ne rezultuje uvek efikasnim kodom na DSP-u. Kada god postoje uslovi za to, potrebno je uprostiti uslovne konstrukcije kao što su komplikovani *switch-case* iskazi ili *if-else* strukture.

Primer:

```
enum OutputConfig { OUT_MONO,
     OUT_STEREO, OUT_4CH, OUT_3_1CH,
     OUT_5_1CH, OUT_5CH, OUT_7CH };
      ...
      switch (output_type)
      {
            case OUT_MONO:
                  value = 0x7a;
                  break;
            case OUT_STEREO:
                  value = 0x82;
                  break;
            case OUT_4CH:
                  value = 0xEB;
                  break;
            case OUT_3_1CH:
                  value = 0x11;
                  break;
            ...
      }
```

Kod nakon optimizacije:

```
enum OutputConfig { OUT_MONO,
     OUT_STEREO, OUT_4CH, OUT_3_1CH,
     OUT_5_1CH, OUT_5CH, OUT_7CH };
      ...
      int OCArray [7] = {0x7a, 0x82,
0xEB, 0x11 ... };
      ...
      value = OCArray[output_type];
```

Rezultat prevođenja:

```
    a0 & a0
    if (a == 0) jmp (case_0)
    uhalfword(a1) = (0x1)
    a0 - a1
    if (a == 0) jmp (case_1)
    uhalfword(a1) = (0x2)
    a0 - a1
    if (a == 0) jmp (case_2)
    uhalfword(a1) = (0x3)
    a0 - a1
    if (a == 0) jmp (case_3)
    jmp (__epilogue_104)
case_0:
    uhalfword(a0) = (0x7a)
    xmem[_value + 0] = a0h
    jmp (__epilogue_104)
case_1:
    uhalfword(a0) = (0x82)
    xmem[_value + 0] = a0h
    jmp (__epilogue_104)
case_2:
    uhalfword(a0) = (0xeb)
    xmem[_value + 0] = a0h
    jmp (__epilogue_104)
case_3:
    uhalfword(a0) = (0x11)
    xmem[_value + 0] = a0h
__epilogue_104:
    ret
```

Rezultat prevođenja:

```
    i0 = a0
    nop #empty cycle
    i0 = i0 + (_OCArray + 0)
    a0 = xmem[i0]
    xmem[_value + 0] = a0h
    ret
```

5.3.4 Optimizacija pristupa memoriji

5.3.4.1 Kopiranje struktura

Ova optimizaciona tehnika namenjena je optimizaciji potrošnje programske memorije, ali u određenim slučajevima može da doprinese i poboljšanju brzine izvršavanja koda. Ova tehnika podrazumeva da se uvek teži kopiranju čitavih struktura koristeći operator dodele između dve instance, ili njihovih delova koji se nalaze u memoriji na uzastopnim lokacijama. U tom slučaju, kompajler je u stanju da generiše hardversku petlju za kopiranje podataka. U nastavku je prikazano kopiranje strukture polje po polje i kopiranje koristeći operator dodele između struktura, koji je podržan od strane CCC2 kompajlera. Dati su rezultati prevođenja u slučaju kada su X i Y lokalne i globalne strukture.

Primer:

```
struct S {
      int a, b, c, d, e, f, g;
};
typedef struct S tS;

...
tS X, Y;
...
X.a = Y.a;
X.b = Y.b;
X.c = 1;
X.d = Y.d;
X.e = Y.e;
X.f = Y.f;
X.g = Y.g;
...
```

Kod nakon optimizacije:

```
...
X = Y;
X.c = 1;
...
```

Rezultat prevođenja za X i Y lokalno:

```
i7 = i7 + (0xe)
i0 = i7 - (7 - 0)
a0 = xmem[i0]
i0 = i7 - (14 - 0)
xmem[i0] = a0h
i0 = i7 - (7 - 1)
a0 = xmem[i0]
i0 = i7 - (14 - 1)
xmem[i0] = a0h
uhalfword(a0) = (0x1)
i0 = i7 - (14 - 2)
xmem[i0] = a0h
i0 = i7 - (7 - 3)
a0 = xmem[i0]
i0 = i7 - (14 - 3)
xmem[i0] = a0h
i0 = i7 - (7 - 4)
a0 = xmem[i0]
i0 = i7 - (14 - 4)
xmem[i0] = a0h
i0 = i7 - (7 - 5)
a0 = xmem[i0]
i0 = i7 - (14 - 5)
xmem[i0] = a0h
i0 = i7 - (7 - 6)
a0 = xmem[i0]
i0 = i7 - (14 - 6)
xmem[i0] = a0h
i7 = i7 - (0xe)
```

Rezultat prevođenja za X i Y lokalno:

```
    i7 = i7 + (0xe)
    i0 = i7 - (0x7)
    i1 = i7 - (0xe)
    do (0x7), label_end_11_0
label_begin_11_0:
    x0 = xmem[i0]; i0 += 1
label_end_11_0:
    xmem[i1] = x0; i1 += 1
    uhalfword(a0) = (0x1)
    i0 = i7 - (14 - 2)
    xmem[i0] = a0h
    i7 = i7 - (0xe)
```

Rezultat prevođenja za X i Y globalno:

```
    a0 = xmem[_Y + 0]
    xmem[_X + 0] = a0h
    a0 = xmem[_Y + 1]
    xmem[_X + 1] = a0h
    uhalfword(a0) = (0x1)
    xmem[_X + 2] = a0h
    a0 = xmem[_Y + 3]
    xmem[_X + 3] = a0h
    a0 = xmem[_Y + 4]
    xmem[_X + 4] = a0h
    a0 = xmem[_Y + 5]
    xmem[_X + 5] = a0h
    a0 = xmem[_Y + 6]
    xmem[_X + 6] = a0h
```

Rezultat prevođenja za X i Y globalno:

```
    i0 = (0) + (_Y)
    i1 = (0) + (_X)
    do (0x7), label_end_11_0
label_begin_11_0:
    x0 = xmem[i0]; i0 += 1
label_end_11_0:
    xmem[i1] = x0; i1 += 1
    uhalfword(a0) = (0x1)
    xmem[_X + 2] = a0h
```

Iz datog asemblerskog koda može se zaključiti da je u prvom slučaju za izvršenje neoptimizovanog koda neophodno 29 instrukcionih ciklusa, dok je za optimizovani kod potrebno 23 ciklusa. Isto tako, optimizovana je i programska memorija. Program zauzima 29 reči pre, a 8 nakon optimizacije. U drugom slučaju izvršavanje optimizovanog koda je sporije za 6 ciklusa, međutim, i dalje postoji ušteda programske memorije za 7 reči (14 pre, 7 posle optimizacije).

U narednom primeru polja *c* i *f* nisu postavljena, odnosno, njihova vrednost treba da ostane nepromenjena. U ovom slučaju optimizacija se može izvršiti promenom redosleda polja u strukturi, tako da se polja koja se kopiraju nalaze jedno iza drugog u memoriji. Na taj način se omogućava kopiranje polja u okviru petlje.

Primer:

```
      struct S {
            int a, b, c, d, e,
f, g;
      };
      typedef struct S tS;

      ...
      tS X, Y;
      ...
      X.a = Y.a;
      X.b = Y.b;
      X.d = Y.d;
      X.e = Y.e;
      X.g = Y.g;
      ...
```

Kod nakon optimizacije:

```
      struct S {
      int a, b, d, e, g, c, f; //promenjen
redosled polja strukture
      };
      typedef struct S tS;
      ...
      tS X, Y;
      ...
      int i;
      int* tmpX = &(X.a);
      int* tmpY = &(Y.a);
      for (i = 0 ; i < 5 ; i++)
            *tmpX++ = *tmpY++;
      ...
```

Rezultat prevođenja za X i Y lokalno:

```
i7 = i7 + (0xe)
i0 = i7 - (7 - 0)
a0 = xmem[i0]
i0 = i7 - (14 - 0)
xmem[i0] = a0h
i0 = i7 - (7 - 1)
a0 = xmem[i0]
i0 = i7 - (14 - 1)
xmem[i0] = a0h
i0 = i7 - (7 - 3)
a0 = xmem[i0]
i0 = i7 - (14 - 3)
xmem[i0] = a0h
i0 = i7 - (7 - 4)
a0 = xmem[i0]
i0 = i7 - (14 - 4)
xmem[i0] = a0h
i0 = i7 - (7 - 6)
a0 = xmem[i0]
i0 = i7 - (14 - 6)
xmem[i0] = a0h
i7 = i7 - (0xe)
```

Rezultat prevođenja za X i Y lokalno:

```
    i7 = i7 + (0xe)
    i0 = i7 - (7 - 0)
    i1 = i7 - (14 - 0)
    do (0x5), label_end_92
label_begin_92:
    a0 = xmem[i1]; i1 += 1
label_end_92:
    xmem[i0] = a0h; i0 += 1
for_end_0:
    i7 = i7 - (0xe)
```

Rezultat prevođenja za X i Y globalno:

```
a0 = xmem[_Y + 0]
xmem[_X + 0] = a0h
a0 = xmem[_Y + 1]
xmem[_X + 1] = a0h
a0 = xmem[_Y + 3]
xmem[_X + 3] = a0h
a0 = xmem[_Y + 4]
xmem[_X + 4] = a0h
a0 = xmem[_Y + 6]
xmem[_X + 6] = a0h
```

Rezultat prevođenja za X i Y globalno:

```
    i0 = (0) + (_X + 0)
    i1 = (0) + (_Y + 0)
    do (0x5), label_end_92
label_begin_92:
    a0 = xmem[i1]; i1 += 1
label_end_92:
    xmem[i0] = a0h; i0 += 1
for_end_0:
```

I u ovom primeru će se iskoristiti hardverska petlja sa dve instrukcije za kopiranje vrednosti. Kao i u prethodnom primeru za lokalne strukture, uočava se poboljšanje i brzine izvršavanja (22 ciklusa pre, 16 posle optimizacije) i potrošnje programske memorije (22 reči pre, 7 posle optmizacije). Za globalne strukture takođe dolazi samo do optimizacije potrošnje programske memorije (10 reči pre, 5 reči posle) dok se izvršenje usporava za 4 instrukcije (postavaljanje inicijalnih pokazivača i inicijalizacija petlje).

Može se uočiti da kopiranje bilo kog broja sukcesivnih lokacija će kao rezultat dati kod koji zauzima 5 reči (2 za postavljanje pokazivača, 1 instrukciju hardverske petlje i 2 instrukcije za čitanje i upis vrednosti unutar tela hardverske petlje).

5.3.4.2 Optimalno korišćenje memorijskih zona

Kao što je već rečeno, DSP CS48x sadrži dve odvojene memorijske zone kojima se pristupa pomoću dve odvojene magistrale podataka, što omogućava simultani pristup podacima u dve različite memorijske zone. Sa aspekta CCC2 kompajlera, potrebno je uvesti kvalifikatore memorijskih zona i time omogućiti precizno zauzimanje resursa obe memorijske zone. Prilikom uvođenja kvalifikatora potrebno je voditi računa da se podaci raspoređuju ravnomerno u obe memorijske zone, kako bi se izbegla pojava da je jedna memorijska zona prepunjena, a druga prazna. Takođe, poželjno je podatke nad kojima se vrši obrada rasporediti tako da se podaci koji se čitaju iz memorije ili upisuju u memoriju u približno istom trenutku nalaze u odvojenim memorijskim zonama. Ovakav pristup omogućava prevodiocu da generiše paralelno čitanje ili upis podataka. U suprotnom, ako se podaci nalaze u istoj memorijskoj zoni, proširenja DSP-a za paralelni pristup podacima ne mogu da se iskoriste.

Primer:

```
__memX fract* p_a = a;
__memY fract* p_b = b;
for(i = 0; i< 10; i++)
{
      res += *p_a++ *
*p_b++;
}
```

Rezultat prevođenja:

```
    i0 = (0) + (_a + 0)
    i4 = (0) + (_b + 0)
    do (0xa), label_end_92
label_begin_92:
    x0 = xmem[i0]; i0 += 1; y0 = ymem[i4];
i4 += 1
label_end_92: a0 += x0 * y0
```

5.4 Zadaci za samostalnu izradu

5.4.1 Zadatak 1: Profilisanje DSP aplikacije

U ovom zadatku potrebno je izvršiti procenu utrošenih resursa od strane aplikacije koja se izvršava na DSP uređaju. Potrebno je izvršiti procenu utroška memorije, utroška instrukcija funkcije obrade i funkcije za inicijalizaciju i locirati deo koda koji troši najviše ciklusa.

5.4.1.1 Postavka zadatka:

1. Otvoriti projekat *multitapEcho_model3* u okviru CLIDE razvojnog okruženja i prevesti ga.
2. Na osnovu generisane MAP datoteke izračunati ukupan utrošak programske memorije, i memorije u memorijskoj zoni X i Y, od strane modula.
3. U okviru *.groovy* skripte omogućiti profilisanje koda.
4. Pokrenuti aplikaciju na simulatoru i pustiti da se izvršava neko vreme. Nakon toga zaustaviti izvršavanje i proveriti izmerene vrednosti koje označavaju potrošnju procesorskih ciklusa.
5. U okviru funkcije koja troši najviše ciklusa izmeriti potrošnju delova funkcije upotrebom *cl_get_cycle_count* funkcije. Odrediti potencijalne kandidate za optimizaciju.

5.4.2 Zadatak 2: Optimizacija DSP aplikacije pisane upotrebom programskog jezika C

U ovom zadatku potrebno je izvršiti dodatno prilagođenje C koda kako bi CCC2 programski prevodilac generisao što efikasniji kod. Potrebno je među navedenim tehnikama optimizacije koda odabrati one koje su primenljive na datu aplikaciju. Nakon svake izmene nad kodom izvršiti proveru da li kod i dalje daje ispravne rezultate, kao i ponovnu procenu utroška resursa kako bi se dobila informacija da li je promena dovela do više ili manje efikasnog koda.

Primeri prilagođavanja koda koji omogućavaju optimizacije od strane kompajlera, dati su u okviru projekta *ccc2Lessons,* datoteka *lesson05Optimizations* .

5.4.2.1 Postavka zadatka:

1. Otvoriti projekat *multitapEcho_model3* u okviru CLIDE razvojnog okruženja i prevesti ga.
2. Zameniti poređenja u sklopu svih *if* iskaza tako da se umesto relacija između elemenata ispituje relacija rezultata oduzimanja sa nulom.

3. Čitanjem generisane *output\src\main.s* datoteke ispitati da li su sve petlje unutar funkcija obrade prevedene u hardverske petlje. Ukoliko nisu, prepraviti C kod tako da budu.
4. Zameniti pristupe poljima struktura pristupom elementima preko pokazivača.
5. Rasporediti promenljive u memorijske zone tako da se unutar obrade omogući paralelno učitavanje ili upis podataka. Da li je uspešno generisano paralelno čitanje ili pisanje proveriti čitanjem generisane *main.s* datoteke.

Nakon svakog koraka izvršiti procenu utrošenih resursa.