Kruhový buffer

Cyklický buffer je další ohromující možnost DSP engie. Umožní ještě dále zkrátit výpočty, ve kterých se používají zpožďovátka.

Vše je popsáno v datashetu procesoru, strana 76, kap 4.5 Modulo Addressing.Funkce

Při inkrementování registrů u mac prefetches často potřebujeme, aby se hodnota registru po překročení nějaké hranice vrátila na začátek. A to je právě ono ! Začátek nacpeme do registru XMODSRT ( pokud pracujeme s X-space ), konec do registru XMODEND ( opět u X-space ). V registru MODCON nastavíme registr W, který má být takto adresován ( MODCON je v datasheetu na straně 42, tab4-1. Pořádný výklad k tomu chybí. Bity 15 a 14 zapínají modulo addressing pro X-space nebo Y-space, bity 7-4 definují registr W pro Y space, pokud zadáte 0xf , je to disabled. Obdobně bity 4-0 definují W registr pro X space. Bity 11-8 definují bit reverse mode register 0xf disabled

Do MPLABX si dáme oba file, tedy cykpod.s a cykbuf.s

Na začátek cykbuf.s si dáme breakpoint , mačkáme F7 a díváme se, co to dělá.

Nadefinovali jsme dvě pole , pole a jinepole . Pole je v X space, jinepole je Y space. Do polí jsme uložili nějaké hodnoty. Jsou samozřejmě nesmyslné, ale jsou voleny tak, abychom se podle nich snadno orientovali v poli. Takže v pole[0] je číslo 100 , v pole[1] je číslo 101 , …. V pole[5] je číslo 105 Podobně jsme v main uložili do jinepole čísla 20 až 24 , jednotky souhlasí s indexem pole.

Dále nastavujeme kruhový buffer

XMODSRT = pole ; začátek kruhového bufferu, registr XMODSRT. Do něj ukládáme počáteční adresu pole. Pamatujete si doufám, že název pole je pointer, a ten obsahuje adresu v paměti, samozřejmě adresu začátku pole.

XMODEND = &pole[5] ; do registru XMODEND ukládáme adresu konce pole. Pokud netušíte, co je & , tak vám bohužel nemohu pomoci a musíte si tuto problematiku dostudovat. Do Variables si dáme uvedené registry, taktéž si tam dáme pole , a díváme se na adresu začátku a konce pole a na čísla, která se nám uložila do registrů. X na začátku označení registrů znamená X space. V definici pole je const int pole[6] , takže se pole nachází v EEPROM a budeme do něj přistupovat pomocí PSV. Proto mají adresy pole na 15. Bitu 1 . Račte si hexadecimální adresy převést do binárních čísel a je to jasné. Nebo to prostě z toho čísla vidíte, to je ta lepší varianta.

Dále následují instrukce

YMODSRT = jinepole ; // zacatek pole zacatek buffru pto modulo adresovani Y space

YMODEND = &jinepole[4] ; // konec pole konec bufferu pro modulo adresovani Y space

To je obdoba předchozích, tentokrát pro Y space .

Následuje zapnutí modulo addressing , instrukce jsou dostatečně komentované .

MODCONbits.XMODEN = 1 ; // modulo addr pro X space enabled

MODCONbits.YMODEN = 1 ; // modulo addr pro Y space enabled

MODCONbits.XWM = 8 ; // cislo registru pro X space W8 nebo W9 a zadne jine

MODCONbits.YWM = 10 ; // cislo registru pro Y space W10 nebo W11 a zadne jine

MODCONbits.BWM = 15 ; // Bit-Reversed Addressing mode is disabled 15 je 1111

Dále voláme podprogram uvod

Mačkáme F7, to znamená, že „vlezeme“ do podprogramu.

Přepínáme na sadu 1 , aby nám Cčko nezměnilo obsah našich registrů.

MOV # \_pole , W8 ; pocatecni adresa pole do prislusneho registru

MOV # \_jinepole , W10

Do registrů pro X space a Y space jsme nastavili počáteční adresy polí.

movsac A, [W8] +=2, W5 , [W10] +=2, W4

Funkce nedělá nic. Má ale MAC prefetches, které nacpou do registrů W5 a W4 čísla, které jsou na nultých indexech polí. Registry si dejte do Variables a podívejte se. V desítkovém zobrazení by mělo být krásně vidět, jaká tam jsou čísla, a ta čísla odpovídají indexu pole.

CTXTSWP #0 ; prepiname zpet na sadu 0 a odcházíme zpět do Cčkové části .

Dále vstupujeme do cyklu while. V něm dokolečka voláme funkci modulo01.

Centrální instrukcí této funkce je mac W4\*W5 , A, [W8] +=2, W5 , [W10] +=2, W4 . Tady nepřemýšlejte nad tím, co je výsledkem násobení, ale dívejte se na to, co dělají registry W8 a W10 . Registry „dolezou“ na hodnoty , které jsou v registrech YMODEND a XMODEND , a potom se jejich obsah změní na hodnotu, která je v registrech XMODSRT a YMODSRT . Mačkejte F7, dívejte se do Variables. Jinak to nepochopíte. Opravdu je nutno, abyste si to takhle udělali, pouze koukáním na Kubalíka se to opravdu nenaučíte.

Obě pole jsou záměrně jinak dlouhé, to znamená, že tady máme dva kruhové buffery s různou délkou. Ještě jednou opakuji, že celý tento program je napsán proto, abyste se naučili, jak kruhové buffery fungují.

Zkusil jsem udělat nějaký výstup z funkce modulo01 (push W0 – popW0) , výsledek se ukládá do proměnné i. To, co ukládá instrukce sac, je hodnota na 31. – 16. Bitu akumulátoru. Proto jsme to posunul doleva, abychom v proměnné i měli nějakou použitelnou hodnotu. Ale to je jenom taková hračka, celý smysl tohoto programu je pochopit, jak funguje kruhový buffer. Vy budete při návrhu filtru FIR samozřejmě pracovat s čísly signed fractional, a ta se ukládají do 31.-16. Bitu akumulátoru, bity 15 – 0 slouží pro zanedbatelně malé výsledku násobení, takže lac a sac budou fungovat bez problémů.

Ještě jedno upozornění: cyklický buffer funguje pouze na mac prefetches. Na obyčejnou inkrementaci nebo dekrementaci, tedy na instrukce INC W1,W1 DEC2 W3,W3 NEFUNGUJE !!! Pokud tedy potřebujete posunout ukazovátka, aniž byste dělali nějaký výpočet, použijte nám známou instrukci movsac . Ta samozřejmě také něco uloží do registrů, ale to následně přepíšete tím, co potřebujete. Musíte dát jenom pozor, abyste „nevylezli“ z rozsahu adres pro nadefinovaná pale, to by procesor udělal address error.

A nyní se pokusíme vysvětlit použití kruhového bufferu pro výpočet filtru FIR. Pokud nevíte, co to je a jak se počítá, jděte na gymnázium.

Máme dvě pole, x a b , v souladu s běžným značením. V poli x jsou vzorky, v poli b jsou koeficienty. Pole x vidíme na následujícím obrázku, pole b je analogické.



Výstupní vzorek vypočteme pomocí vztahu

Vystup = x[0]\*b[0] + x[1]\*b[1] + x[2]\*b[2] + x[3]\*b[3] + x[4]\*b[4] + x[5]\*b[5] + x[6]+b[6]

Nyní přijde další vstupní vzorek x . Uložíme ho na pozici x[0] . Předtím ale musíme celé pole posunout. To lze udělat například takto:

MOV #konecpole , W5

MOV #(konecpole-2) , W6

Repeat #delkapole

MOV [W6--] [W5--]

Je to jednoduché, ale trvá to dlouho. Proto si zavedeme kruhový buffer. Jak jistě víte, program se píše odprostředka. Myslíme si tedy, že už jsme někde uprostřed a neustále přicházejí další a další vzorky. Zavedeme si ještě další označení pomocí písmene T. T znamená „teď“ . T-1 znamená „předtím“ . T-2 znamená před předtím atd. atd.



Na obrázku je stejné pole pro uložení vstupních vzorků x . Vzorek, která přešel právě teď, je uložen v poli x[2] . Vzorek, která přišel předtím, je v poli x[3] . Vzorek ještě starší je v poli x[4]. Vzorek úplně nejstarší je v poli x[1] . Ještě jednou připomínám, že stáří vzorku je označeno pomocí písmene T .

Výstupní vzorek vypočteme úplně stejně jako předtím.

Vystup = x[T]\*b[0] + x[T-1]\*b[1] + x[T-2]\*b[2] + x[T-3]\*b[3] + x[T-4]\*b[4] + x[T-5]\*b[5] + x[T-6]+b[6]

Je to opravdu stejné, jenom jsme použil jiné označení. A vzorek „teď“ je na pozici x[2] , není na začátku, ale to přece vůbec nevadí. Musíme jenom vynásobit ta dvě správná čísla.

No a přijde další vstupní vzorek. Umístíme ho na pozici x[T-6], to samozřejmě znamená, že tenhle vzorek mizí. Ale tak to u filtru FIR chodí, ten nejstarší vzorek vypadne z posouvací fronty .



V obrázku nahoře máme vzorek, který přišel právě teď ( x[T] ) . Uložili jsme ho do buněk x[1]. Máme to vyznačeno v červené řádce. Všechny ostatní vzorky samozřejmě zestárly (červená řádka) , ale zůstaly v paměti na stejných pozicích, jako byly předtím. Jediná změna je na pozici x[1]. Výsledek vypočítáme úplně stejně jako předtím, tedy

Vystup = x[T]\*b[0] + x[T-1]\*b[1] + x[T-2]\*b[2] + x[T-3]\*b[3] + x[T-4]\*b[4] + x[T-5]\*b[5] + x[T-6]+b[6]

Ale protože vzorky zestárly, jsou již na jiných pozicích v poli x . To znamená, že při násobení musíme začít od jiné skutečné adresy v poli x ( například vzorek x[T-3] byl v minulém čase - černá řádka -na pozici x[5] , v přítomném čase – červeně- je na pozici x[4] , ale je to dáno jenom tím, že „zestárnul“, čísla zůstala na stejných místech v poli , samozřejmě kromě vzorku „Teď“ )

No a můžeme pokračovat dále. Opět přišel další vzorek, dáváme ho na pozici x[0] .



A dále přišel další vzorek – opět červený řádek. Uložili jsme ho na pozici x[6] . Tady vidíme krásnou funkci cyklického bufferu , protože na „menší index než je x[0]“ již nic uložit nejde, takže ukazovátka adres přeskočila na konec pole.



No a ještě jednou, opět přišel vzorek, stav „teď“ je opět červeně , a takhle je to pořád dokola.



Až se to budete učit, vezmete si velký papír, na něj si nakreslíte pole **x**  a pole **b** , a budete si malovat čárami, co s čím se násobí. Pak postoupíte o jednu periodu vzorkování k dalšímu vzorku, vezmete si jinou barvu, nakreslíte si přišedší vzorek do pole, a zase namalujete, co s čím se násobí. Pak přijde další vzorek, vezmete další barvu, a tak dále a tak dále.

No a potom podle toho uděláte program, stěžejní instrukce je mac, musíte ji několikrát repeat-nout , a potom pomocí „plonkové mac prefetch“ posunout ukazovátka tak, aby ukazovala na správnou adresu v poli. Funkci doporučuji udělat tak, abyste do ní zadali to, co vám vyleze z převodníku (ADCBUF je unsigned int ! i když je ten převodník v módu „signed fractional“) . Funkce vrací hodnotu – nejlépe si výsledek už uvnitř funkce předělejte na int nebo unsigned int, abyste to pak mohli poslat do SPI .

Chápu, že je tady nutno myslet. Ale to je právě to, co se od vás právem očekává. Jsme ve třetím ročníku, už opravdu nestačí se naučit nazpaměť tabulku funkce AND, aniž bychom věděli, k čemu to je. A pokud vám to nevyhovuje, přejděte na gymnasium !