Fixed point simplex

v tomto adresáři máme minimální potřebné znalosti pro ovládání proměnné typu Fixed Point a pro míchání assembleru a Cčka. Pokusil jsem se vše zredukovat na minimum. Rozšířenou versi naleznete v adresáři fixed\_point

předpokládané znalosti: DSP engine

O fixed point si přečteme na straně 149 datasheetu k překladači MPLAB\_XC16\_C\_Compiler\_Users\_Guide.pdf adresář je něco jako C:\Program\_Files(x86)\Microchip\xc16\v1.36\docs

Podívejte se do toho datasheetu, nejde to kopírovat

V tabulce 8.3 máme všechny možné typy. Vidíme, že je to nachystáno na práci s DSP engine.

Dále máme od strany 168 – kapitola 9 – popsán Fixed-point aritmetic support. Račte si přečíst.

dále si pustíme program fixed01.c , slouží pro základní orientaci v problematice. Poznámka v datasheetu říká: -menable-fixed must be specified.

Vyřešil jsem to tak, že jsem to prostě dopsal

Cvaknout pravým tlačítkem na název projektu – dole Properties – XC16 – XC16 Global Option – dopsat do řádku aditional options

Při Build si pak zkontrolujte, že se vám to objeví ve volání gcc :

 "C:\Program Files (x86)\Microchip\xc16\v1.36\bin\xc16-gcc.exe" ../Vrh2018/diskP/mikroproc/33EV32/programovani/progr\_a\_vysv/fixed\_point/fixed01.c -o build/default/production/\_ext/1713490250/fixed01.o -c -mcpu=33EV32GM002 -MMD -MF "build/default/production/\_ext/1713490250/fixed01.o.d" -mno-eds-warn -g -omf=elf -DXPRJ\_default=default -legacy-libc  **-menable-fixed** -O0 -msmart-io=1 -Wall -msfr-warn=off

Vzorový file je fixed01.c

Tento program slouží k základní orientaci pro práci s typem fixed\_point , více od něj nečekejte. Prohlédněte si, jak jsou zobrazena čísla fixed – hexadecimálně.

Na první řádek kódu si dáme breakpoint, mačkáme F7, díváme se do variables. Čísla si zobrazíme také jako Binary, abychom viděli, co se doopravdy děje.

Neumí to míchat integer a Fract, proto je při násobení (-1) přetypování ( číslo -1 je samozřejmě integer ):

b = ( signed \_Fract )(-1) \* b ;

neumí to úplně dobře přetypovávat, hroutí se to u záporných čísel

V adresáři C:\Program Files\Microchip\xc16\v2.10\docs nebo nějakém podobném máte soubor MPLAB\_XC16\_C\_Compiler\_Users\_Guide.pdf , zde je kapitola

**11.6 Mixing C and Assembly Language Code**

**15.7 Function Call Conventions**

When calling a function:

• Registers W0-W7 are caller saved. The calling function must preserve these values before the function call if

their value is required subsequently from the function call. The stack is a good place to preserve these values.

• Registers W8-W14 are callee saved. The function being called must save any of these registers it will modify.

• Registers W0-W4 are used for function return values.

• Registers W0-W7 are used for argument transmission.

• DSRPAG/PSVPAG should be preserved if the -mconst-in-code (auto\_psv) memory model is being used.

Dále si přeložíme oba programy v adresáři ./engine\_simplex

Zde máme spolupráci asemblerovského programuc Cčkovým. Do source file addeneme oba zdrojové file a přeložíme. Pokusíme se udělat úvod k tomu, jak by se asi mohlo pracovat s DSP engine. potřebujeme k tomu znát látku v

<http://ozeas.sdb.cz/panska/mikroproc/33EV32/programovani/asm_progr_a_vysv/asm_a_C/>

Soubor podprog.s slouží pouze pro nadefinování podprogramů, které se budou volat v programu hlavni.c . Pokud chceme definovat nějaké buňky, musíme k tomu použít sekci .bss . Překladač pak skombinuje tyto buňky s tím, co jsme nadefinovali v cčkovém file. Pokud použijete konkrétní adresu ( konkrétní číslo ) pro vaše proměnné, hrozí havarie, neboť překladač v části Cčkového file netuší, co jste udělali.

Zároveň se pokusíme zajistit, aby nám Cčková část programu „nehrabala“ do W registrů. To uděláme tak, že budeme volat podprogramy v assemblerovské části. Podprogram se provede jako celek, takže nehrozí, že by s tím Cčková část něco provedla . **Určitě ale při provádění assemblerovského programu zakažte přerušení,** jinak hrozí, že zavolané přerušení změní obsah registrů ve vašem programu. Můj vzorový program přerušení nepoužívá, nepovolil jsem ho.

Na začátku programu je definováno několik globálních polí. Pokud chcete, aby proměnné byly vidět v assemblerovské části, musí být v Cčku globální.

const signed \_Fract - říká, že pole je v paměti EEPROM, bude se s ním pracovat pomocí PSV. !!!! Pozor, je tady opravdu const, nikoli static !!!!! Z hlediska DSP engine je PSV považováno za X-space paměti.

signed \_Fract \_\_attribute\_\_(( space(ymemory) )) jinepole[100] ; říká, že jinepole bude uloženo v Y-space paměti , uvidíme to na adresách v simulátoru.

Dáme si breakpoint na první instrukci d=0.25; pustíme debugger, ten se kousne na této instrukce. Dále mačkáme F7, abychom vlezli do podprogramu.

První podprogram je první(). V assemblerovské části je to

.global \_prvni

\_prvni:

Podtržítko je zásadně důležité, Cčkový program díky němu uvidí tento assemblerovský program. Dále je důležitá direktiva .global, ta zařídí, že název bude vidět i mimo assemblerovský file. V Cčkové části je

void prvni ( void ) ;

tím říkáme Cčku, jakého typu má být podprogram.

Mačkáme F7 a vlezli jsme do assembleru. ( F8 udělá funkci jako jednu instrukci, F7 leze dovnitř funkce)

mov #\_polefract, W0

ukládá do registru W0 adresu začátku pole polefract. Opět je zásadně důležité podtržítko, assembler vidí to, co je nadefinováno v Cčku. Zde musí být proměnná jako globální. Pole polefract je nadefinováno v Cčkové části.

Do variables si dáme polefract a polePSV , rozklikneme, abychom viděli všechny prvky. Za repeat je příklad přesunu dat z jednoho pole do druhého. Bude se to hodit například jako posouvátko u FIR, tam samozřejmě přesouváme data z nějakého pole do téhož pole.

Nezapomeneme, že číslo u repeat je o 1 menší než celkový počet opakování.

Dále máme v hlavním Cčkovém programu ukládání čísel do proměnných. Překladači musíme říct, jakého typu je číslo, které jsme napsali . Viz Table 8-5 a následující v souboru MPLAB\_XC16\_C\_Compiler\_Users\_Guide.pdf.

Typ čísla je udán sufixem, u nás:

r – fracitonal

ur - unsigned fractional

Funkce druha() ukazuje, jak se dá v assemblerovské části použít proměnná z Cčkové části. Prostě před ni uděláme podtržítko. Proměnná musí být deklarována jako globální. A dále nezapomeňte, že registry W mají 16 bitů, takže do nich můžete uložit jenom šestnáctibitovou proměnnou.

Funkce treti() předvádí předávání parametrů. To, co je v závorce jako první parametr, je uloženo do registru W0. To, co je jako druhý, je uloženo do W1. A tak dále ….. . Funkce vrací hodnotu. Vrátí tu hodnotu, která byla v registru W0, pouze a jedině W0, nejde to přepnout na žádný jiný registr. A zase to znamená, že vrací 16 bitů, tedy to může být int , fract, případně char – použije se jen polovina registru W0 . Typ vrácené hodnoty nadefinujete v Cčkové části jako deklaraci funkce, zde

int treti ( int a , int b ) ;

Mačkáme F7 a lezeme do podprogramu. Je tam součet, ale můžeme tam mít cokoli jiného.

Dále v Cčkové části nastavujeme DSP engine. Zapínáme saturaci u akumulátorů a saturaci při zápisu z akumulátoru do registru ( sac ) – SATDW, supersaturaci nebudeme používat. ( podrobněji viz file ../pdf/assembler.pdf , počínaje kapitolou 4.11 )

Funkce ctvrta provádí úplně obyčejný součet. Používá aritmetiku se saturací. V simulátoru je trochu problém, že neumí zobrazit číslo jako fractional. Nezbývá, než se na výsledek podívat jako na hex.

 a = ctvrta( 0.1r , 0.2r );

 a = ctvrta ( 0.25r , 0.25r );

 a = ctvrta ( 0.99r , 0.8r ) ;

 a = ctvrta ( -1.0r , -1.0r ) ;

 a = ctvrta ( -0.8r , -0.7r ) ;

Dívejte se a kochejte se tím, jak to krásně saturuje. Pokud nejste schopni převést hexadecimální hodnotu čísel na float, tak máte bohužel smůlu. Pro kontrolu připomínám, že 0x8000 znamená -1.0 , 0x7fff znamená +0.9999 a kousek .

K tomu, aby instrukce sac uložila správnou saturovanou hodnotu akumulátoru je nutné, abyste nastavili CORCONbits.SATDW = 1 ;

Dále už se blížíme k funkci paty . Funkce nám přiblíží mac prefetches. Proto nejdříve ukládáme několik hodnot do pole jinepole. Toto pole je uloženo v Y-space peměti RAM, to se dělá direktivou

signed \_Fract \_\_attribute\_\_(( space(ymemory) )) jinepole[100] ;

Funkce pata je už dost návodná, ale samozřejmě jsem vám nechtěl napsat úplně všechno.

Zapneme si také

Window – target memory views – Program memory

a můžeme sledovat instrukce v assembleru, tak, jak je překladač přeložil. Funguje to i v Cčkové části.

mov # tabulka, W8

je známé z loňska. Tabulka hodnot je na konci file, račte se podívat. Můžete použít buď tuto věc, nebo si tabulku nadeklarovat jako pole const v Cčkové části , je to jedno. Pak samozřejmě použijete něco jako mov # \_nejakepole , W8 , tedy podtržítko kvůli proměnné v Cčkové části a # , protože to \_nejakepole je adresa buňky, tedy krásné číslo.

clr A nuluje akumulátor a také umí mac prefetches.

Tady ukládáme první hodnotu v poli tabulka do W5 a první hodnotu v poli jinepole do W4

V Cčkové notaci W4 = jinepole[0] ;

instrukce mac je ta správná pro výpočet filtru FIR. Musíte ji samozřejmě použít vícekrát, takže repeat. (repeat tam není, račte si doplnit !!!!!) msc je tam jen pro ukázku dalších možností.

Počínaje

sac A,#4, W0

je už funkce udělána tak, abyste její výsledek mohli přímo poslat do DA převodníku pomocí SPI. Ostatně proto je v Cčkové čásli deklarována jako unsigned int .

Čísla fract, která vybleje algoritmuf FIR, zabírají všech 16 bitů. Vy ale potřebujete do DA převodníku posílat 12 bitů. Proto je u sac posuv #4, kterým se číslo z akumulátoru šoupne o 4 bity doprava.

Dále toto číslo and-neme s 0x0fff , čímž zajistíme, že horní 4 bity jsou opravdu 0.

Dále negujeme 11. bit. Tím děláme z posunutého fractional krásné číslo unsigned int v rozsahu 0-4095. Negaci bitu provádíme pomocí XOR-nutí s číslem 0x0800 . Pokud netušíte, o čem mluvím, pak vizte, že vše jsme dělali loni v části <http://ozeas.sdb.cz/panska/mikroproc/33EV32/programovani/asm_progr_a_vysv/logicke_instr/> - račte si doplnit.

K pochopení uvedené operace je nutno znát chování převodníku, viz <http://ozeas.sdb.cz/panska/mikroproc/33EV32/programovani/progr_a_vysv/ADC/ADC_06.doc> , strana 13

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| napětí na výstupu DA | číslo dec na vstupu DA | odpovídající fractional | fractional hex | fract hex o 4 bity posunutý |
| 0V | 0x0000 | -1.0  | 0x8000 | 0x0800 |
| polovina rozsahu | 0x0800 |  0.0 | 0x0000 | 0x0000 |
| maximum rozsahu | 0x0fff |  0,999755859375  | 0x7fff | 0x07ff |
|  |  |  |  |  |

Připomínám, že rozsah napětí máme posunutý, číslo 0 odpovídá napětí 2,5V ( doufám, že to není problém, rozsah převodníku je 0 – 5V , referenční napětí je napětí zdroje, opět si v případě potřeby doplňte vzdělání). Tohle je samozřejmě situace u vstupního AD převodníku. Na výstupu je MCP4822 , ten má referenční napětí 4.095 nebo 2.047 , podle toho, jak jste si ho naprogramovali řídícím slovem, takže tam je situace obdobná s trochu jinými hodnotami napětí.

Nejvíce záporné číslo je -1.0 , to je hex 0x8000 . To odpovídá napětí 0V, tedy do převodníku DA budeme posílat číslo 0. Číslo jsme posunuli o 4 bity, takže 0x0800 . A z toho udělám 0 prostě tak, že znegujeme 11. bit.

Polovině rozsahu, tedy napětí 2.048V, odpovídá číslo 0. Musíme z něj tedy pro DA převodník dostat 0x0800 , takže opět negace 11. bitu. A maximu rozsahu, tedy 4.095V, odpovídá číslo fractional 0,999755859375 , hexadecimálně a po posunutí o 4 bity 0x07ff . A z toho máme udělat úplné maximum, tedy 0x0fff . Opět negujeme 11. bit .

Do funkce paty si dopište maskování, které musíte provést, abyste správně nastavili převodník MCP4822 – horní 4 bity.

Dále v úvodu funkce a v jejím závěru vidíme push a pop s registry W8 a W10 . V návodu - MPLAB\_XC16\_C\_Compiler\_Users\_Guide.pdf - se v části charper 16 – Mixing C and Assembly code – píše, ... functions ... must preserve registers W8 – W15 . To děláme pomocí push a pop. Pokud si vymyslíte nějaké jiné registry, nezapomeňte na to .

No a funkce pro výpočet filtru FIR je tedy úplně jednoduchá.

Deklarace bude něco jako

unsigned int nasefunkce ( \_Fract vstup ) ;

po jejím zavolání se hodnota proměnné vstup nacpe do registru W0

Dále máme nachystáno pole vzorků. To si nejprve posuneme, něco jako mov [W1++] , [W2++] – samozřejmě to napíšete správně, takhle je to špatně !

Na začátek pole vzorků pak uložíme hodnotu z registru W0

Provedeme výpočet vzorku - pomocí mac. nezapomeneme na repeat

Výsledné číslo upravíme , funkce vrací hodnotu, která je v registru W0 .

Ono by se to mohlo volat nějak takhle:

SPI2BUF = nasefunkce ( ADC1BUF0 ) ;

Hezké, že ?

Je to hezké, ale není to ono. Tohle totiž bude trvat příliš dlouho, protože to nejdříve přečte ADCBUF, potom spočte FIR, potom vyšle hodnotu do SPI2BUF. Potřebujeme, aby tyto věci běžely paralelně. Takže nějak takhle:

pomocADC = ADC1BUF0 ;

// odstartovat převod převodníku - a už se převádí

SPI2BUF = pomocna ; // a tady už to vzsila na SPI hodnotu vzorku

pomocna = nasefunkce( pomocADC ) ; // tady počítáme FIR, může to trvat dost dlouho

Poslední věc je přerušení od SPI. To nám vrací CS obvodu MCP4822 zpět do 1. To má větší prioritu, takže se bude volat během výpočtu filtru FIR. Aby nám nezničilo naše registry, doporučuji ho napsat v assembleru - jsou to dvě instrukce, nastavení pinu RB. a nulování požadavkového bitu, obojí se dá v assembleru udělat tak, abychom nesáhli na žádný registr. Je docela možné, že v Cčku to bude také tak nějak podobně, podívejte se do výpisu ProgramMemory, simulátor převádí hexadecimální instrukce na assemblerovské instrukce, takže je to docela čitelné. Samozřejmě, když píšu, že „ …..To má větší prioritu“ , tak to znamená, že tu prioritu nastavíte, „vono samo“ se to fakt neudělá.

A teď už nic nebrání tomu, abyste se vrhli na svůj krásný filtr FIR.

Dále jsou v programu funkce sedma a osma. Je to tatáž funce , v asm vidíme, že voláme tentýž úsek programu. Nedělá nic zvláštního, funkce jsou určeny pro demonstraci chyby při volání. Funkce sedma má jako parametr fract, funkce osma má jako rarametr unsigned int. Ještě jednou zdůrazňuji, že osma a sedma je totéž, podívejte se do asm. Rozdíl je pouze v tom, jak s touto funkcí zachází Cčko. Demonstrace je na konci programu, dejte si tam breakpointz, mačkejte F7 a dívejte se na registr W0. Ten má sloužit pro uložení hodnoty, kterou předáváme funkci. V obou případech je jako parametr celecislo = 0x4000; U funkce sedma se přenos do registru W0 nepovede, podívejte se do variables. Povede se pouze u osma, kde je jako parametr funkce číslo signed integer. Ve funkci sedma tedy neproběhne konverse integer na fixed, přesněji se při té konversi číslo zmrší.

Úplně na konci před while máte přiřazení

a = celecislo ;

kde je vidět, že to takhle nejde.

Píšu o tom proto, že své podprogramy budete používat na filtr FIR. Ten dostává vzorky z AD převodníku, úplně přesně z registru ADC1BUF0 . No ale ten je definován jako unsigned int !

Definici máte v adresáři C:\Program Files\Microchip\xc16\v2.10\support\dsPIC33E\h nebo tak nějak podobně, file p33EV32GM002.h

#define ADC1BUF0 ADC1BUF0

extern volatile uint16\_t ADC1BUF0 \_\_attribute\_\_((\_\_sfr\_\_));

Jediný správný postup tedy je „dopravit“ hodnotu z ADC1BUF0 až dovnitř podprogamu jako unsigned int, uvnitř podprogramu ji převezme registr W0 , a následně s touto hodnotou pracujeme v DSP engie. A ten s tím číslem pracuje tak, jak mu říkají bity IF a US v registru CORCON.

Dále máte v adresáři FIR nějaký pokus o filtr FIR, ale je dost nedodělaný