SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 2917

OPTIMIRANJE KODA ZA NAVIGACIJU U MIKROKONTROLERU

Marin Bulatović

Zagreb, srpanj 2013.

Sadržaj

1. Uvod

Programiranje pic-eva ne može se izvesti bez potrebne strategije tj. planiranja. Brojni nedostaci kao što su ograničenje memorije i ograničenost kompajlera mogu programeru uzrokovati velike probleme, čak do te granice da nije u stanju izvršiti sve što je zamislio. Kad imate jako malo memorije svaki algoritam optimizacije ili funkcije koje određeni posao obavljaju brže nego standardne su vam od velikog značaja. U ovom slučaju, ako koristimo mikokontroler koji prima i obrađuje podatke sa senzora (dakle ne koristi brojeve velikog raspona), fixed point aritmetika pruža nam znatno ubrzanje naspram standarnog korištenja float point aritmetike.

Što nam to omogućava? Brže izvođenje često korištenih matematičkih funkcija može uvelike ubrzati sveukupno vrijeme računanja kalibracije. Predviđanje je da bi to ubrzanje moglo iznositi oko 20%. No, s druge strane, moramo i uračunati hardevrsku komponentu. Kako bi implementirali sve funkcije koje koriste fixed point aritmetiku potrebno je u program uključiti nekoliko header-a i soruce file-ova. Sve u svemu, mala cijena memorije za osjetno ubrzanje programa.

Na kraju se treba istaknuti još jedna bitna stavka. Program razrađen i napisan u float point aritmetici može biti nekompatibilan za prijenos na fixed point aritmetiku. Za prebacivanje na fixed point potrebno je izmjeniti tip svih varijabli koje se koriste za matematičke proračune te sve matematičke operacije. Najveći problem predstavlja nekonzistentnost broja parametara i njihov redosljed u matematičkim funkcijama (primjer: u float point aritmetici možemo zbrajati koliko god parametara želimo, dok funkcija fix16_add ima samo 2 parametra koji se mogu zbrojiti).

2. Uvod u fixed point aritmetiku

Kolekcija od N (N je pozitivni broj) binarnih znamenaka ima 2^N mogućih stanja. U najopširnijem slučaju, dopuštamo da ta stanja reprezentiraju bilo što smisleno. Naprimjer: studente na sveučilištu, vrste biljaka, kemijski elementi... Iako većina ljudi gleda na ta stanja kao pozitivne brojeve, značenje N-bitne binarne riječi potpuno ovisi o njenoj interpretaciji, tj. kontekstu u kojoj te riječi koristimo. Ovdje će te riječi reprezentirati određene podskupove racionalnih brojeva. Da se podsjetimo, racionalni brojevi su skupovi brojeva koje izražavamo kao a/b, gdje su a, b € Z, Z ≠ 0. Podskup brojeva koji nas zanima je $b = 2^n$. Također ćemo ovaj podskup promatrati kao skup gdje svi brojevi imaju jednak broj binarnih znamenki i gdje svi imaju na istom mjestu postavljenu biranu točku (fixed binary point).

2.1. Unsigned Fixed Point Racionalni brojevi

N-bitna binarna riječ, interpretirana kao unsigned Fixed-Point Racionalni broj može poprimati vrijednosti iz skupa P ne-negativnih racionalnih brojeva:

$$P = \{p/2^b \mid 0 \le p \le 2^N - 1, p \in Z\}.$$

Primjetimo da skup P ima 2^N elemenata. Definiramo pojam *U(a, b)*, gdje je a = N – b. Unutar tog pojma N-ti bit, brojeći s desna na lijevo počevši od nultog bita, ima težinu $2^n/2^b = 2^{n-b}$. Slično kao i u svakodnevnoj decimalnoj notaciji, binarna toča je između bita sa težinom n = b i onoga njemu s desna. To se ponekad označava kao implicirana binarna točka. *U(a, b)* notacija predstavlja izraz sa *a* bitova lijevo od binarne točke i *b* bitova desno od binarne točke. Vrijednost određenog N-bitnog binarnog broja x u *U(a, b)* notaciji dana je ovim izrazom:

$$x = (1/2^b) \sum_{n=0}^{N-1} 2^n x_n$$

gdje x_n predstavlja n-ti bit od x. Veličina izraza U(a, b) seže od 0 do $(2^N – 1) / 2^b$.

2.2. Operacije 1-komplement I 2-komplement

Razmotrimo N-bitnu binarnu riječ x kao obični binarni broj (tj. U(N, 0)). 1-komplement od x definira se kao operacija koja invertira svaki bit broja x. To možemo postići aritmetički u U(N, 0) notaciji tako da oduzmemo x od $2^N – 1$. 1-komplement broja x se označava sa x~.

2-komplement od x (oznaka x^) računa se tako da se pribroji 1 1-komplementu:

$$X^ = X~ + 1$$
$$= 2^N – X.$$

2.2.1. Signed 2-komplement fixed point racionalnih brojeva

N-bitna binarna riječ, signed 2-komplement fixed-point racionalni broj, preuzima vrijednosti iz skupa P:

$$P = \{p/2^N \mid -2^{N-1} \leq p \leq 2^{N-1} - 1, p \in Z\}.$$

Primjetimo da P sadrži 2^N elemenata. Ovaj izraz označavamo kao *A(a, b)*, gdje je a = N – b – 1. Vrijednost određenog N-bitnog binarnog x u A(a, b) notaciji dan je izrazom:

$$x = (\frac{1}{2^b})[-2^{N-1} + \sum_0^{N-2} 2^n x_n],$$

gdje x_n predstavlja n-ti bit od x. Interval vrijednost A(a, b) izraza jednak je

$$-2^{N-1-b} \leq x \leq +2^{N-1-b} - 1/2^b.$$

2.1. Osnovna pravila fixed point aritmetike

Pravila koja sljede koriste se u primjeni fixed point aritmetike. Kada se neki izraz može skalirati kao signed (A(a,b)) ili unsigned (U(a,b)) koristimo notaciju X(a,b).

- Unsigned Wordlength

Broj bitova potrebnih za ostavarenje U(a,b) je a + b.

- Signed Wordlength

Broj bitova potrebnih za ostvarenje A(a,b) je a + b +1.

- Unsigned Range

Doseg U(a,b) je $0 \leq x \leq 2^A - 2^{-B}$.

- Signed Range

Doseg A(a,b) je $-2^A \leq \alpha \leq 2^A - 2^{-B}$.

2.3.1. Dodatne operacije

Dva binarna broja moraju biti jednako skalirana kako bi se mogla zbrojiti. Npr X(c,d) + Y(e,f) je jedino valjana operacija kada je X = Y (oboje su A ili U) te ako vrijedi c = e i d = f.

- Unsigned Multiplication

$U(a_1,b_1) \times U(a_2,b_2) = U(a_1 + a_2,b_1 + b_2)$.

- Signed Multiplication

$A(a_1,b_1) \times A(a_2,b_2) = A(a_1 + a_2 +1,b_1 + b_2)$.

2.4. Posmak

Definiramo dva tipa posmaka bitova, literal i virtual (logički i virtualni) i opisujemo skaliranje rezultata za svakog od njih. Posmicanje bitova ostvareno je kao posmak za n bitova udesno. Lijevi posmak ostvaruje se kad je n < 0.

- Literal Shift

Literal (logički) posmak je ostvaren ako se pozicije bitova u registru miču ulijevo ili udesno. Logički posmak uvodimo iz dva razloga, množenje ili dijeljenje broja potencijom broja 2, ili za promjenu skaliranja. U oba slučaja mora se naznačiti mogućnost da rezultati izgubi na preciznosti ili se dogodi

preljev (overflow).

- Množenje / Dijeljenje broja potencijom broja 2

Imamo sljedeće skaliranje:

$$X(a,b) >> n = X(a,b).$$

Primjer:

Recimo da je X 16-bitni signed 2 – complement integer koji je skaliran kao A(14,1), ili Q1. Također postavimo taj integer koji je jednak 128:X += +128 = 0x0080, pa je njegova skalirana vrijednost:

$$x = X/2^1$$
$$= 128/2$$
$$= 64.0$$

Sada želimo taj broj podjeliti sa 4, pa posmičemo udesno za 2, X = X >>2, tako da nova vrijednost X je 32. Posmak nije promjenio skaliranje jer djelimo pomoću posmaka, pa je broj i dalje skaliran kao Q1 i nakon posmaka.

Primjetimo da je ovo vjerovatno loš način da dijelimo i množimo fixed point vrijednosti jer pri množenju postoji mogućnost preljeva bitova, a ako dijelimo postoji mogućnost gubitka preciznosti. Mnogo bolje rješenje je

virtualni posmak opisan u sljedećem poglavlju

Modificirano skaliranje

Logički posmak koji modificira skaliranje pomiče pozicije bitova i za posljedicu ima drugačije skaliranje izlazne i ulazne vrrijednosti. Tako imamo:

$$X(a,b) >> n = X(a + n,b - n).$$

Primjer:

Opet recimo da je X 16-bitni signed 2 – komplement integer koji je skaliran kao A(14,1) tj G1, te recimo da je taj integer jednak 128: X = +128 = 0x0080, te kao takav ima skaliranu vrijednost:

$$x = X/2^1$$
$$= 128/2$$
$$= 64.0$$

Sada naprimjer želimo promijeniti skaliranje sa Q1 na Q3 (ili ekvivalentno sa A(14,1) na A(12,3)). Posmaknemo integer 2 bita ulijevo: X = X << 2. Tako da

naš novi integer ima vrijednost 512, ali smo također primjenili skaliranje jednadžbe tako da smo dobili A(12,3) (n = -2 jer smo imali posmak ulijevo).

Tako u ovom slučaju naša konačna vrijednost je i dalje 512 / 8 = 64.

- Virtual Shift

Virtualni posmak je posmak virtualne binarne točke bez promjene vrijednost integera. Može se koristiti kao alternativna metoda za množenje i dijelenje sa potencijom broja 2. No, za razliku od logičkog posmaka, virtualnim posmakom ne gubimo preciznost i izbjegava se preljev. To se događa zato što se pozicije bitova zapravo ne miču – operacija predstavlja reinterpretaciju skaliranja.

$$X(a,b) >> n = X(a - n,b + n).$$

3. Razlika fixed point i floating point aritmetike

Tu je razliku lako uočiti na promjeru procesora digitalnih signala (DSP). DSPovi koji koriste fixed point aritmetiku pohranjuju svaki broj koristeći minimalno 16 bitova. Postoje 4 različita načina kako ovih 2^{16} = 65536 mogućih riječi mogu biti prikazane. Kao unsigned integer, spremljeni brojevi poprimaju vrijednosti od 0 do 65535. Signed integer, koji koriste 2-komplement za pohranu poprimaju vrijednosti od -32768 do 32767. Ako koristimo unsigned fraction notaciju, 65536 razina uniformno se razdjeljuju od 0 do 1. Naposljetku, signed fraction dopušta i negativne vrijednosti, pa te razine raspoređuje od -1 do 1.

S druge strane, floating point DSPovi obično koriste minimalno 32 bita za pohranu svakog broja. Kao rezultat imamo mnogo veći broj riječi, točnije 4,294,967,298. Glavna značajka pohrane brojeva u floating point aritmetici je da oni nisu uniformno raspoređeni. U najčešćem formatu (ANSI/IEEE Std. 754-1985), najveći i najmanji brojevi su $\pm 3.4\times10^{38}$ i $\pm 1.2\times10^{-38}$ redom. Svi brojevi nejednako su raspoređeni između ovih ekstrema, tako da je razlika između bilo koja dva broja otprilike 10 miliona puta manja nego vrijednost tih brojeva. Ovo je bitno zato što su velike rupe postavljene između velikih brojeva, a male rupe između malih.

Fixed point aritmetika je mnogo brža nego floating point ukoliko koristimo računala opće namjene (ukoliko je hardver visoko optimiziran za matematičke operacije razlika je bitno manja). Interna arhitektura floating point DSPova je kompliciranija. Svi registri moraju biti 32-bitni (za razliku od 16-bitnih kod fixed point), aritmetičko logička jedinica mora biti brža jer instrukcijski set je veći. No, floating point aritmetika je preciznija i ima veći dinamički doseg. Nadalje, programi imaju manji razvojni ciklus, jer programeri ne moraju brinuti o preljevu bitovi i greškama prilikom zaokruživanja.

3.1. Primjer iz svijeta tehnologije (analogni DSP uređaji)

2.1.1.1ADI Blackfin® Fixed-Point Digital Signal Processors

Analogni uređaji 12/32-bitni fixed point Blackfin procesori digitalnog signala (Slika 1.) su posebno dizajnirani da bi dosegli izvedbene zahtjeve današnjih ugrađenih audio, video i komunikacijskih aplikacija. Blackfin procesori imaju izvanrednu izvedbu i efektivnost snage sa RISC programiranim modelom, kombinirajući naprednu obradu signala sa jednostavnim za koristiti atributima koji se mogu naći u mikrokontrolerima široke primjene. Kombinacija procesnih atributa omogućuju Blackfin procesorima da imaju jednako dobru izvedbu sa obradom signala kao i sa aplikacijama za kontrolu obrade.

Slika 1. Fixed point DSP-ovi

2.1.1.2ADI SHARC® Floating-Point Digital Signal Processors

Analogni uređaji 32-bitni floating point SHARC procesori digitalnog signala (Slika 2.) bazirani su na Super Harvard arhitekturi koja balansira izvanserijsku izvedbu glavne memorije sa jako dobrim ulazno izlaznim mogućnostima. Ova Super Harvard arhitektura je proširenje originalnog koncepta odvojenog programa od memorijskih sabirnica dodavši U/I procesor sa svojim sabirnicama. SHARC procesori integriraju velike blokove memorije i odvojene dijelove memorije specifično dizajnirane za aplikaciju. [5]

Slika 2. Floating point DSP-ovi

4. Općenito o libfixmath i libfixmatrix

Libfixmath je platformno neovisna fixed point matematička biblioteka napravljena ciljano za programere koji koriste brze matematičke operacije na platofrmama koje nemaju (ili imaju spori) FPU (floating – point unit, računalni sustav specijalno dizajniran za izvršavanje matematičkih operacija u floating point aritemtici). Biblioteka pruža slično sučelje kao i standardne „math.h“ funkcije, no samo za Q16.16 fixed point brojeve. Ona ne ovisi ni o kojim vanjskim sutavima osim „stdint.h“ i kompajera koji podržava 64-bitnu aritmetiku (GCC je jedan primjer). Postoje dodatne mogućnosti kompajliranja kako bi se uklonila ovisnost o 64-bitnim kompajlerima te tako pružila mogućnost korištenja na brojnim mikrokontrolerima i DSP-ovima. Libfixmath je stvorio Ben Brewer (aka flatmush) te je prvi put objavljena kao dio Dingoo SDK projekta. Od tada se koristi kako bi se implementirala 3D grafička biblioteka FGL. [2]

Q16.16 Funkcije

fix16_acos	Arkus kosinus
fix16_asin	Arkus sinus
fix16_atan	Arkus tangens (jedan parametar)
fix16_atan2	Arkus tangens (dva parametra)
fix16_cos	Kosinus
fix16_exp	Eksponencijalna funkcija
fix16_sin	Sinus
fix16_tan	Tangens
fix16_mul	Množenje
fix16_div	Dijeljenje
fix16_sadd	Zbrajanje (overflow)

fix16_smul	Množenje (overflow)
fix16_sdiv	Dijeljenje (overflow)

Tablica 1. Libfixmath funkcije

Ostale funkcije

fix16_to_dbl	Konvertira Q16.16 u double
fix16_to_float	Konvertira Q16.16 u float
fix16_to_int	Konvertira Q16.16 u integer
fix16_from_dbl	Konvertira double u Q16.16
fix16_from_float	Konvertira float u Q16.16
fix16_from_int	Konvertira integer u Q16.16

Tablica 2. Funkcije konverzije

Procesori i performanse

Za najčešće korištenu funkciju atan2 rezultati su:

Procesor	**Vrijeme u odnosu na float point**
ARM Cortex-M0	26.30%
Marvell PXA270 (ARM) @ 312MHz	58.45%
Intell T5500	120%
Intel Atom N280	141%

Tablica 3. Procesori i brzine izvođenja

Libfixmatrix je primitivna i mala biblioteka izgrađena nad libfixmath bibliotekom za osnovne računske operacije sa matricama. Koriste se manje matrice, najčešće ispod 10x10 dimenzija. Ova biblioteka ne podržava aritmetiku sa zasićenjem (overflow), ali koristi određene zastavice za

detekciju zasićenja. Cilj biblioteke je što manji utjecaj na sustav i što veća brzina izvođenja te su zbog toga samo najosnovnije operacije implementirane. Dimenzija matrica varira od 1x1 do FIXMATRIX_MAX_SIZE (može se konfigurirati), također je moguće raditi i sa ne-kvadratnim matricama, ali uvijek se alocira memorija za najveću veličinu. Detekcija grešaka upravlja se putem zastavica u matričnim strukturama. Zbog ovoga je lako detektirati bilo kakvu grešku u bilo kojoj računskoj operaciji, bez povratne vrijednosti funkcije. [3]

Funkcije

void mf16_fill(mf16 *dest, fix16_t value);	Ispuni sva polja sa istom vrijednošću
void mf16_fill_diagonal(mf16 *dest, fix16_t value);	Ispuni dijagonalu sa danom vrijednošću, a sva ostala polja postavi na 0
void mf16_mul(mf16 *dest, const mf16 *a, const mf16 *b);	Množenje matrica
void mf16_mul_at(mf16 *dest, const mf16 *at, const mf16 *b);	Množenje transponirane matrice matricom
void mf16_mul_bt(mf16 *dest, const mf16 *a, const mf16 *bt);	Množenje matrice transponiranom matricom
void mf16_add(mf16 *dest, const mf16 *a, const mf16 *b);	Zbrajanje matrica
void mf16_sub(mf16 *dest, const mf16 *a, const mf16 *b);	Oduzimanje matrica
void mf16_transpose(mf16 *dest, const mf16 *matrix);	Transponiranje matrice

void mf16_mul_s(mf16 *dest, const mf16 *matrix, fix16_t scalar);	Množenje matrice skalarom
void mf16_qr_decomposition(mf16 *q, mf16 *r, const mf16 *matrix, int reorthogonalize);	QR dekompozicija matrice – pronađe Q i R tako da vrijedi Q*R = A, Q je ortogonalna, a R je gornje – trokutasta matrica.
void mf16_solve(mf16 *dest, const mf16 *q, const mf16 *r, const mf16 *matrix);	Rješava sustav linearnih jednadžbi Ax = b, ili ekvivalentno A\b, koristeći QR – dekompoziciju. b je matrica, a x se sprema na odredište.
void mf16_cholesky(mf16 *dest, const mf16 *matrix);	Cholesky dekompozicija simetrične matrice – pronađe L tako da vrijedi L * L’ = A, dok je L donje trokutasta. Bilo kakvi negativni brojevi priliko korjenovanja se evaluiraju na 0. Mali negativni brojevi su česta pojava zbog grešaka u zaokruživanju.

Tablica 4. Funkcije libfixmatrix

5. Libfixmath i XC8 kompajler

Prilikom uvoza *headera* i *source fileova* u MPLAB X IDE v1.70 koji koristi XC8 kompajler javlja se veći broj grešaka. Kako bi implementirali poboljšane navigacije algoritme, te su greške morale biti uklonjene. Promjene u kodu:

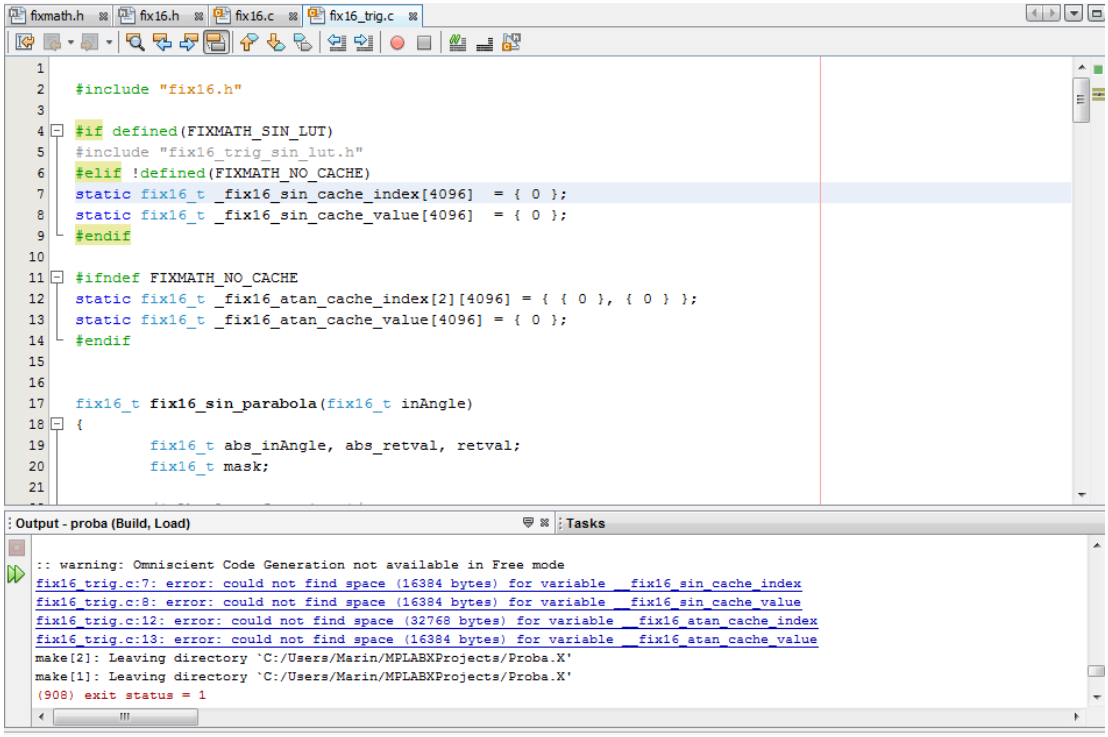
„fix16.h“

```
static const fix16_t FOUR_DIV_PI  = 0x145F3;           /*!< Fix16 value of 4/PI */
static const fix16_t _FOUR_DIV_PI2 = 0xFFFF9840;       /*!< Fix16 value of -4/PIÂ² */
static const fix16_t X4_CORRECTION_COMPONENT = 0x399A; /*!< Fix16 value of 0.225 */
static const fix16_t PI_DIV_4 = 0x0000C90F;            /*!< Fix16 value of PI/4 */
static const fix16_t THREE_PI_DIV_4 = 0x00025B2F;      /*!< Fix16 value of 3PI/4 */

static const fix16_t fix16_max = 0x7FFFFFFF; /*!< the maximum value of fix16_t */
static const fix16_t fix16_min = 0x80000000; /*!< the minimum value of fix16_t */
static const fix16_t fix16_overflow = 0x80000000; /*!< the value used to indicate overflows when FIXMATH_NO_OVERFLOW is not specified */

static const fix16_t fix16_pi  = 205887;     /*!< fix16_t value of pi */
static const fix16_t fix16_e   = 178145;     /*!< fix16_t value of e */
static const fix16_t fix16_one = 0x00010000; /*!< fix16_t value of 1 */
```

Slika 3. Fix16.h

U ovom dijelu ovoga *headera* (Slika.3) definirane su konsante koje se koriste u funkcijama nad 16-bitnim operatorima. Uglavnom se odnose na iznose čestih razlomaka broja pi te maksimalne i minimalne vrijednosti registara. Iz nepoznathih razloga, kompajler je javljao grešku kako su ove konstante definirane na dva mjesta. U nemogućnosti boljeg rješenja, imena koja su identifikatori konstanti zamjenjene su točnim vrijednostima konstanti u svim datotekama koje koriste ovaj *header* („fix16.c“, „fix16_trig.c“).

„fix16_trig.c“



Slika 4. Fix16_trig.c

Za konstane definirane kao polje od 4 bajta kompajler nije našao prostora te su uklonjene (zakomentirane) (Slika 4.).

„int64.h“

U originalnoj libfixmath biblioteci na početku ovog *headera* uključen je samo „stdint.h“, koji standardno ima definirane sve tipove podataka od 8 do 64-bitnih (signed ili unsigned). No, XC8 komajler ima samo neke od njih definirane, te je potreban dodatan *header* kako bi se mogli koristiti tipovi podataka int64_t, uint32_t itd... Osim „stdint.h“ u ovom fileu dodan je i „inttypes.h“ (Slika 5.) kako bi pokrili sve korištene varijable.

```
#ifndef INTTYPES_H
#define INTTYPES_H

#ifdef  __cplusplus
extern "C" {
#endif

typedef long long int64_t;
typedef unsigned long long uint64_t;

typedef unsigned int      uint32_t;
```

Slika 5. Inttypes.h

6. Optimizacija koda

Kako bi se kod koji je napisan u floating point aritmetici prebacio u fixed point kod potrebno je napraviti dvije stvari:

1.varijable koje se koriste u matematičkim operacijama prebaciti u fixed point varijable

2. matematičke funkcije prebaciti u fixed point matematičke funkcije

U kodu kojeg je potrebno optimizirati 1. pravilo se uglavnom odnosi na double varijable i matrice koje primaju podatke sa seznora, dok se 2. pravilo odnosi na računske operacije sa matricama, matematičke funkcije sinus, kosinus, korijen te arkus tangens, te pretvaranja varijabli priliko ispisa ili spremanja u memoriju.

Optimizirani kod (prikaz samo promjenjih dijelova, točne promjene istaknute):

```
int i, j, k;

fix16_t x_y, x_z;

fix16_t roll, pitch, yaw, yaw_c, yaw_old = 0, hx, hy, c = 0;

fix16_t temp, x[5], xyy[5], xz[5], y[5], yyy[5], z[5], zz[5], x_min = 10000, x_max = -10000, y_min = 10000, y_max = -10000, z_min = 10000, z_max = -10000, x_0, y_0, z_0, modul, mm, mmm;

fix16_t x_acc, y_acc, z_acc, x_mag, y_mag, z_mag, x_gyr, y_gyr, z_gyr, x_mag_m, y_mag_m, z_mag_m, x_gyr_k = 0, y_gyr_k = 0, z_gyr_k = 0;

char x_acc_l, x_acc_h, y_acc_l, y_acc_h, z_acc_l, z_acc_h;

char x_mag_l, x_mag_h, y_mag_l, y_mag_h, z_mag_l, z_mag_h;

char x_gyr_l, x_gyr_h, y_gyr_l, y_gyr_h, z_gyr_l, z_gyr_h;

unsigned char xor = 0, cal = 0;

unsigned char *chptr;

int zar = 0, p_c;

fix16_t Q, R, xr, xr_, xr_1, xp, xp_, xp_1, xyw, xy_1, Pr, Pr_, Pr_1, Pp, Pp_, Pp_1, yr, yp, yyw, Kr, Kp;

char lat[15], lon[15], time[15], hdop[6], N, E, T, NN, EE, hd, eep;

fix16_t poly[5] = {70142,-280163,419843,-279761,69940};

//double poly[5] = {841501,-3363000,5041500,-3360000,840000};

mf16 Qrp[2][2] = {0.00002, 0, 0, 0.00002}, Rrp[2][2] = {0.001, 0, 0, 0.001}, Hrp[2][2] = {1, 0, 0, 1}, Irp[2][2] = {1, 0, 0, 1};

mf16 Prp[2][2] = {0, 0, 0, 0}, xrp[2][1], xrp_[2][1], Prp_[2][2], yrp[2][1], Srp[2][2], Krp[2][2], Frp[2][2], EKF2[2][2], EKF1[2][1];
```

```
fix16_t Qy = 0.1, Ry, Hy = 1, Iy = 1, Py = 0, xy, xy_, Py_, yy, Sy, Ky, Fy;
fix16_t tm, h;
int It, In;
…
    putrsUSART("\r\n");
    putrsUSART(ftoa1(fix16_to_dbl(x_0)));
    putrsUSART("\r\n");
    putrsUSART(ftoa1(fix16_to_dbl(y_0)));
    putrsUSART("\r\n");
    putrsUSART(ftoa1(fix16_to_dbl(z_0)));
    putrsUSART("\r\n");
    putrsUSART(ftoa1(fix16_to_dbl(x_y)));
    putrsUSART("\r\n");
    putrsUSART(ftoa1(fix16_to_dbl(x_z)));
    putrsUSART("\r\n");
    putrsUSART(ftoa1(fix16_to_dbl(x_max)));
    putrsUSART("\r\n");
    putrsUSART(ftoa1(fix16_to_dbl(y_max)));
    putrsUSART("\r\n");
    putrsUSART(ftoa1(fix16_to_dbl(z_max)));
    putrsUSART("\r\n");
    putrsUSART(ftoa1(fix16_to_dbl(modul)));
    putrsUSART("\r\n");
…
    x_acc = fix16_from_dbl((x_acc_h << 8) + x_acc_l);
    y_acc = fix16_from_dbl((y_acc_h << 8) + y_acc_l);
    z_acc = fix16_from_dbl((z_acc_h << 8) + z_acc_l);

    x_mag = fix16_from_dbl((x_mag_h << 8) + x_mag_l);
    y_mag = fix16_from_dbl((y_mag_h << 8) + y_mag_l);
    z_mag = fix16_from_dbl((z_mag_h << 8) + z_mag_l);

    x_mag /= fix16_from_dbl(1055);
    y_mag /= fix16_from_dbl(950);
    z_mag /= fix16_from_dbl(1055);
…
roll = fix16_atan2(y_acc, z_acc); //fp_fix16_atan2
    pitch = fix16_atan2(x_acc, fix16_sqrt(y_acc * y_acc + z_acc * z_acc));
```

```
   pitch *= (-1);
   putrsUSART(ftoa1(fix16_to_dbl(roll)));
   putrsUSART(ftoa1(fix16_to_dbl(pitch)));
   hx=x_mag*fix16_cos(pitch)+y_mag*sin(pitch)*fix16_sin(roll)+z_mag*
fix16_cos(roll)*fix16_sin(pitch);
   hy = y_mag*fix16_cos(roll)-z_mag*fix16_sin(roll);
   yaw = fix16_atan2(-hy, hx);
…
          x_gyr = fix16_from_dbl((x_gyr_h << 8) + x_gyr_l);
         y_gyr = fix16_from_dbl((y_gyr_h << 8) + y_gyr_l);
         z_gyr = fix16_from_dbl((z_gyr_h << 8) + z_gyr_l);

         x_acc /= fix16_from_dbl(-16384);
         y_acc /= fix16_from_dbl(-16384);
         z_acc /= fix16_from_dbl(-16384);

         x_gyr *= fix16_from_dbl(0.00875 / 180 * M_PI);
         y_gyr *= fix16_from_dbl(0.00875 / 180 * M_PI);
         z_gyr *= fix16_from_dbl(0.00875 / 180 * M_PI);
…
xrp_[0][0] = xrp[0][0] + (x_gyr + y_gyr*fix16_sin(xrp[0][0])*tan(xrp[1][0]) +
z_gyr*fix16_cos(xrp[0][0])*tan(xrp[1][0]))*0.1;
         xrp_[1][0] = xrp[1][0] + (y_gyr*fix16_cos(xrp[0][0]) -
z_gyr*fix16_sin(xrp[0][0]))*0.1;
         xy_ = xy + (y_gyr*fix16_sin(xrp[0][0])/fix16_cos(xrp[1][0]) +
z_gyr*fix16_cos(xrp[0][0])/fix16_cos(xrp[1][0]))*0.1;

         Frp[0][0] = 1 + (tan(xrp[1][0])*(y_gyr*fix16_cos(xrp[0][0]) -
z_gyr*fix16_sin(xrp[0][0]))*0.1);
         Frp[0][1] = (y_gyr*fix16_sin(xrp[0][0]) +
z_gyr*fix16_cos(xrp[0][0]))*0.1/fix16_cos(xrp[1][0])/fix16_cos(xrp[1][0]);
         Frp[1][0] = -(y_gyr*fix16_sin(xrp[0][0]) + z_gyr*fix16_cos(xrp[0][0]))*0.1;
         Frp[1][1] = 1;

         mf16_transpose(EKF2, Frp);
         mf16_mul(Prp_, Prp, EKF2);
         mf16_mul(EKF2, Frp, Prp_);
         mf16_add(Prp_, EKF2, Qrp);

         yrp[0][0] = ((roll - xrp_[0][0]));
```

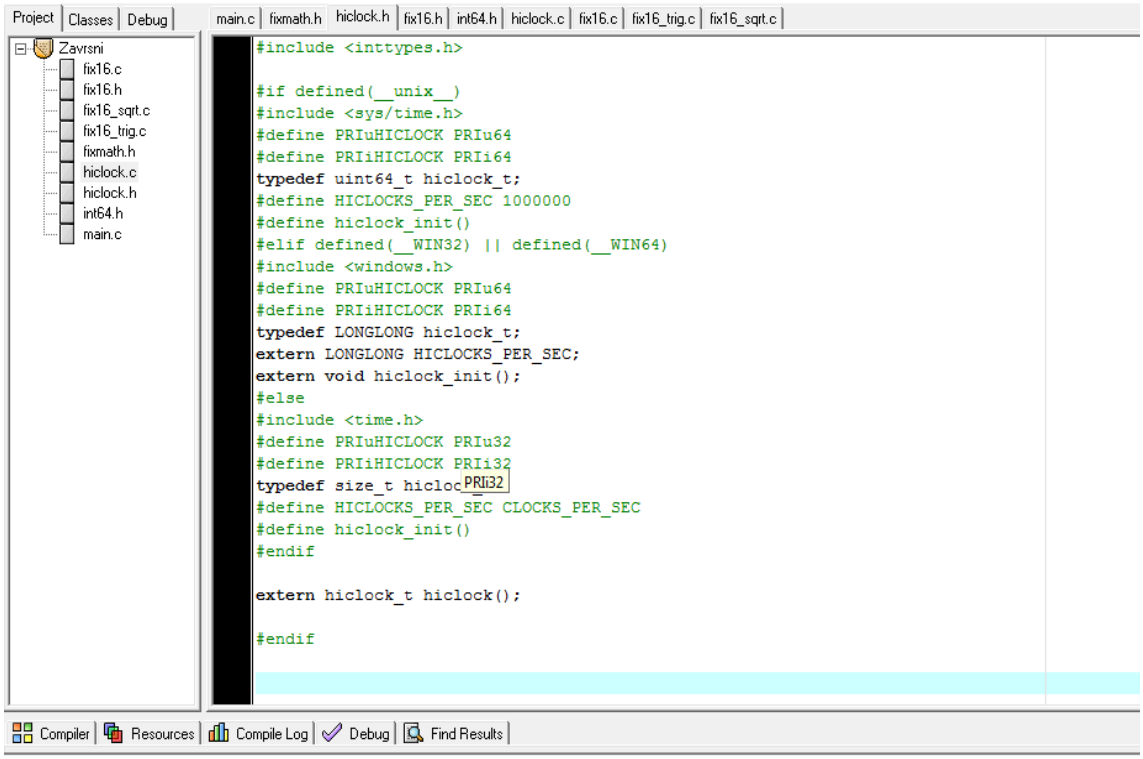
```
yrp[1][0] = ((pitch - xrp_[1][0]));

mf16_transpose(EKF2, Hrp);
mf16_mul(Srp, Prp_, EKF2);
mf16_mul(EKF2, Hrp, Srp);
mf16_add( Srp, EKF2, Rrp);

//i2x2(Srp, Krp);
mf16_mul(EKF2, Hrp, Krp);
mf16_mul(Krp, Prp_, EKF2);

mf16_mul(EKF1, Krp, yrp);
mf16_add(xrp, xrp_, EKF1);

mf16_mul(Prp, Krp, Hrp,);
mf16_sub(EKF2, Irp, Prp);
mf16_mul(Prp, EKF2, Prp_);

 hx = x_mag * fix16_cos(xrp[1][0]) + y_mag * fix16_sin(xrp[1][0]) * fix16_sin(xrp[0][0])+ z_mag * fix16_cos(xrp[0][0]) * fix16_sin (xrp[1][0]);
hy = y_mag * fix16_cos(xrp[0][0]) - z_mag * fix16_sin(xrp[0][0]);
yaw = fix16_atan2(-hy,hx);

yy = fix16_asin(fix16_sin(yaw-xy));
...
 double x_acc_temp = fix16_to_dbl(x_acc);
chptr = (unsigned char *) &x_acc_temp;
for(i=0; i<4;i++) {
   xor ^= *chptr;
   putByteUSART(*chptr++);
}
double y_acc_temp = fix16_to_dbl(y_acc);
chptr = (unsigned char *) &y_acc_temp;
for(i=0; i<4;i++) {
   xor ^= *chptr;
   putByteUSART(*chptr++);
}
double z_acc_temp = fix16_to_dbl(z_acc);
chptr = (unsigned char *) &z_acc_temp;
```

```
for(i=0; i<4;i++) {
   xor ^= *chptr;
   putByteUSART(*chptr++);
}
double x_gyr_temp = fix16_to_dbl(x_gyr);
chptr = (unsigned char *) &x_gyr_temp;
for(i=0; i<4;i++) {
   xor ^= *chptr;
   putByteUSART(*chptr++);
}
double y_gyr_temp = fix16_to_dbl(y_gyr);
chptr = (unsigned char *) &y_gyr_temp;
for(i=0; i<4;i++) {
   xor ^= *chptr;
   putByteUSART(*chptr++);
}
double z_gyr_temp = fix16_to_dbl(z_gyr);
chptr = (unsigned char *) &z_gyr_temp;
for(i=0; i<4;i++) {
   xor ^= *chptr;
   putByteUSART(*chptr++);
}
double x_mag_temp = fix16_to_dbl(x_mag);
chptr = (unsigned char *) &x_mag_temp;
for(i=0; i<4;i++) {
   xor ^= *chptr;
   putByteUSART(*chptr++);
}
double y_mag_temp = fix16_to_dbl(y_mag);
chptr = (unsigned char *) &y_mag_temp;
for(i=0; i<4;i++) {
   xor ^= *chptr;
   putByteUSART(*chptr++);
}
double z_mag_temp = fix16_to_dbl(z_mag);
chptr = (unsigned char *) &z_mag_temp;
for(i=0; i<4;i++) {
   xor ^= *chptr;
```

7. Libfixmath i GCC kompajler

Kao što je već rečeno, ukoliko koristimo računala opće namjene (osobna računala) fixed point aritmetika bi trebala biti mnogo brža u izračunu osnovnih matematičkih funkcija. Prikazat ćemo to na primjeru funkcije arkus tangens.

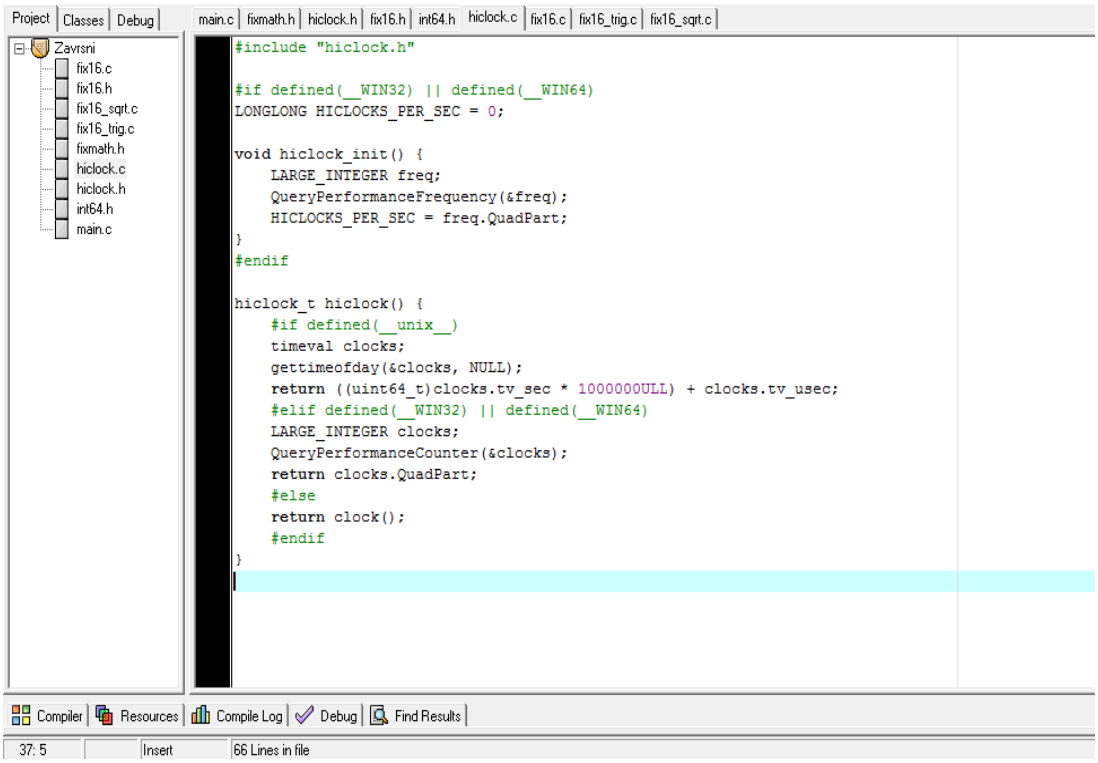
Kako bi odredili koja je funkcija brža, uspoređivat ćemo vrijeme izračuna mjereno u ciklusima procesora. Ta funckionalnost ostvarena je datotekama “hiclock.h” i “hiclock.c” . Hiclock.h (Slika 6.) predstavlja *header file* u kojemu su definirane varijable i funkicije koje koristi “hiclock.c”:



Slika 6. Hiclock.h

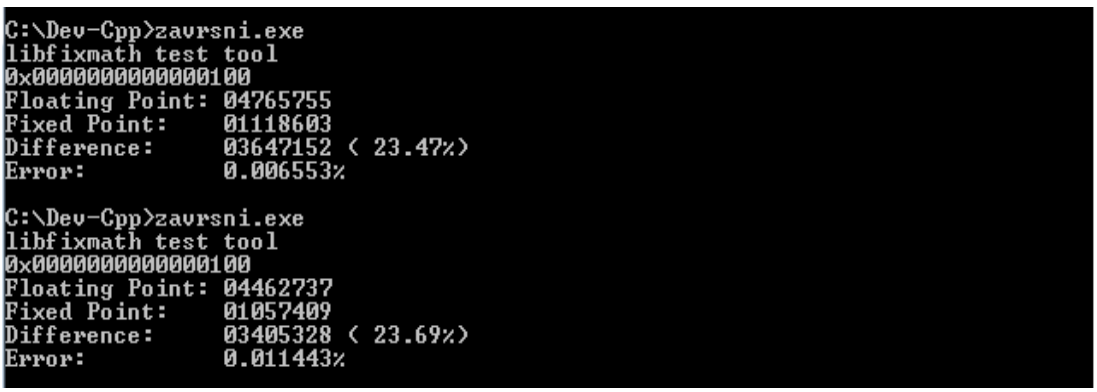
„hiclock.c“ (Slika 7.) *source file* ima implementirane funkcije koje vraćaju koliko smo iskoristili procesorski resursa. Vrijednosti koja se vraća

izražena je u *clock ticks*, a to predstavlja jedinica vremena konstante ali sistemski – specifične duljine.



Slika 7. Hiclock.c

U glavnom programu generator slučajnih brojeva bira argument za matematičke funkcije (jedna u fixed point jedna u float point aritmetici), računa vrijeme koje je potrebno svakoj u od njih da se izdračunaju (izraženo u *clock ticks*), računa razliku vremena izraženu kako postotak te odstupanje od prave vrijednosti (postotak greške). Prijmjer dva pokretanja programa (Slika 8.):



Slika 8. Rezultati testne aplikacije

Uglavnom se postotci razlike kreću između 20% – 30%, dok se postotak greške nalazi u intervalu od 0.005% – 0.015%.

8. Korištenje fixed point aritmetike

Osim u libfixmath i libfixmatrix bibliotekama, fixed point aritmetika se koristi u raznim sustavima poput:

GnuCash je aplikacija koja služi za praćenje novca (napisana u C-u). Prebacila je korištenje iz floating point u fixed point u verziji 1.6. Ova promjena je učinjena zbog veće kontrole prilikom zaokruživanja brojeva.

Tremor, Toast i MAD su biblioteke koje kodiraju Ogg Vorbis, GSM Full Rate i MP3 audio formate redom. Ovi formati koriste fixed point aritmetiku zbog velikog broja hardverskih uređaja koji nemaju FPU (dijelom zbog uštede novca, dijelom zbog očuvanja energije) te zbog ubrzanja performansi.

Svi 3D grafički strojevi na Sony-evom originalnom PlayStationu, Sega-inom Saturnu, Nintendo-vom Game Boy Advance-u (samo 2D), Nintendo DS-u i GP2X Wizu su koristili fixed point iz istog razloga kao i Tremor I Toast – dobitak kroz zaobilazak FPU arihtekture.

VisSim je visualno programiran blok dijagram jezik koji dopušta simulaciju i automatsko generiranje fixed point operacija.

Doom je posljednja first - person shooter igra od id Software-a koja je koristila 16.16 fixed point reprezentaciju za sve ne-integerske račune, uključujući sustava mapa, geometriju, kretanje lika itd. Ovo je učinjeno da bi igra bili igriva na 386 i 486SX CPU-ovima bez FPU-a.

9. Zaključak

Libfixmath i libfixmatrix biblioteke sadrže sve osnovne matematičke operacije te najosnovnije operacije za rad nad matricama. Na primjeru sa GCC kompajlerom prikazana je prava snaga fixed point aritmetike za računala opće namjene i brojeve malog raspona: dobiveno je ubrzanje od 25%. No, postoje dva problema: svaka matematička operacija mora biti implementirana, tako da za obično zbrajanje umjesto simbola “+” moramo koristiti funkciju sa 3 argumenta. Drugi problem koji se nameće je ne jednostavni prelazak programa pisanog u float point aritmetici na program pisan u fixed point aritmetici, zbog drugačije strukturiranih funkcija. Ako programiramo u fixed point aritmetici nekim problemima ćemo pristupiti na drugačiji način, matematičke izračune predefinirati da bi nam ih lakše bilo implementirati. Uz malo planiranja i strategije, još ako imamo program na mikrokontroleru sa ograničenom memorijom i mnogo korištenja matematičkih funkcija, libfixmath i libfixmatrix biblioteke se sigurno isplate.

10. Literatura

[1] http://www.microchip.com/pagehandler/en-us/family/mplabx/

[2] https://code.google.com/p/libfixmath/

[3] https://github.com/PetteriAimonen/libfixmatrix

[4] http://en.wikipedia.org/wiki/Fixed-point_arithmetic

[5] http://www.microchip.com/wwwproducts/Devices.aspx?dDocName=en547748

[6]http://www.analog.com/en/content/fixed-point_vs_floating-point_dsp/fca.html

Optimiranje koda za navigaciju na mikrokontroleru

Libfixmath i libfixmatrix su biblioteke koje koriste aritmetiku nepomičnog zareza. Da bi bile korištene na PIC18F26K22 uz XC8 kompajler morale su biti dorađene. Krajnji cilj je iskorištavanje njihovih funkcija za optimiranje navigacijskih alogoritama koji koriste osnovne matematičke funkcije nad skalarima i matricama. Rezultati na GCC kompajleru indiciraju pobljšanje performansi u prosjeku za 25% no sama implementacija u navigacijskim algoritmima je složenija jer je cijeli program napisan za float point aritmetiku.

Ključne riječi: Libfixmath, libfixmatrix, fixed point aritmetika, optimiranje navigacijskih alogoritama

Optimizing code for navigation on a microcontroller

Libfixmath and libfixmatrix are libraries which use fixed point arithmetic. For them to be used on a PIC18F26K22 using a XC8 compiler they had to be refined. The goal is to use their functions to optimize navigational algorithms that use basic mathematical functions of scalar and matrix. The results on a GCC compiler indicate an improved performance by an average of 25%, but the implementation of fixed point arithmetic on the navigational algorithms is more complex due to the program being written for float point arithmetic.

Key words: libfixmath, libfixmatrix, fixed point arithmetic, optimization of navigational algorithms

.