VEŽBA 4 – Izrada DSP aplikacije 2. deo – proširenja C kompajlera za platforme sa ograničenim resursima

4.1 Uvod

Razvoj softvera u nekom od viših programskih jezika (koji su nezavisni od fizičke arhitekture), smanjuje troškove realizacije programskih sistema u svim fazama metodologije razvoja aplikacija za ciljni procesor. Neke od prednosti programiranja DSP arhitektura u višim programskim jezicima:

- programiranje je bliže domenu problema: moguće je programirati bez znanja o resursima fizičke arhitekture. Samim tim nema grešaka usled neodgovarajuće raspodele resursa, kao što su akumulatori, registri, memorijske zone i sl…,
- kod je pregledan, lako se čita i razume, što je važno kod održavanja programa i rada u timu,
- kod je lako prenosiv: uz minimalne modifikacije i odgovarajućeg prevodioca dobija se izvršni kod za drugu platfromu.

Programski jezik C je dobar izbor zbog svoje rasprostranjenosti, kao i zbog svog relativno niskog nivoa koji ostavlja programeru veliki prostor za optimizacije. U prethodnoj vežbi je prikazana metodologija razvoja algoritma za digitalne signal procesore, kao i načini ispitivanja rešenja u skladu sa predloženom metodologijom. Funkcionalne optimizacije koda, koje se primenjuju u okviru Modela 2, prilagođavaju C kod nekim od proširenja digitalnih signal procesora, a omogućavaju prevođenje i izvršavanje na opštenamenskim platformama (što olakšava uočavanje grešaka i poređenje sa referentnim kodom).

Ipak, programski jezik C ne podržava neke od važnih proširenja DSP procesora, pre svega aritmetiku u nepokretnom zarezu i memorijske zone. Da bi dobili kod koji je izvršiv na DSP platformi, potrebno je prevesti ga namenskim C kompajlerom koji ima podršku za aritmetiku u nepokretnom zarezu i omogućava raspoređivanje podataka u skladu sa memorijskim zonama. Ovo podrazumeva i dodatna prilagođenja koda, koja treba da olakšaju prevodiocu generisanje optimalnog koda za ciljnu platformu (Model 3).

Tokom izrade ove vežbe, studenti će se upoznati sa C kompajlerom razvijenim za arhitekturu CS48x, pod nazivom CCC2 (*Cirrus Logic C Compiler 2*) i C proširenjima koje ovaj prevodilac podržava. Biće prikazano kako se u okviru C referentog koda:

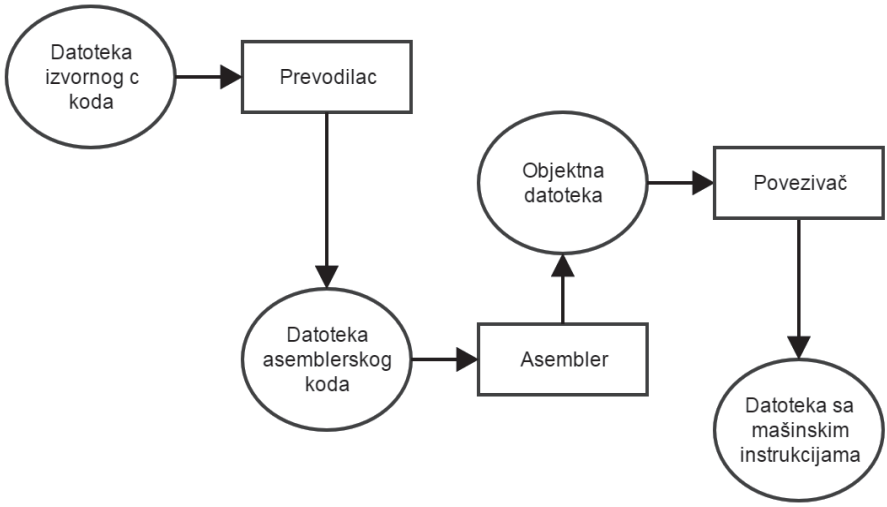
- rukuje sa memorijskim prostorima,
- koriste tipove u nepokretnom zarezu,

- rukuje adresnim generatorom (koristi modulo adresiranje),
- realizuju delove koda upotrebom ASM jezika,
- pozivaju funkcije realizovane u ASM i pripremaju ASM funkcije za poziv iz programskog jezika C.

4.2 Programski prevodilac CCC2 (*Cirrus Logic C Compiler 2*)

Kao što je rečeno, programski jezik C ne podržava aritmetiku u nepokretnom zarezu i memorijske zone. Zbog ovoga je ISO/IEC komitet uveo proširenje C standarda za namenske procesore, pod nazivom ISO/IEC TR 18037. Primer C kompajlera koji podržava ovaj prošireni standard jeste *Cirrus Logic C Compiler 2* (u daljem tekstu CCC2) namenjen prevođenju koda za *Cirrus Logic* CS47x, CS48x i CS49x familije čipova. Ovaj prevodilac razvijen je u celosti na Institutu za računarsku tehniku i računarske komunikacije u Novom Sadu. Detaljan opis proširenja koje podržava CCC2 kompajlera nalazi se u uputstvu za korišćenje CCC2 (*C-Compiler User's Manual.pdf*) koje je distribuirano u okviru paketa softverskih alata *Cirrus Logic* i nakon instalacije se nalazi na putanji „<*Instalacioni direktorijum*>*doc*“

Tok prevođenja kod CCC2 kompajlera sastoji se iz sledećih koraka: CCC2 prevodi C kod i kao rezultat prevođenja daje datoteke sa asemblerskim kodom za ciljnu arhitekturu. Nakon toga CCC2 poziva asembler koji na osnovu asemblerskog koda generiše objektne datoteke. Poslednji korak jeste pozivanje povezivača (linkera) koji kao izlaz daje softversku biblioteku ili izvršnu datoteku. Izvršna datoteka se upisuje u memoriju DSP uređaja nakon čega sledi njeno izvršavanje, slika 4.1.



.

Slika 4.1 - Dijagram toka prevođenja izvornog koda

4.2.1 Opcije CCC2 prevodioca

Format komandne linije kompajlera je:

```
ccc2 -<opcije> <C datoteka sa izvornim kodom>
```

Polje opcije može da sadrži nijednu, jednu ili više kontrolnih direktiva, tabela 4.1, od kojih svaka mora da počinje znakom "-„.

Tabela 4.1 - Kontrolne direktive CCC-a.

Direktiva:	Opis:
-h	Ispis pomoćnih informacija. (help)
-v	Ispis informacija o verziji.
-w	Potiskivanje upozorenja.
-S	Prevođenje izvršiti samo do dela kada je izlaz .s datoteka. (Zaustaviti pre nego što se pozovu asmebler i povezivač.)
-c	Prevođenje izvršiti samo do dela kada je izlaz .o datoteka. (Zaustaviti nakon što se pozove asmebler ali pre nego što se pozove povezivač.)
-I<include_path> / -I <include_path>	Putanja do direktorijuma u kojem će prevodilac da traži .h datoteke.
-D<symbol> / -D <symbol>	Definisani simbol.
-o<filename> / -o <filename>	Ime izlazne datoteke. (Ako se ovo ne navede, koristiće se ime ulazne datoteke sa promenjenom ekstenzijom.)
-femit-asm-struct	U .s datoteku upisati asemblerske .struct definicije za struct tipove koji se koriste u C kodu.
-fcse	Koristiti uklanjanje zajedničkih podizraza.
-fslt	Koristiti tablu vrednosti za pomeranje.
-fprc	Koristiti propagiranje konstanti i evaluaciju konstantnih izraza.
-fcompact	Koristiti kompaktor za paralelizaciju.
-fconst-pool[=M]	Napraviti skup konstanti u programskoj memoriji za brži pristup. M predstavlja memorijsku zonu u kojoj će se skup nalaziti: X, Y, ili P. Ako se M izostavi podrazumevana je P memorija.
-fif	Pokušati više različitih strategija prevođenja i izabrati najbolji rezultat. (Produžava vreme prevođenja.)
-fno-hw-loop-detection	Ne pretvarati petlje u hardverske.
-flimitBbSize[=N]	Interno ograničiti veličinu osnovnih blokova. N označava maksimalnu veličinu bloka. Manji broj skraćuje vreme prevođenja ali povećava količinu koda koji će biti proizveden. Ako se N ne navede, koristi se podrazumevana vrednost od

	150. Ako kod sadrži veći broj dužih blokova ovo podešavanje može znatno da skrati vreme prevođenja.
-fwrapv	Definiše prekoračenje kod označenih celih brojeva kao obmotavanje (u suprotnom prekoračenje kod označenih celih brojeva dovodi do nedefinisanog stanja). Proizvodi malo veći kod.
-funsafe-loop-optimizations	Pretpostaviti da broj iteracija petlje nikada neće biti 0. (Ovo omogućava pravljenje hardverskih petlji u nekoliko dodatnih slučajeva, ali treba oprezno korsititi jer može da dovede do pogrešnog izvršenja u slučaju kada broj iteracija jeste 0.)
-fno-loop-invar-opt	Isključiti optimizaciju invarijatnog koda. (Automatsko pomeranje koda iz petlji u slučaju da je nezavistan od te petlje.)
-falias-analysis[=N]	Koristiti informacije dobijene analizom aliasa. N predsavlja koliko duboko na steku poziva treba ići prilikom analize poziva funkcija (-1 znači da nema ograničenja). Povećanje ograničenja za rezultat ima precizniju analizu ali produžava vreme prevođenja. Podrazumevana vrednost je 1.
-fnfs	Koristiti statički ram stek. Ovo podešavanje je validno samo ako u program nema rekurzije, bilo neposredne ili posredne.
-wpo	Optimizacija celog programa. Uključiti ukoliko želite da se sve datoteke izvornog koda prevode odjednom. (Ovo dovodi do dužeg prevođenja i nemogućnosti inkrementalnog prevođenja)
-finline-functions	Automatski pronaći funkcije koje su pogodne za učešljavanje . Dostupno samo uz –wpo.
-fno-inline	Ne učešljavati funkcije, odnosno ignorisati *inline* atribut funkcija.
-g	Zapisati DWARF2 informacije za kontrolisano izvršavanje programa.
-esc	.s datoteku smestiti u isti direktorijum gde se nalazi izvorna .c datoteka, nezavisno od –o podešavanja.
-ida	Ignorisati debug_on i debug_off atribute funkcija.
-ira	Koristiti međuproceduralnu dodelu resursa.
-edi-lines	U .s datoteci u vidu komentara ispisati odgovarajuće brojeve linija uz kod koji potiče od njih.
-emit-hints	Ispisati savete za optimizaciju, kako izmeniti kod da bi se dobio bolje optimizovan rezultat.
-Cl	U .s datoteci u vidu komentara ispisati liniju C koda od koje potiče odgovarajući blok asemblerskih funkcija. Važi samo uz –g opciju.
-fseparateVarSegments	Upisivati svaku globalnu promenljivu u zaseban segment u .s datoteku. Imena segmenta će biti imena promenljive sa prefiksom koji označava memorijsku zon u kojoj se nalazi.

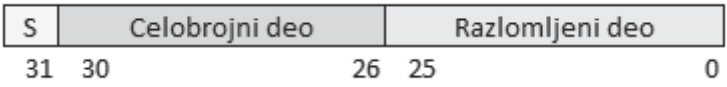
-cdl	Napraviti spisak zavisnih datoteka. Bitno za inkrementalni process prevođenja.
-ruf	Prijaviti grešku za pozive nedeklarisanih funkcija.
-fno0init	Preskočiti inicijalizaciju objekata osim ako su eksplicitno inicijalizovani. Korisno za smanjenje veličine .uld datoteka.

4.2.2 Tipovi podataka u nepokretnom zarezu

CCC2 koristi aritmetiku u nepokretnom zarezu, dok aritmetika sa pokretnim zarezom nije podržana. Tipovi podataka u nepokretnom zarezu koji su deklarisani *Embedded C* proširenjem (ISO/IEC DTR 18037) podržani su od strane CCC2 kompajlera. Spisak podržanih tipova i njihova preciznost data je u tabeli 4.2. Za predstavu formata brojeva u nepokretnom zarezu korišćena je sledeća notacija:

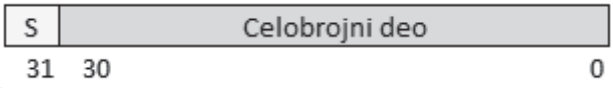
[s][m].[n]

gde simbol *s* označava da li je postoji bit znaka, *m* broj bita sa kojim je predstavljen celobrojni deo broja i *n* broj bita sa kojim je predstavljen razlomljeni deo broja. Ukoliko *m* ili *n* nisu prisutni, podrazumeva se vrednost 0. Primer formata brojeva predstavljen ovom notacijom dat je na slici 4.2. Dati primer odgovara formatu *s5.26*.



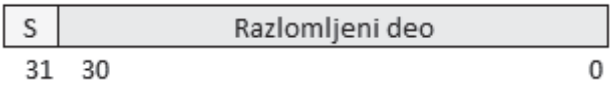
Slika 4.2 - Prikaz tipa u nepokretnom zarezu sa formatom s5.26

Opseg vrednosti za prikazani format jeste [-32.0, 32.0), a preciznost (razlika dva susedna broja) iznosi 2^{-26}. Format tridesetdvobitnog označenog celobrojnog tipa (*int* kod arhitekture CS48x) predstavljen u pomenutoj notaciji jeste s31.



Slika 4.3 - Prikaz tipa u nepokretnom zarezu sa formatom s31

Na slici 4.4 prikazan je tip podataka sa formatom *s.31*, čiji je opseg [-1.0, 1.0). Preciznost kod ovakvog tipa jeste 2^{-31}.



Slika 4.4 - Prikaz tipa u nepokretnom zarezu sa formatom s.31

Tabela 4.2 - Tipovi u aritmetici sa nepokretnim zarezom

Sintaksa			Veličina u bitima	Format
signed	short	_Fract	32	s.31
signed		_Fract	32	s.31
signed	long	_Fract	64	s.63
unsigned	short	_Fract	32	0.31
unsigned		_Fract	32	0.31
unsigned	long	_Fract	64	0.63
signed	short	_Accum	40	s8.31
signed		_Accum	40	s8.31
signed	long	_Accum	72	s8.63
unsigned	short	_Accum	40	9.31
unsigned		_Accum	40	9.31
unsigned	long	_Accum	72	9.63

Tipovi podataka počinju sa znakom "_" iza kojeg sledi veliko slovo. U zaglavlju *stdfix.h* su definisana alternativna imena (*aliases*) tako da se umesto *_Fract* i *_Accum* mogu koristiti *fract* i *accum*:

primer.c:

```
#include <stdfix.h>

accum foo (fract a, fract b)
{
    return a*b;
}
```

primer.s:

```
    .public _foo
    .extern __laccum_sat_accum
    .code_ovly


_foo
    x1 = a0
    x0 = a1
    a0 = x1 * x0
    call (__laccum_sat_accum)
    ret
```

Sa stanovišta arhitekture CS48x preporučeno je koristiti sledeće tipove:

- *long accum* (odgovara akumulatoru u CS48x),
- *fract* (odgovara registru podataka u CS48x),
- *long fract* (odgovara XY paru registara podataka u CS48x).

Ostali tipovi iz tabele 4.2 su podržani ali će operacije nad tim tipovima zahtevati više instrukcija te ih treba izbegavati.

4.2.3 Konstante u nepokretnom zarezu

Konstante se definišu vrednošću i sufiksom koji određuje tip konstante (tabela 4.3). Konstante tipa *_Fract* i *unsigned _Fract* je moguće definisati i heksadecimalnom vrednošću pod uslovom da se za to iskoristi odgovarajući sufiks.

Tabela 4.3 - Konstante u aritmetici sa nepokretnim zarezom

Tip konstante			Sufiks
signed	shor	_Fract	hr
signed		_Fract	r
signed	long	_Fract	lr
unsigned	shor	_Fract	uhr
unsigned		_Fract	ur
unsigned	long	_Fract	ulr
signed	shor	_Accum	hk
signed		_Accum	k
signed	long	_Accum	lk
unsigned	shor	_Accum	uhk
unsigned	shor	_Accum	uk
unsigned	long	_Accum	ulk

4.2.4 Operacije nad podacima u nepokretnom zarezu

Kao što je opisano u poglavlju koje se bavi prenosom podataka u i iz akumulatora, zaokruživanje se obavlja postavljanjem vrednosti *mr_sr* registra željene vrednosti. Vrednost mr_sr registra je moguće promeniti i očitati pomoću *set_mr_sr* i *get_mr_sr* funkcija deklarisanih u *stdfix.h* zaglavlju.

Kako definiše C standard, slučaj kada se kao operandi neke operacije pojave izrazi koji su različitog tipa, rešava se na sledeći način:

- nađe se zajednički osnovni tip svih izraza i zatim se svaki izraz pretvara u taj tip,
- operacija se obavi i njen rezultat je tog zajedničkog tipa,
- na kraju se rezultat pretvara iz zajedničkog tipa u tip izraza u koji se upisuje rezultat.

Za operacije koje sadrže operande nekog od tipova nepokretnog zareza važi drugačije pravilo:

- ako su oba operanda nekog od tipova nepokretnog zareza, rezultat je neki od tipova nepokretnog zareza koji može da primi sve bite rezultata

*fract * fract = long acum*

- ako je kod operacije množenja jedan operand celobrojnog tipa, onda se ne primenjuje nikakvo pretvaranje već se odrađuje specijalna operacija množenja koja daje željeni rezultat

int a = 2;
fract b = 0.25r;
*b = b * a;*
Rezultat je 0.5.

Nekada je neophodno umesto pretvaranja vrednosti iz jednog tipa u drugi samo preslikati bite tako da heksadecimalna predstava vrednosti ostane ista. U tabelama 4.4 i 4.5 su dati opisi funkcija koje vrše opisano preslikavanje. Predstavljene funkcije su deklarisane u *stdfix.h* zaglavlju.

Tabela 4.4 - Preslikavanje tipa _Fract u tip int

Tip Parametra	Tip Vraćene Vrednosti	Naziv Funkcije
signed _Fract	signed int	bitsr
unsigned _Fract	unsigned int	bitsur
signed short _Fract	signed int	bitshr
unsigned short _Fract	unsigned int	bitsuhr

Tabela 4.5 - Preslikavanje tipa int u tip _Fract

Tip Parametra	Tip Vraćene Vrednosti	Naziv Funkcije
signed int	signed _Fract	rbits
unsigned int	unsigned _Fract	urbits
signed int	signed short _Fract	hrbits
unsigned int	unsigned short _Fract	uhrbits

4.2.5 Kvalifikatori memorijskih zona

Kao što je već rečeno, procesor CS48x ima dve memorijske zone za podatke, X i Y, i programsku memoriju P. CCC2 podržava ove memorijske zone tako da se prilikom deklarisanje promenljive može specifikovati kvalifikator koji određuje memorijsku zonu kojoj će promenljiva pripadati. Ukoliko kvalifikator nije definisan CCC2 pretpostavlja da je reč o memorijskoj zoni X. Spisak kvalifikatora dat je u tabeli 4.6.

Tabela 4.6 - Kvalifikatori

Memorijska zona	Kvalifikator
X	__memX
Y	__memY
P	__memP
XY (L)	__memXY

4.2.5.1 Deklaracija globalnih promenljivih

Memorijska zona može biti deklarisana samo za globalne promenljive, što znači da se promenljivama unutar funkcija ne mogu dodeliti kvalifikatori osim ukoliko nisu eksplicitno deklarisane kao *static* ili *extern*.

Primeri:

- ukoliko nije specifikovana zona globalna promenljiva će biti smeštena u X zonu:

primer.c:

```
int global = 0;

int main(void)
{
    return global;
}
```

primer.s:

```
    .public _global
    .public _main
    .xdata_ovly
_global:
    .dw (0x0)
    .code_ovly

_main
    a0 = xmem[_global + 0]
    ret
```

- korišćenje kvalifikatora:

primer.c:

```
__memY int global = 0;

int main(void)
{
    return global;
}
```

primer.s:

```
    .public _global
    .public _main
    .ydata_ovly
_global:
    .dw (0x0)
    .code_ovly


_main
    a0 = ymem[_global + 0]
    ret
```

- definisanje pokazivača na memorijsku lokaciju:

primer.c:

```
__memY int global = 0;
__memY int * __memX pglobal =
&global;

int main(void)
{
    return *pglobal;
}
```

primer.s:

```
    .public _global
    .public _pglobal
    .public _main
    .ydata_ovly
_global:
    .dw (0x0)
    .xdata_ovly
_pglobal:
    .dw _global
    .code_ovly

_main
    i0 = xmem[_pglobal + 0]
    nop #empty cycle
    a0 = ymem[i0]
    ret
```

Prilikom upotrebe kvalifikatora memorijskih zona u kombinaciji sa pokazivačima treba obratiti pažnju na poziciju kvalifikatora u odnosu na karakter '*'. U slučaju kada se kvalifikator nalazi ispred, on predstavlja deo tipa, odnosno, pokazuje u kojoj memorijskoj zoni se nalazi podatak na koji pokazivač pokazuje. U suprotnom, kada se nalazi posle karaktera '*', daje informaciju o memorijskoj zoni u kojoj se nalazi vrednost samog pokazivača.

4.2.5.2 Deklaracija lokalnih promenljivih

Prilikom definisanja lokalne promenljive ukoliko je deklarisan kvalifikator memorijske zone, CCC2 će prijaviti grešku i prekinuti prevođenje. Pošto pokazivači, kako im samo ime kaže, pokazuju na memoriju procesora, potrebno je definisati i na koju memoriju pokazuje neki pokazivač. To je jedini slučaj kada se kvalifikator memorijske zone može koristiti i kod lokalnih promenljivih; pri tom treba imati na umu prethodni primer, odnosno činjenicu da kod pokazivača možemo da iskoristimo kvalifikator na dva različita načina. U slučaju lokalnih promenljivih, od interesa je memorijska zona na koju pokazivač pokazuje (kvalifikator ispred '*').

Praksa je da se uz svaku deklaraciju pokazivača navede i memorijska zona na koju pokazuje. U slučaju da se to ne uradi, CCC2 će objaviti upozorenje. I pored ovog upozorenja, kod će i dalje biti uspešno preveden, a podrazumevana memorijska zona je X.

primer.c:

```
__memY int data1 = 1;
__memX int data2 = 2;

int main (void)
{
    __memY int * localP1 = &data1;
    int * localP2 = &data2;

    return *localP1 + *localP2;
}
```

primer.s:

```
    .public _data1
    .public _data2
    .public _main
    .ydata_ovly
_data1:
    .dw (0x1)
    .xdata_ovly
_data2:
    .dw (0x2)
    .code_ovly
_main
    a0 = ymem[_data1 + 0]
    a1 = xmem[_data2 + 0]
    a0 = a0 + a1
    ret
```

4.2.5.3 Promena memorijske zone promenljive

Kada se jednom deklariše kvalifikator memorijske zone, zona se ne može promeniti korišćenjem eksplicitne promene tipa. Eksplicitnom promenom može se izvršiti redirekcija na drugu memorijsku zonu pokazivača, što može biti veoma korisno.

primer.c:

```
#include<stdfix.h>

__memXY long fract whole =
0.27192006LR;
__memX fract high;
__memY fract low;


void main(void)
{
    long accum accumulator;
    __memXY long fract *pointer;

    pointer = &whole;
    accumulator = *pointer;
    whole = accumulator;
    high = *(__memX fract*)pointer;
}
```

primer.s:

```
    .public _high
    .public _low
    .public _whole
    .public _main
    .xdata_ovly
_high
    .bsc (0x1), 0x00000000
    .ydata_ovly
_low
    .bsc (0x1), 0x00000000
    .data_ovly
_whole
    .dw  (0x22ce46ca),(0x69c61751)
    .code_ovly
_main:
    i0 = (0) + (_whole)
    a0 = xymem[i0]
    a0 = long(a0)
    xymem[_whole + 0] = a0
    a0 = xmem[i0]
    xmem[_high + 0] = a0h
    ret
```

4.2.6 Korišćenje kružnih bafera

Modulo adresiranje, opisano u poglavlju sa opisom arhitekture, omogućava funkcija CIRC_INC koja se nalazi unutar biblioteke *circbuff*. Režimi rada prilikom modulo adresiranja dati su u tabeli 4.7.

Tabela 4.7 - adresni režimi

Simbol	Adresni režim
MOD_LIN	Linearno adresiranje
MOD_4	Moduo 4
MOD_8	Moduo 8
MOD_16	Moduo 16
MOD_32	Moduo 32
MOD_64	Moduo 64
MOD_128	Moduo 128
MOD_256	Moduo 256
MOD_512	Moduo 512
MOD_1024	Moduo 1024
MOD_2048	Moduo 2048
MOD_4096	Moduo 4096
MOD_8192	Moduo 8192
MOD_16384	Moduo 16384
MOD_32768	Moduo 32768
MOD_BITREV	Bit inverzno adresiranje

primer.c:

```
#include <circbuff.h>
int *foo(int *p)
{
    return CIRC_INC(p, MOD_16 + 2);
}
```

primer.s:

```
    .public _foo
    .code_ovly


_foo:
    nm0 = (0x3002)
    nop #empty cycle
    i0 += n
    nm0 = (0x0)
    a0 = i0
    ret
```

Da bi se koristilo modulo adresiranje, memorijski nizovi se moraju nalaziti u memoriji poštujući pravila opisana u poglavlju 2.4.1 dokumenta koji sadrži opis arhitekture procesora. Memorija može biti zauzeta iz asemblerskog koda ili C koda.

Ako se zauzimanje obavlja u C kodu, treba koristiti direktive koje su date u sledećem primeru:

primer.c:

```
#include <stdfix.h>
#include <circbuff.h>
__memX fract __attribute__((__aligned__(16)))
a[16];
__memY fract __attribute__((__aligned__(32)))
b[16];

void foo(__memX fract *p2, __memY fract *p3)
{
    int i;
    fract y0 ;
    for (i=0;i<16;i++)
    {
        y0 = *p3; p3 = CIRC_INC(p3, MOD_16 + 4);
        *p2++ = y0;
    }
}

void main()
{
    foo(a, b);
}
```

primer.s:

```
    .public _a
    .public _b
    .public _foo
    .public _main
    .xdata_ovly align 16
_a
    .bsc            (0x10),
0x00000000
    .ydata_ovly align 32
_b
    .bsc            (0x10),
0x00000000
    .code_ovly



_foo:
    uhalfword(a0) = (0x1)
    do              (0x10),
label_end_92
label_begin_92:
    nm1 = (0x3004)
    a1 = ymem[i1]
    i1 += n
    nm1 = (0x0)
    xmem[i0] = a1h
    a1 = i0
    a1 = a1 + a0
label_end_92:
    AnyReg(i0, a1h)
for_end_0:
    ret

_main:
    i0 = (0) + (_a)
    i1 = (0) + (_b)
    call (_foo)
    ret
```

4.2.7 Atributi Funkcije

Svakoj funkciji je moguće dodeliti jedan ili više atributa koji određuju kako će CCC2 prevesti datu funkciju. Lista podržanih atributa je data u tabeli 4.9.

U slučaju da je uključena opcija *–ida,* atributi *debug_on* i *debug_off* se ignorišu.

Prilikom korišćenja *fg_call* i *bg_call* atributa, moraju se definisati i globalni nizovi *__c_stack_fg* i *__c_stack_bg* koji će predstavljati memorijski stek za funkcije pisane u C-u. Element ovih nizova je tipa *int*, dok im se veličina definiše u zavisnosti od potrebe.

Tabela 4.9 - Lista podržanih atributa funkcije

Sintaksa Atributa	Opis
__attribute__((debug_on))	Emituju se debug informacije za datu funkciju (zanemaruje se –g opcija)
__attribute__((debug_off))	Ne emituju se debug informacije za datu funkciju (zanemaruje se –g opcija)
__attribute__((fg_call))	Funkcija predstavlja ulaznu tačku za *foreground thread* modula
__attribute__((bg_call))	Funkcija predstavlja ulaznu tačku za *background thread* modula
__attribute__((fg_primitive_call))	Funkcija predstavlja ulaznu tačku za *foreground thread* Composer primitive
__attribute__((bg_primitive_call))	Funkcija predstavlja ulaznu tačku za *background thread* Composer primitive

primer.c:

```
void __attribute__((fg_call)) main()
{
}
```

primer.s:

```
        .extern ___C_STACK_FG
        .public _main
        .extern cl_clearNM0_7
        .code_ovly

_main:
    call (cl_clearNM0_7)
    i7 = (0) + (___C_STACK_FG)
    ret
```

4.3 Zadaci za samostalnu izradu

4.3.1 Zadatak 1: Primeri upotrebe proširenja koje nudi CCC2 programski prevodilac

U ovom zadatku na priloženom primeru upoznaje se sa praktičnom primenom CCC2 programskog prevodioca i proširenjima koja on nudi.

4.3.1.1 Postavka zadatka:

1. Pokrenuti razvojno okruženje CLIDE. U radno okruženje uvucite priloženi projekat *ccc2Lessons.*
2. Pronaći datoteku izvornog koda *ccc2LessonsMain.c* u i u njoj postaviti tačku prekida na liniji 37.
3. Pokrenuti priloženi projekat u režimu kontrolisanog izvršavanja (obeležite opciju *Start debugging during boot* i pritisnite dugme *Slave Boot*).
4. Nakon pokretanja programa pritisnuti taster *f5* kako bi se izvršavanje nastavilo do tačke prekida.
5. Od tačke prekida koristeći taster *f11* uđite u telo svake od funkcija. Svaka funkcija predstavlja jednu lekciju o specifičnostima koje nudi CCC. Lekcije su:
    a. *lesson01Introduction* – uvod, prikaz različitih podešavanja koja mogu da se koriste kroz komandnu liniju ili u okviru podešavanja projekta,
    b. *lesson02FixedPointTypes* – opis primene tipova u nepokretnom zarezu,
    c. *lesson03VariablesAndMemory* – skladištenje vrednosti u memoriji, pokazivači, kružni baferi,
    d. *lesson04BasciFileIO* – osnove rukovanja datotekama.
6. Svaka funkcija sadrži kratak opis u okviru komentara u telu finkcije.
7. Koristeći mogućnosti koje nudi razvojno okruženje (*RegisterView*, *MemoryView*, *ExpressionView*...) proveriti rezultate operacija.

4.3.2 Zadatak 2: Realizacija bloka za dodavanje višestrukog eho efekta audio signalu – Model 3

U okviru ovog zadatka potrebno je realizovani Model 2, iz prethodne vežbe, prilagoditi za prevođenje upotrebom CCC2 prevodioca.

4.3.2.1 Postavka zadatka:

1. Otvoriti projekat *multitapEcho_model3* u okviru CLIDE razvojnog okruženja. Ovaj projekat je *Standalone ULD* projekat, namenjen izvršavanju na

simulatoru. U okviru njega je realizovano čitanje i upis podataka iz datoteke.

2. Prekopirati kod koji se odnosi na obradu iz Modela 2 u Model 3.
3. U okviru *main* izvršiti poziv funkcija za inicijalizaciju i obradu kao što je to bilo urađeno u Modelu 2.
4. Podacima koji se nalaze u memoriji pridružiti odgovarajući kvalifikator memorijske zone. Voditi računa da se prilikom svake dodele adrese pokazivaču memorijski prostori slažu.
5. Postaviti veličinu memorijskog niza za skladištenje zakasnelih odbiraka na prvi veći broj koji predstavlja stepen broja 2.
6. Poravnati memorijski niz za skladištenje zakasnelih odbiraka na memorijsku lokaciju deljivu sa njegovom veličinom.
7. Iskoristiti proširenje CCC2-a za rukovanje kružnim baferom umesto operacije %.
8. Proširiti skriptu *Test.bat* pozivom CLIDE simulatora iz komandne linije.
9. Proširiti skrptu *Test.bat* tako da se vrši poređenje izlaznih datoteka za module 0, 1, 2 i 3 za minimum 3 različite ulazne datoteke.