

Technická zpráva



Akademie věd České republiky
Ústav teorie informace a automatizace AV ČR, v.v.i.

Viterbi koprocesor obvodu TMS320C6416

Ing. Jan Kloub, Ing. Antonín Heřmánek PhD.
{kloub, hermanek}@utia.cas.cz, +420-2-6605 2511

Obsah

1	Úvod	1
2	Konvoluční kodér	1
3	Dekódování konvolučního kódu	2
3.1	Viterbiho algoritmus	2
4	Koprocesor pro podporu výpočtu Viterbiho algoritmu	2
5	Řadič EDMA	5
6	Knihovna podporující funkce DSP v jazyce C	6
6.1	EDMA	6
6.2	VCP	8
6.3	RTDX	9
7	Základní práce s RTDX v prostředí MATLAB	10
8	Ověření funkce	11
8.1	Postup ověření funkce	12
9	Výsledky	13
10	Výpis obsahu CD-ROM	14

Revize

Revize	Datum	Autor	Popis změn v dokumentu
0	19.12.2007	Kloub	Vytvoření dokumentu
1			
2			
3			
4			

1 Úvod

Tento dokument popisuje implementaci Viterbiho dekodéru pomocí signálového procesoru a jeho ko-procesoru pro podporu Viterbiho algoritmu. Viterbi algoritmus dekóduje konvoluční kód, který může být zatížen chybou při přenosu dat. Příklad využití ko-procesoru je implementován pomocí nástroje "Code Composer Studio". Zdrojové kódy jsou napsány v jazyce C a assembleru daného procesoru. Pro ověření funkce a předávání dat je využit nástroj MATLAB, který komunikuje s cílovou platformou přes rozhraní JTAG pomocí takzvaného "Real-Time Data Exchange" (RTDX).

2 Konvoluční kodér

Při přenosu dat může dojít chybám, které se v praxi často shlukují. Pro dostatečné zabezpečení přenosu dat lze použít zabezpečovací kódy. Určitou skupinu z nich jsou kódy konvoluční.

Konvoluční kódování dat lze velmi snadno realizovat obvodovým řešením.

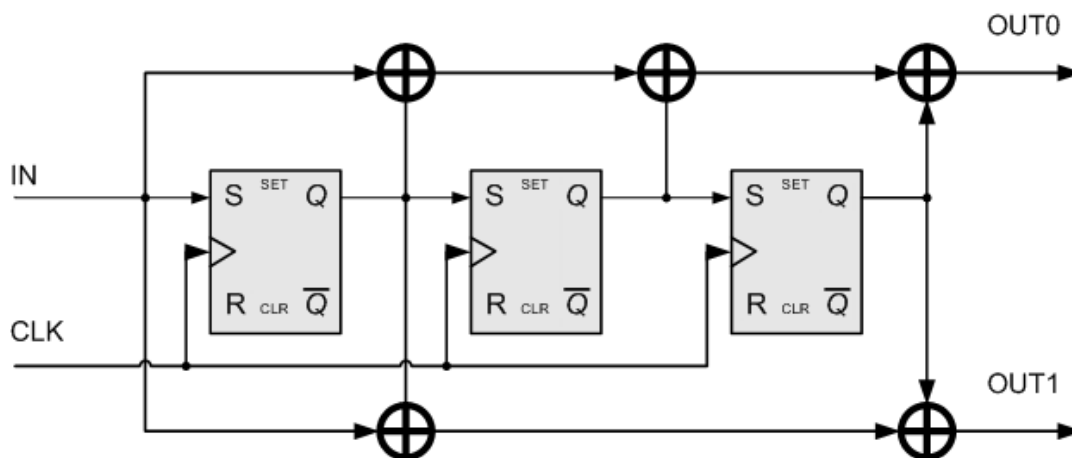
Konvoluční kodér je charakterizován několika základními parametry:

- R = informační poměr $R = k/n$, kde k je počet informačních bitů generující n výstupních bitů
- K = počet možných vazeb z posuvného registru do exkluzivních součtů
- $G_0(x) \dots G_{N-1}(x)$ = polynomy generující výstup

V následujícím příkladu konvolučního kodéru bude popsána jeho obvodová realizace. Mějme následující parametry (popis způsobu volby a vhodnosti parametrů není součástí tohoto textu):

$$\begin{aligned} R &= 1/2 \\ K &= 4 \\ G_0(x) &= x^4 + x^3 + x^2 + x \\ G_1(x) &= x^4 + x^3 + x \end{aligned} \quad (1)$$

Základem kodéru je posuvný registr, který uchovává část z historie příchozích dat. Z této historie a z příchozích dat je odvozen výstup dekodéru podle generujících polynomů. Na obrázku 1 je znázorněno konkrétní zapojení pro zadané parametry, kde registry obsahují historii dat (stav kodéru). Výstupy jsou odvozeny pomocí součtu modulo-2 z hodnot určených polynomy. Potřebná datová redundance pro zabezpečení přenosu dat je odvozena z počtu výstupů kodéru, tedy počtem generujících polynomů.



Obrázek 1: Příklad obvodové realizace konvolučního kodéru ($K=4$, $R=1/2$, $G_0=(1111)$, $G_1=(1101)$)

Výstupy kodéru bývají serializovány do bitové posloupnosti. Pro zvýšení informačního poměru se v posloupnosti vynechávají některé bity podle stanoveného schématu (např. cyklicky pomocí pevně stanovené masky). Těto technice se říká děrování - puncturing (popis metody děrování není součástí tohoto textu).

3 Dekódování konvolučního kódu

Jak bylo popsáno v kapitole 2, může být datový přenos zabezpečen proti chybám konvolučním kódem. Pokud uvažujeme, že při přenosu došlo k chybě a nebo je použito takzvané děrování, musíme odhadovat nejpravděpodobnější zakódovanou bitovou posloupnost.

Jedním z algoritmů pro určení nejpravděpodobnější bitové posloupnosti je Viterbiho algoritmus. V dalším textu bude popsána implementace dekodéru pomocí koprocessoru signálového procesoru TMS320C6416.

3.1 Viterbiho algoritmus

Viterbiho algoritmus se opírá o grafovou teorii hledání nejlépe ohodnocené cesty. Graf je tvořen všemi možnými přechody mezi jednotlivými stavy historie konvolučního kodéru, tvořící mřížku (označováno také jako trellis). Vzhledem k tomu, že podstatou obvodové realizace konvolučního kodéru je posuvný registr, lze velmi snadno určit strukturu přechodů v grafu. Na obrázku 2 je znázorněno jakým způsobem přechody v grafu vznikají (stavy jsou pro názornost označeny binárními hodnotami).

Graf začíná expandovat od počátečního stavu (stav 000). V každém dalším kroku expandují právě dvě hrany z dosažených stavů. Jedna hrana vede do stavu 1XX a druhá do stavu 0XX, kde XX jsou posunuté bity hodnoty stavu, z kterého hrany vychází (všimněme si, že nejvyšší bit následujícího stavu odpovídá zakódovanému bitu). Takto se tvoří kombinace cest, končící vždy v jednom ze stavů mřížky.

Pro výběr cesty, která odpovídá přijatým datům, musíme zavést pro jednotlivé hrany metriky. Ohodnotíme jednotlivé hrany hodnotou, která odpovídá hodnotě výstupu konvolučního kodéru ve stejném stavu jako je stav mřížky. Pro jeden stav kodéru existují dvě možné hodnoty výstupu v závislosti na vstupu kodéru. Pro hranu vedoucí do stavu s nejvyšším bitem rovným nule odpovídá ohodnocení pro vstup roven nule a obdobně i pro vstup rovný jedné.

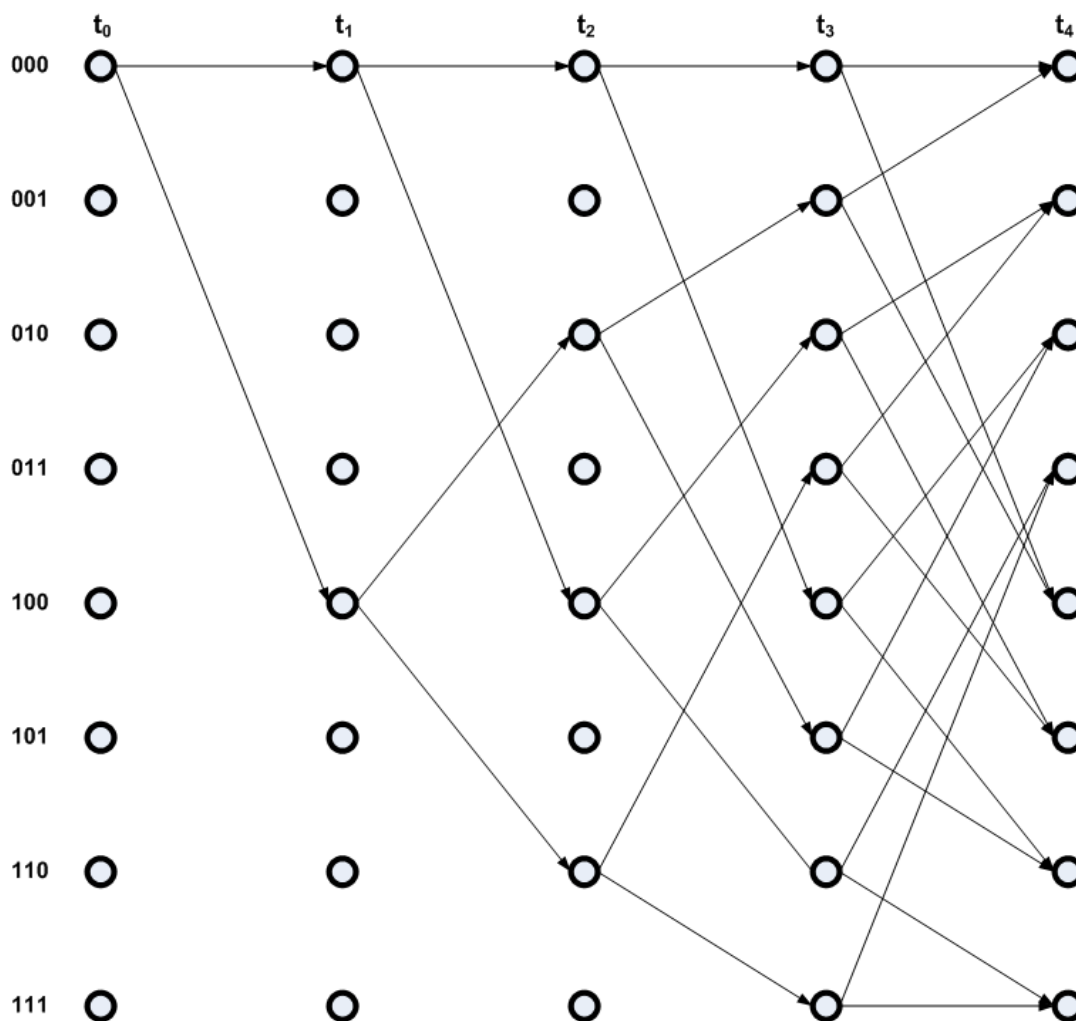
V každém kroku vytváření grafu porovnáme ohodnocení hrany se vstupním symbolem. Metrika pro každou hranu je vypočítána jako kódová vzdálenost mezi ohodnocením hrany a vstupním symbolem v daném čase (např. Hamingova vzdálenost). Každá vytvořená cesta je ohodnocena součtem jednotlivých hranových metrik. Do jednoho stavu mřížky vedou maximálně dvě cesty. Cesta s horším ohodnocením může být "zapomenuta". Jsou-li ohodnocené cesty totožné, vybere se deterministicky jedna z nich (např. vždy ta cesta, která do stavu vede ze stavu s nižší hodnotou). V každém stavu je uchována hodnota ohodnocení vítězné cesty do stavu vedoucí (označována jako akumulovaná metrika).

Vlastní dekodování dat spočívá ve zpětném průchodu grafem po nejlépe ohodnocené cestě. Teoreticky se graf může tvořit v čase do nekonečna. V praxi je graf omezen na časové okénko dané délky. Při zpětném průchodu grafem určuje aktuální stav přímo výstupní hodnotu. Jak bylo popsáno výše, výstupu odpovídá nejvyšší bit dosaženého stavu. Pořadí bitů výstupu je dekodováno v opačném pořadí, než byla data dekodována a pořadí musí být prohozeno.

4 Koprocessor pro podporu výpočtu Viterbiho algoritmu

Signálový procesor TMS320C6416 (DSP) je vybaven koprocessorem pro Viterbiho dekodování (VCP), který byl navržen pro bezdrátové standardy IS2000 a 3GPP.

Vlastnosti koprocessoru:

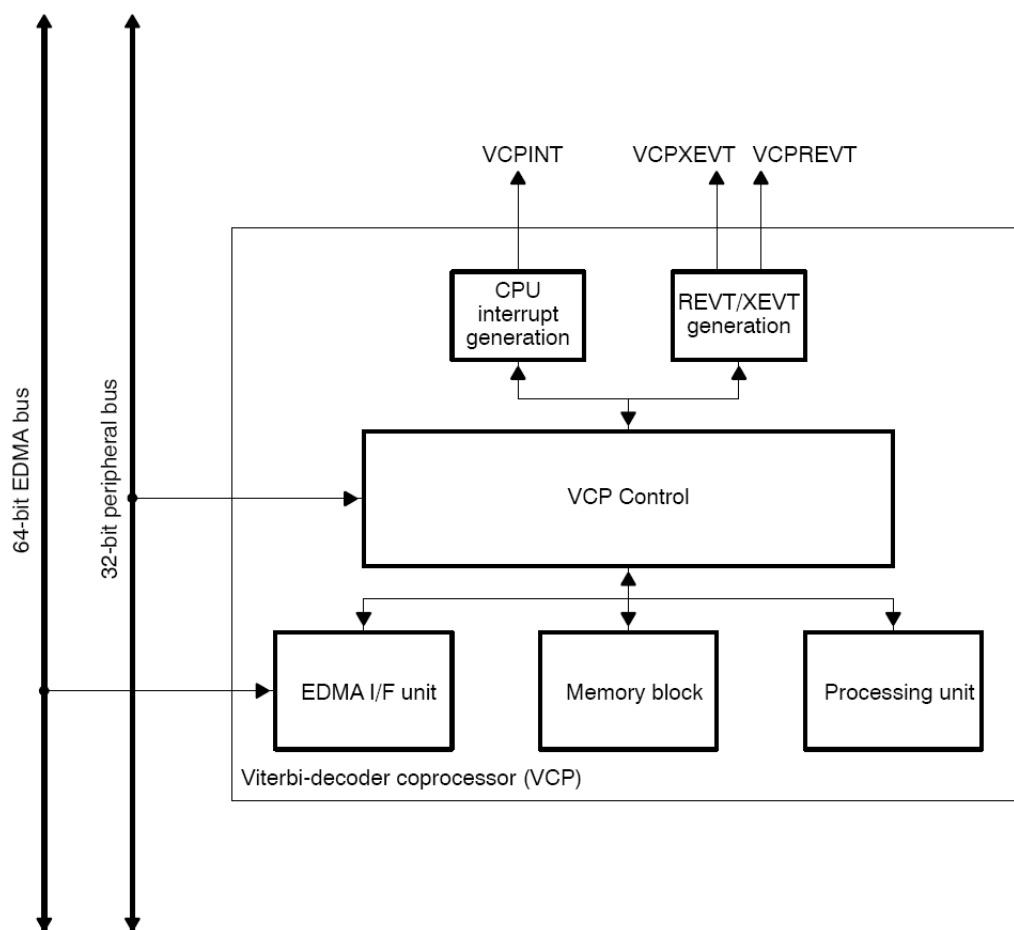


Obrázek 2: Expanze hran grafu mřížky v čase (pro $K=4$)

- Podpora dekódování pro $K = 5, 6, 7, 8$ nebo 9 .
- Uživatelsky zadávané koeficienty polynomů ¹
- Informační poměry $1/2, 1/3$ nebo $1/4$.
- Možnosti nastavení zpětného průchodu při dekódování.
- Výpočet metrika a zpracování tzv. děrovaného kódu je prováděna na straně DSP
- Koprocesor obsahuje vlastní optimalizovanou paměť.
- Komunikace mezi DSP a VCP je zajištěna pomocí EDMA řadiče.

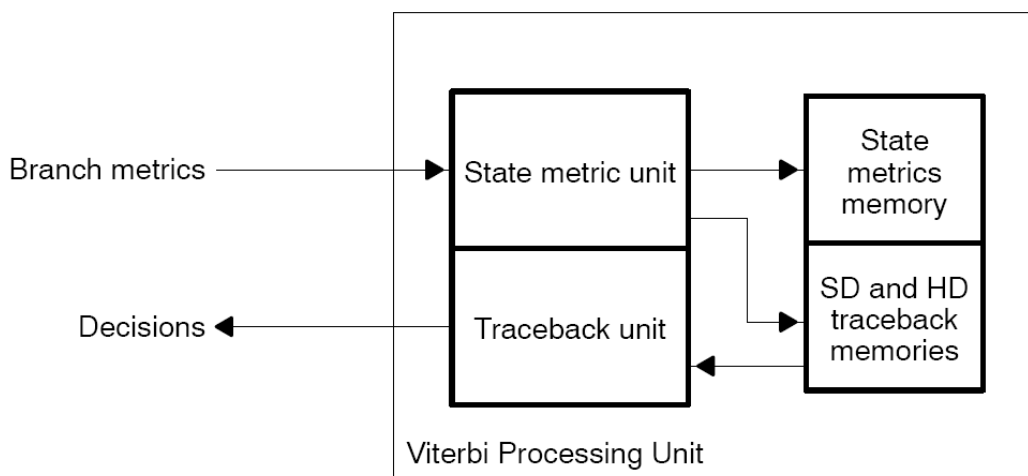
Na obrázku 3 je znázorněno blokové schéma koprocesoru. Koprocesor komunikuje s DSP pomocí 32-bitové sběrnice a s řadičem EDMA pomocí 64-bitové sběrnice.

¹Polynomy jsou reprezentovány binární hodnotou uloženou v konfiguračních registrech koprocesoru. Z registrů určujících polynomy je uvažováno pouze $K-1$ nejvyšších bitů. K -tý bit polynomu je považován vždy za 1. Binární hodnoty určující polynom je nutné příslušným způsobem posunout doleva.



Obrázek 3: Blokové schéma koprocessoru pro Viterbiho algoritmus (převzato z [1])

Zahájení přenosu dat je na základě signálu VCPXEVT a VCPPREVT, kde VCPXEVT indikuje žádost o vstupní data a VCPPREVT indikuje žádost o vyčtení výstupních dat. Pro přenos dat mezi paměťovým prostorem a dekodérem je nutné inicializovat struktury v parametrické paměti řadiče EDMA. Způsob práce s řadičem EDMA a datové struktury budou popsány dále v textu. Na obrázku 4 je znázorněna architektura koprocessoru.

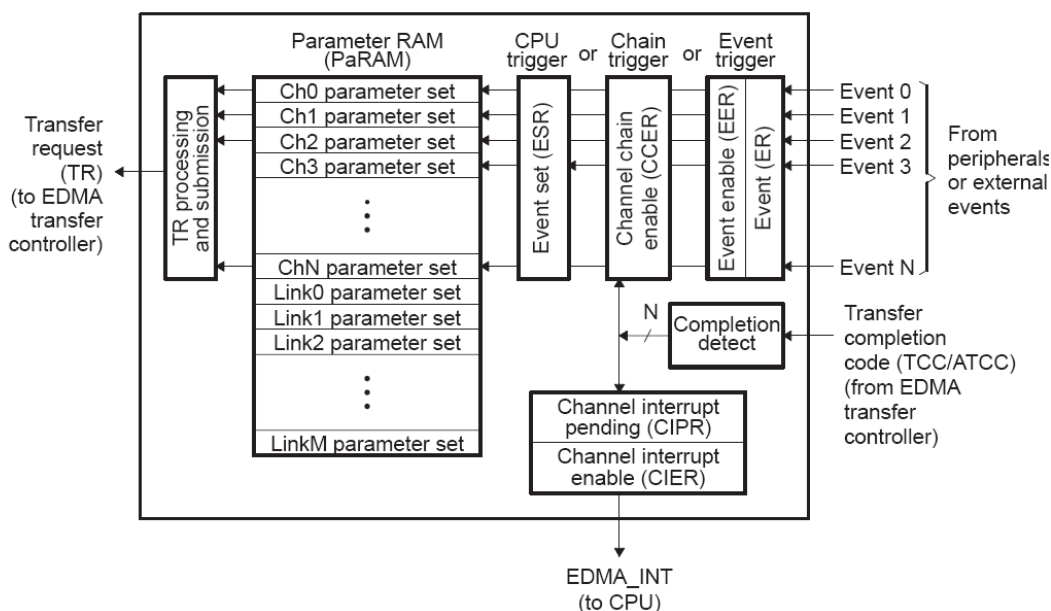


Obrázek 4: Architektura VCP (převzato z [1])

Konfigurační registry koprocessoru jsou mapovány do adresního prostoru procesoru. Význam obsahu jednotlivých registrů je popsán v [1]. V našem případě je obsah registrů vygenerován pomocí funkcí a maker implementovaných v knihovně CSL (Chip Support Library). Popis těchto funkcí a maker je uveden v [2]. Příklad použití knihovny pro práci s koprocessorem bude popsán dále v textu.

5 Řadič EDMA

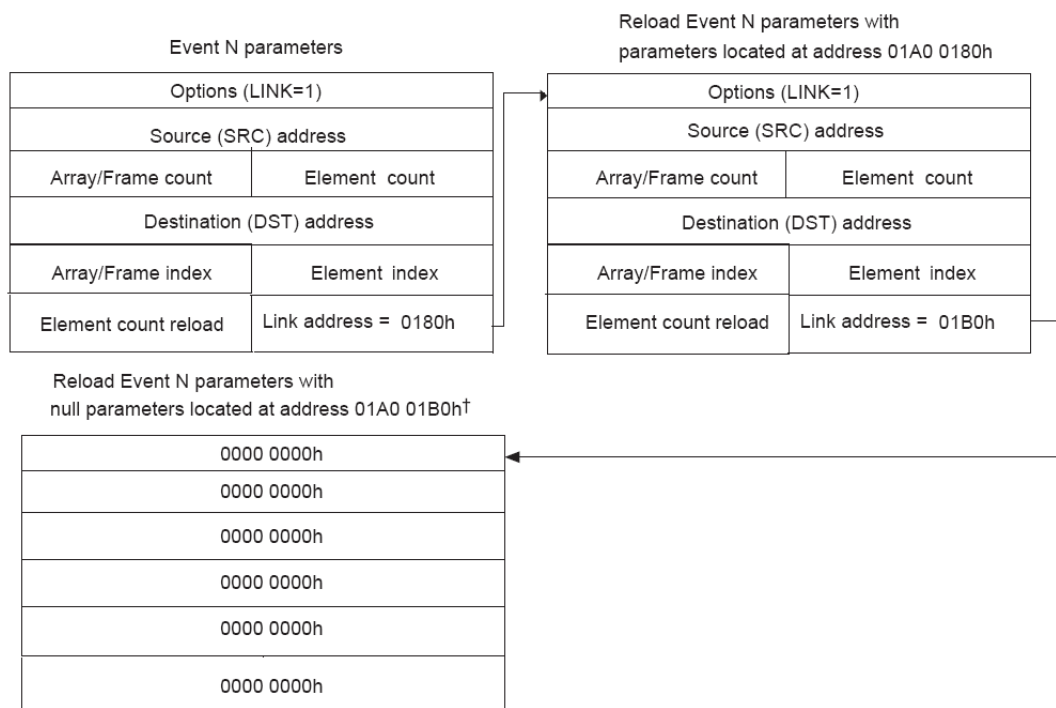
Jak bylo zmíněno v předchozím textu, je pro přenos dat mezi koprocessorem a pamětí adresního prostoru realizován pomocí řadiče EDMA. Přenos pomocí EDMA bývá zpravidla spouštěn pomocí signálů generovaných periferiemi. Parametry přenosu (zdroj, cíl, velikost datového bloku, šířka slov, atd.) je uložena v parametrické paměti řadiče. Blokové schéma řadiče je znázorněno na obrázku 5.



Obrázek 5: Blokové schéma řadiče EDMA (převzato z [3])

Pro každý signál je vyhrazena určitá část paměti. Zbývající prostor paměti je využit pro záznamy k

takzvaným zřetěženým přenosům a pro odkládací paměť. Každý záznam lze zřetěžit s jiným, tak že je do záznamu zapsána adresa parametrů, které se mají použít při další události vyvolané signálem stejným signálem. Takto lze velmi snadno vytvořit stavový automat přenosů, jehož přechody jsou vyvolány jedním signálem. Zřetěžení záznamů paměti je ukázáno na obrázku 6.



Obrázek 6: Zřetězování záznamů v parametrické paměti řadiče EDMA (převzato z [3])

Podrobnější informace o nastavení řadiče a parametrech lze nalézt v [3] a [4].

6 Knihovna podporující funkce DSP v jazyce C

Softwarové knihovny umožňují snadné využití prostředků procesoru DSP a jeho periférií v jazyce C. V následujícím textu budou stručně popsány funkce, makra a datové typy, umožňující pracovat s VCP koprocесorem, EDMA řadičem a RTDX.

6.1 EDMA

Pro využití funkcí pro EDMA v jazyce C je nutné pomocí direktivy `#include` zavést knihovnu `cs1_edma.h`. Následující fragment zdrojového kódu ukazuje využití knihovny a deklaraci struktur pro manipulaci s jednotlivými kanály řadiče EDMA.

```
#include <cs1_edma.h>
```

```
EDMA_Handle hEdmaVcpDec,      /* EDMA channel used for Hard Decisions */
             hEdmaVcpIc,       /* EDMA channel used for IC Values */
             hEdmaVcpBrMet;    /* EDMA channel used for Branch Metrics */
/* data */
```

Řadič EDMA může generovat přerušení (například po dokončení nějakého přenosu) a následující kód ukazuje funkce pro inicializaci řadiče, povolení přerušení, alokaci struktur pro jednotlivé kanály


```

/* Completely reset the EDMA */
EDMA_resetAll();

/* Enable EDMA -> CPU interrupt */
IRQ_enable(IRQ_EVT_EDMAINT);

/* Allocate transfer completion code */
vcpTcc = EDMA_intAlloc(-1);

...

/* Enable the TCC to generate a CPU interrupt */
EDMA_intEnable(vcpTcc);

...

IRQ_disable(IRQ_EVT_EDMAINT);          /* Disable EDMA -> CPU interrupt */

```

Alokaci záznamů v parametrické paměti lze provést funkcí `EDMA_open()`, `EDMA_allocTableEx()` a případně `EDMA_allocTable()`. Funkce `EDMA_open()` alokuje část paměti, která je přímo určená typem signálu (např. `VCPREVT`). Pro alokaci paměti pro záznamy, které se využijí pro zřetězení přenosu jsou alokovány funkcemi `EDMA_allocTableEx()` a `EDMA_allocTable()`. Tímto vytvoříme přenosový kanál.

Nastavení parametrů přenosu lze provést funkcí `EDMA_config()` a pro zřetězení záznamů funkce `EDMA_link()`.

```

/* Open handles to the EDMA channels */
hEdmaVcpDec = EDMA_open(EDMA_CHA_VCPREVT, EDMA_OPEN_RESET);

EDMA_allocTableEx(1, &hEdmaVcpBrMet);

...

/* Configure channel parameters for IC registers */
EDMA_config(hEdmaVcpIc, &edmaConfig);
EDMA_link(hEdmaVcpIc, hEdmaVcpBrMet);

```

Vytvořené kanály můžeme povolovat a zakazovat funkcemi `EDMA_enableChannel()` a `EDMA_disableChannel()`. Uzavření kanálu lze provést funkcí `EDMA_close()`. Alokované struktury v parametrické paměti uvolníme pomocí funkce `EDMA_freeTable()`.

```

/* Enable the transmit and receive channels */
EDMA_enableChannel(hEdmaVcpDec);
EDMA_enableChannel(hEdmaVcpIc);

...

EDMA_close(hEdmaVcpDec);          /* Close the hard decisions EDMA channel */
EDMA_close(hEdmaVcpIc);          /* Close the IC values EDMA channel */
EDMA_freeTable(hEdmaVcpBrMet);    /* Close the branch metrics EDMA channel */
EDMA_freeTable(hEdmaVcpOutPar);   /* Close the output parameters EDMA channel */
EDMA_freeTable(hEdmaVcpNull);     /* Close the NULL EDMA channel */

```

Ve funkci pro obsluhu přerušení můžeme využít funkce pro test přerušení a potvrzení přerušení pomocí funkcí `EDMA_intTest()`, `EDMA_intClear()` a `EDMA_intFree()`.

```
if (EDMA_intTest(vcpTcc)){
    EDMA_intClear(vcpTcc);          /* Clear the interrupt in the CIPR    */
    EDMA_intFree(vcpTcc);          /* Free the TCC value to the system */
    vcpTcc = -1;                   /* Reset flag                      */
    ...
}
```

Funkce pro obsluhu přerušení musí být uvedena v sekci *.vectors* pod přerušením číslo 8. Například pro funkci `edmaIsr()` následovně:

```
int8 push      b0
    mvkl      .s2    _edmaIsr,b0
    mvkh      .s2    _edmaIsr,b0
    b         .s2    b0
    pop       b0, 4
    nop
    nop
```

Bližší informace o jednotlivých funkcích a datových typech lze nalézt v [2].

6.2 VCP

Před vlastním dekódováním je nutné koprocesor nakonfigurovat pro daný konvoluční kód a formát zpracovávaného datového rámce. Pro vygenerování parametrů lze použít funkci `VCP_genParams()` (podpořenou v knihovně CSL), které jsou uloženy v datové struktuře typu `VCP_Params`. Pomocí funkce `VCP_genIc()` lze tuto strukturu převést na reprezentaci odpovídající přímo obsahu konfiguračních registrů. Obsah registrů je zpravidla nahrán také pomocí přenosu EDMA. Po zavolání funkce `VCP_start()` je generován první signál `VCPXEVT`, pomocí něhož je spuštěn přenos parametrů do registrů koprocesoru. Pomocí zřetězení EDMA přenosů bude následující signál `VCPXEVT` představovat žádost o vstupní data (indikace prázdného vstupního bufferu). Stejným způsobem budou vyčítány výstupní data na základě signálu `VCPREVT`. Následující fragment kódu ukazuje použití jednotlivých funkcí podpořených knihovnou CSL.

```
#include <cs1_vcp.h>

...

/* Fill out VCP parameters */
VCP_genParams(&vcpParameters[0], &VcpConfigParms);

/* Calculate the VCP IC values */
VCP_genIc(&VcpConfigParms, &VcpConfigIc);

/* Start the VCP to begin the EDMA transfers */
VCP_start();
```

Pro dekódování příchozích dat je nutné předpočítat hranové metriky, pro každé vstupní kódové slovo, reprezentované jako 7-bitová čísla se znaménkem. Začátek dat s hranovými metrikami musí být v paměti zarovnaný na dvojité slovo (4 byty). Uspořádání a reprezentace jednotlivých hodnot metrik v paměti je uveden v [1]. Paměť pro výstupní data musí být stejným způsobem zarovnána a musí obsahovat

sudý počet 32-bitových slov. Pro informační poměr $1/n$ je třeba ke každému vstupnímu kódovému slovu vypočítat 2^{n-1} hranových metrik. Vypočtené metriky jsou poté předány pomocí řadiče EDMA.

Uveďme zde příklad výpočtu hodnot metrik pro informační poměr roven $1/2$ a takzvané "hard" dekódování. Pokud se jedná o "hard" dekódování, musíme vstupní symboly převést podle BPSK modulace ($0 \rightarrow 1$, $1 \rightarrow -1$).

Pro informační poměr $1/2$ je nutné pro každé vstupní kódové slovo vypočítat dvě hranové metriky:

$$BM_0(t) = r_0(t) + r_1(t) \quad (2)$$

$$BM_1(t) = r_0(t) - r_1(t) \quad (3)$$

Symbol $r_0(t)$ odpovídá hodnotě symbolu vztahující se k polynomu G_0 v čase t a $r_1(t)$ k polynomu G_1 v čase t .

Codeword	r0	r1	BM0	BM1
00	1	1	2	0
01	1	-1	0	2
10	-1	1	0	-2
11	-1	-1	-2	0

Tabulka 1: Způsob přepočtu hranových metrik

6.3 RTDX

Pro využití RTDX je nutné inicializovat jednotlivé kanály pomocí maker `RTDX_CreateInputChannel` a `RTDX_CreateOutputChannel`. Dále je třeba povolit příslušné přerušení pomocí makra `TARGET_INITIALIZE()`. Toto makro je definované v hlavičkovém souboru `target.h`, který lze nalézt v adresáři kde je nainstalováno "Code Composer Studio" (`<ccsinstall>\examples\<tgt-device>\shared\`). V sekci `.vectors` je nutné přiřadit k příslušným přerušením obslužné rutiny. Vzorový soubor `intvecs.asm` lze najít také v adresáři `<ccsinstall>\examples\<tgt-device>\shared\`. Dále je nutné ve skriptu pro překladač přidat proměnou `RTDX_interrupt_mask` s její inicializací a přidat několik paměťových sekcí jako `.rtdx_text`, `.rtdx_data` a `.pinit`. Vzorový skript lze nalézt ve zmíněném adresáři. Ve vlastním programu lze pak povolit či zakázat jednotlivé kanály pomocí funkcí `RTDX_enableInput()` a `RTDX_disableOutput()`. Pro čtení dat z hostitelského systému lze číst dat pomocí funkce `RTDX_read()` a zapisovat pomocí funkce `RTDX_write()`. Následující fragment kódu ukazuje použití RTDX.

```
#include <rtdx.h>                /* RTDX                                */
#include "target.h"              /* TARGET_INITIALIZE()                */

/* Declare and initialize an:
 * input channel called "ichan"
 * output channel called "results"
 */
RTDX_CreateInputChannel(ichan);
RTDX_CreateOutputChannel(results);

...

/* Target Specific Initialization                                */
```

```

TARGET_INITIALIZE();

/* Enable the channels                                     */
RTDX_enableInput(&ichan);
RTDX_enableOutput(&results);

...

ausrcvd = RTDX_read( &ichan, &data.recvd, sizeof(data.recvd));

...

RTDX_write( &results, &decoded[0], 2*sizeof(decoded[0]) );

...

/* Disable the channels                                   */
RTDX_disableInput(&ichan);
RTDX_disableOutput(&results);

```

7 Základní práce s RTDX v prostředí MATLAB

V prostředí MATLABu lze vytvořit spojení s nástrojem Code Composer Studio (CCS). Spojení je vytvořeno pomocí konstruktoru `ccsdsp`, který instanciuje objekt pro komunikaci s CCS. Pomocí metod nad vytvořeným objektem je možné ovládat CCS a včetně komponenty pro rozhraní RTDX. V následující tabulce jsou uvedeny příklady pro základní práci s objektem pro komunikaci s CCS, použité ve skriptu pro ověření funkce koprocessoru VCP procesoru DSP.

Příklad kódu	Význam
cc = ccstdsp cc.restart	Vytvoření instance objektu <code>cc</code> pro komunikaci s CCS. Metoda <code>restart</code> nastaví čítač instrukcí na začátek aktuálního programu.
cc.run	Metoda spustí aktuální program DSP.
cc.rtdx.set('timeout', 20)	Nastavení časového intervalu (20 s) pro čekání na data.
cc.rtdx.configure(1024,4)	Nastavení počtu vyrovnávacích pamětí a jejich velikosti.
cc.rtdx.open('ichan','w')	Otevření kanálu <code>ichan</code> pro zápis.
cc.rtdx.open('results','r')	Otevření kanálu <code>results</code> pro čtení.
cc.rtdx.close('ichan')	Zavření kanálu <code>ichan</code> .
cc.rtdx.close('results')	Zavření kanálu <code>results</code> .
cc.rtdx.enable('ichan')	Povolení kanálu <code>ichan</code> .
cc.rtdx.enable('results')	Povolení kanálu <code>results</code> .
cc.rtdx.disable('ichan')	Zakázání kanálu <code>ichan</code> .
cc.rtdx.disable('results')	Zakázání kanálu <code>results</code> .
cc.rtdx.flush('results','all')	Uvolnění všech zpráv z komunikačního kanálu <code>results</code> .
cc.rtdx.flush('results', 1)	Uvolnění jedné zprávy z komunikačního kanálu <code>results</code> .
cc.rtdx.writemsg('ichan', indata)	Zápis dat (<code>indata</code>) do komunikačního kanálu <code>ichan</code> .
cc.rtdx.readmsg('results', 'int32')	Čtení dat z komunikačního kanálu <code>results</code> (datového typu <code>int32</code>).
cc.rtdx.msgcount('results')	Zjištění počtu zpráv čekajících v komunikačním kanále <code>results</code> .

Tabulka 2: Metody pro ovládání CCS a modulu pro RTDX.

8 Ověření funkce

Pro ověření funkce koprocessoru pro Viterbiho dekódování použijeme prostředí nástroje Code Composer Studio. Zdrojové kódy programu signálového procesoru jsou napsány v jazyce C a assembleru.

Pro přenos dat mezi vývojovou deskou a hostitelským počítačem je použito rozhraní RTDX. Pro komunikaci pomocí RTDX je využito prostředí nástroje MATLAB.

Pro ověření dekódování je pomocí skriptu MATLABu implementována funkce, která zakóduje data příslušným konvolučním kódem, vnese do dat chybu a opět dekóduje pomocí koprocessoru a pomocí funkce MATLABu. Výsledky dekódování porovná a vrátí dekódovaná data a vyhodnocení úspěšnosti jednotlivých implementací dekódování.

Funkce má dva vstupní parametry a šest výstupních parametrů.

Vstupní parametry:

- Vektor s celočíselnými kladnými hodnotami
- Šum který bude přičten k zakódovaným datům konvolučním kódem (viz funkce MATLABu `awgn()`)

Výstupní parametry:

- Vektor s dekódovanými daty pomocí koprocessoru
- Vektor s dekódovanými daty pomocí funkce MATLABu
- Poměr špatně dekódovaných bitů koprocessorem vůči vstupním datům
- Poměr špatně dekódovaných bitů funkcí MATLABu vůči vstupním datům
- Poměr rozdílně dekódovaných bitů koprocessorem a funkcí MATLABu
- Poměr chybných bitů ve zprávě po kvantizaci vstupních dat s přičteným šumem

Funkce programu signálového procesoru je následující:

1. Otevře komunikační kanály pro RTDX.
2. Přijme rámeček vstupních slov pro dekódování.
3. Ze vstupních slov vypočte hranové metriky a uloží je do paměti.
4. Připraví obsah konfiguračních registrů koprocesoru do paměti.
5. Nastaví přenosy řadiče EDMA, tak aby došlo k nastavení parametrů koprocesoru, k vlastnímu přenosu hranových metrik pro dekódování a k uložení dekódovaných dat. Přenos dekódovaných dat je nastaven tak, aby po jeho dokončení bylo vyvoláno přerušení.
6. Spustí VCP koprocesor funkcí VCP_Start() a procesor přejde do stavu IDLE dokud nebude generováno přerušení.
7. Po spuštění koprocesoru je aktivován signál VCPXEVT pro zahájení přenosu řadiče EDMA. Je přenesen obsah konfiguračních registrů z paměti do koprocesoru, poté je přenesen rámeček s předpočítanými metriky. Dojde k dekódování a po jeho dokončení je aktivován signál VCPREVT. Z výstupního bufferu koprocesoru jsou přeneseny dekódovaná data a je generováno přerušení procesoru.
8. Po vyvolání přerušení začne procesor vykonávat obslužnou rutinu pro dané přerušení a pak pokračuje ve vykonávání programu.
9. Dekódovaná data jsou přenesena pomocí RTDX do hostitelského systému.
10. Program opět začne čekat na příchozí rámeček a celý výše popsaný proces je opakován.

8.1 Postup ověření funkce

1. Připojte vývojovou desku se signálovým procesorem k počítači pomocí USB kabelu.
2. Připojte napájení vývojové desky.
3. Spusťte Code Composer Studio
4. Otevřete projekt `vcp_edma_rtdx.pjt` a zkompilujte (*Project* → *Rebuildall*).
5. Vytvořte spojení mezi deskou a CCS (*Debug* → *Connect* nebo *ALT + C*)
6. Nahrajte zkompilovaný program do vývojové desky (*File* → *LoadProgram* nebo *CTRL + L*).
7. Spusťte MATLAB
8. Nastavte pracovní adresář tam kde je M-soubor s implementovanou funkcí pro ověření dekódování (např.: `cd c:\CCS\`).
9. Vytvořte libovolný vektor s celočíselnými kladnými hodnotami (např.: $msg = [1\ 2\ 3\ 4\ 5\ 6\ 7]$).
10. Zavolejte funkci `vcp_test()` s příslušnými parametry (např.: `[vcp_m, vit_m, vcp_err_m, vit_err_m, dec_err_m, bit_err_m] = vcp_test(msg, 0.1)`)
11. Ve výstupních parametrech jsou uloženy dekódované vektory a statistiky chyb v bitových reprezentacích číslic.
12. Volání funkce `vcp_test()` můžete opakovat s pozměněnými vstupními parametry.

9 Výsledky

Pro určení doby trvání dekódování jednoho rámce dat byl nejdříve použit profiler nástroje CCS. Po změření počtu hodinových cyklů byly tyto hodnoty porovnány s teoretickými hodnotami uvedenými v [1].

Při měření jsem přenášel rámce o délce 32, 64 a 128 kódových slov. Profiler určil, že doba potřebná pro dekódování rámců trvá v průměru 815 hodinových cyklů procesoru, bez závislosti na délce rámce. Počet naměřených hodinových cyklů je menší než počet cyklů vypočtených podle tabulky 27 v [1].

Koprocesor pracuje na čtvrtinové pracovní frekvenci než procesor. Pro dané délky rámců dekóduje data podle [1] za $((72 + 2)/6) * (F + K - 1)$ svých pracovních cyklů, kde F značí délku rámce a K je konstanta daného konvolučního kódu.

Jedním z vysvětlení, že profiler změřil tak nízký počet cyklů, je pravděpodobně nezávislost řadiče a koprocesoru na krokování hlavního procesoru a proto se počet cyklů jeví tak malý.

Pro korektní změření počtu hodinových cyklů jsem použil čítače procesoru. Čítač je nastaven před zahájením přenosu rámce a jeho dekódováním na nulovou hodnotu. Po dekódování rámce je přečtena hodnota čítače a tím také určena doba běhu dekódování. Čítač je nastaven tak, aby se jeho hodnota aktualizovala s osminou hodinové frekvence procesoru (vyplývá z hardwarové implementace čítače).

V tabulce 3 jsou uvedeny naměřené hodnoty počtu hodinových cyklů. Naměřené hodnoty zahrnují režii přenosu dat z a do koprocesoru pomocí řadiče EDMA. Řadič EDMA přenese každé 4 hodinové taktů maximálně 8 bytů. Naměřené hodnoty mohou být také ovlivněny překladačem jazyka C.

Teoretické hodnoty počtu cyklů pro dekódování rámce dekodérem jsou uvedeny v tabulce 4

Délka rámce	Hodnota čítače	Počet cyklů procesoru
32	420	3360
64	605	4840
128	1012	8096

Tabulka 3: Naměřené hodnoty počtu cyklů potřebných pro dekódování rámce

Délka rámce	Počet cyklů koprocesoru	Počet cyklů procesoru
32	469	1875
64	863	3453
128	1652	6611

Tabulka 4: Teoretické hodnoty počtu cyklů potřebných pro dekódování rámce

10 Výpis obsahu CD-ROM

Na CD se nachází text dokumentu, skript pro ověření funkce dekodéru v prostředí Matlab a projekt pro Code Composer Studio s implementací dekodéru komunikujícím přes RTDX.

Přiložené CD má následující adresářovou strukturu:

```
.
|-- doc/                text dokumentu ve formátu PDF
|-- matlab/            skript pro ověření funkce dekodéru v Matlabu
|-- vcp_edma_rtdx/     projekt CCS pro ověření funkce dekodéru
'-- readme.txt
```


Reference

- [1] Texas Instruments. *TMS320C64x DSP, Viterbi-Decoder Coprocessor (VCP), Reference Guide* [online]. Dostupné na WWW:
<http://focus.ti.com/lit/ug/spru533d/spru533d.pdf>
- [2] Texas Instruments. *TMS320C6000, Chip Support Library, API Reference Guide* [online]. Dostupné na WWW:
<http://focus.ti.com/lit/ug/spru401j/spru401j.pdf>
- [3] Texas Instruments. *TMS320C6000 DSP, Enhanced Direct Memory Access (EDMA) Controller, Reference Guide* [online]. Dostupné na WWW:
<http://focus.ti.com/lit/ug/spru234c/spru234c.pdf>
- [4] Texas Instruments. *TMS320C6000 Peripherals Reference Guide* [online]. Dostupné na WWW:
<http://coecsl.ece.uiuc.edu/ge420/datasheets/spru190d.pdf>
- [5] Rizwan Rasheed. *Reconfigurable Viterbi Decoder for Mobile Platform* [online]. Dostupné na WWW:
<http://www.ctr.kcl.ac.uk/mwcn2005/Paper/C200540.pdf>
- [6] Xilinx. *Viterbi Decoder v6.0* [online]. Dostupné na WWW:
http://www.xilinx.com/bvdocs/ipcenter/data_sheet/viterbi.pdf
- [7] Chip Fleming. *Tutorial on Convolutional Coding with Viterbi Decoding* [online]. Dostupné na WWW:
<http://home.netcom.com/~chip.f/Viterbi.html>
- [8] *The Error Correcting Codes (ECC) Page* [online]. Dostupné na WWW:
<http://www.eccpage.com/>
- [9] Russell Tessier, Sriram Swaminathan, Ramaswamy Ramaswamy, Dennis Goeckel and Wayne Burleson. *A Reconfigurable, Power-Efficient Adaptive Viterbi Decoder* [online]. Dostupné na WWW:
[url](#)
- [10] Texas Instruments. *Viterbi Decoding Techniques for the TMS320C54x DSP Generation* [online]. Dostupné na WWW:
<http://focus.ti.com/lit/an/spra071a/spra071a.pdf>
- [11] Altera. *Viterbi Compiler, User Guide* [online]. Dostupné na WWW:
http://www.altera.com/literature/ug/ug_viterbi-compiler.pdf
- [12] L. Brackenbury, M. Cumpstey, S. Furber, P. Riocreux. *An Asynchronous Viterbi Decoder* [online]. Dostupné na WWW:
<http://intranet.cs.man.ac.uk/apt/projects/lowpower/prest/async.vit.pdf>