Facoltà di Ingegneria

Università degli Studi di Trieste

Facoltà di Ingegneria
C.d.L. in Ingegneria Elettronica
Anno Accademico 2007/2008

Corso: Elettronica 3 – DSP e MCU

Realizzazione di una serie di progetti introduttivi all'utilizzo di un processore TMS320C5510

Autore:
Toffoli Valeria

Docente:
S. Carrato

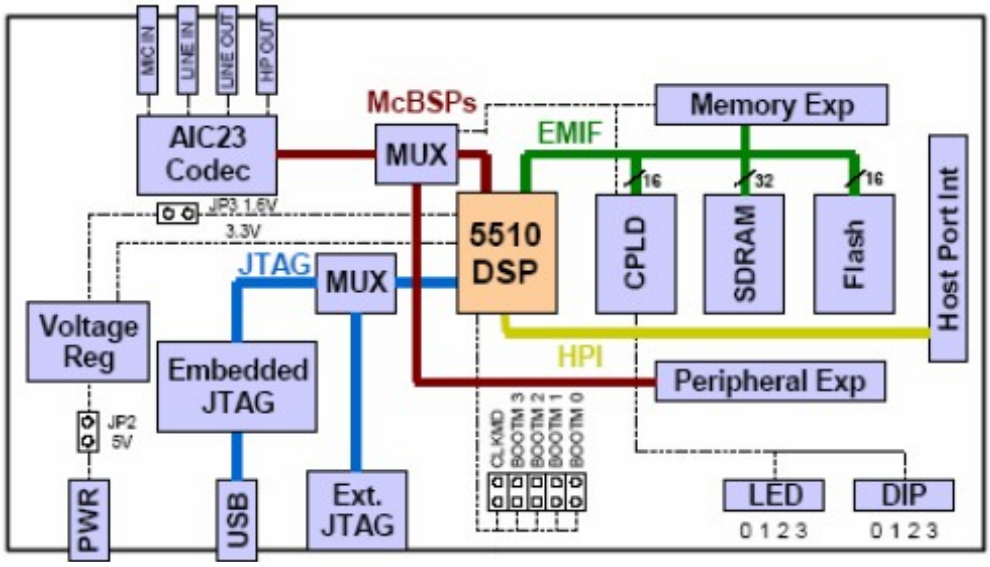
Introduzione

Lo scopo del progetto è stato la realizzazione di una serie di programmi propedeutici all'utilizzo e alla programmazione di una scheda DSP TMS320C5510. Partendo da un semplice algoritmo che consentiva la visualizzazione di un messaggio in uscita, sono stati realizzati inizialmente due progetti che sfruttassero l'hardware messo a disposizione (4 led e 4 switch) per passare poi all'utilizzo del codec AIC23 con due programmi di generazione di una sinusoide in uscita e di un sistema di filtraggio. I progetti sono stati scritti in linguaggio C e compilati con il software offerto dalla Texas Instrumantations "*Code Compose Studio 3.0*". Per la realizzazione degli algoritmi di generazione di una sinusoide e di filtraggio si è ricorso all'ambiente di lavoro *Matlab* per consentire la visualizzazione di:

- un segnale sinusoidale a partire dai dati forniti dall'utente (frequenza e ampiezza)
- i coefficienti del filtro a risposta impulsiva finita (FIR)
- la risposta in frequenza del FIR

I vari programmi sono stati poi testati collegando, quando richiesto, un generatore di funzioni alla porta in ingresso della scheda e un oscilloscopio sia sulla porta di ingresso che su quella di uscita del codec per osservare l'andamento della funzione d'onda.

I DSP e la scheda TMS320C5510



I DSP sono microprocessori ottimizzati per eseguire sequenze di istruzioni ricorrenti nel condizionamento di segnali digitalizzati e vengono impiegati in vari ambiti quali (medicina, telefonia, militare, industriale, commercio, …). Caratteristiche tipiche sono:

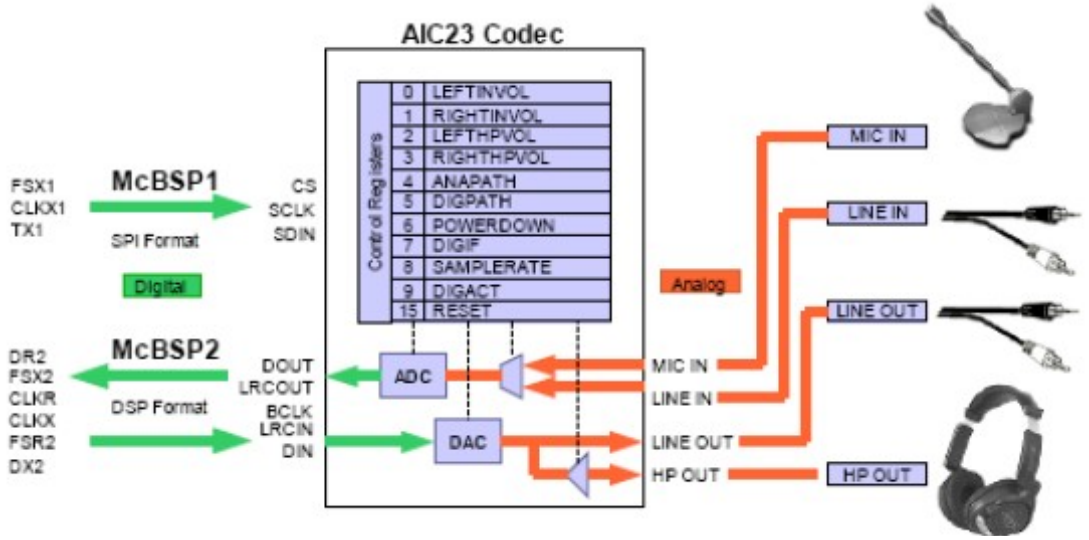
- ottimali per progetti di elaborazioni real-time e per operazioni di streaming dati.
- separazione fra memoria programma e memoria dati (architettura Harvard).
- speciali istruzioni dedicate ad operazioni SIMD (Single Instruction, Multiple Data).
- possibilità di effettuare l'accesso diretto (DMA) alla memoria di un host.
- elaborazione di segnali digitali ottenuti dalla conversione di segnali analogici, mediante uso di un convertitore analogico-digitale (ADC). Il segnale ottenuto può successivamente essere riconvertito in analogico usando un convertitore digitale-analogico (DAC).

In particolare la scheda TMS320C5510 presenta:

- il DSP 5510 della Texas Instrumentations che opera a 200MHz
- lo stereo codec AIC23
- 8 Mbytes di DRAM

- 512 Kbytes di memoria Flash non volatile
- 4 led e 4 switch accessibili dall'utente
- configurazione software della board attraverso i registri implementati nella CPLD
- single voltage power supply (+5V)

Il codec AIC23



Il TMS320AIC23 è un codec stereo audio di elevate performance che può operare su dati con lunghezza di parola 16, 20, 24 e 32 bit, e con una frequenza di campionamento compresa tra 8kHz e 96kHz. I convertitori analogici-digitali (ADCs) e digitali-analogici (DACs) che utilizza presentano un rapporto segnale-rumore superiore a 90 dBA a una frequenza di campionamento superiore a 96kHz consentendo di memorizzare il segnale di ingresso con grande fedeltà e garantendo un'alta qualità del segnale audio in uscita. L'ingresso e l'uscita analogia sono selezionate del codec e sono costituite a 4 jack audio di 3.5mm corrispondenti al microphone input, line input, line output e headphone output. La frequenza di campionamento dei segnali audio è impostata a 48 kHz e la comunicazione tra il digital signalprocessors (DSPs) e il computer avviene attraverso un universal serial bus (USB).

Nei progetti sotto riportati sono stati processati segnali audio proveniente dalla line input del codec AIC23 e sono state poi visualizzate le uscite nella line output (line input e line output sono entrambe di tipo stereo). Per rendere più efficiente il trasferimento dei dati si è ricorso al McBSP (Multichannel Buffered Serial Port). I dati audio provenienti e diretti al codec sono trasmessi attraverso la porta seriale bidirezionale McBSP2 mentre McBSP1 è utilizzato per controllare e configurare l' AIC23. I dati provenienti dal AIC23 sono costituiti da intervalli di un campione di 16-bit provenienti dal canale sinistro seguiti immediatamente da un campione di 16-bit del canale destro. Poiché gli algoritmi di processione dati operano su un canale alla volta il DSP utilizza buffer *ping-pong* per il trasferimento dei segnali. Questa tecnica prevede l'impiego di 2 buffer (denominati *ping* e *pong*) in modo tale da poter avere continuamente un buffer che impegnato in attività di trasferimento mentre l'altro è processato senza il pericolo che venir riscritto

Per esaminare il corretto funzionamento del codec mediante l'uso di un oscilloscopio collegato alla *line out* del TMS320C551 è stata generata una funzione che si limita a fornire in ingresso all' AIC23 una sinusoide e leggere il valore di uscita fornito. La libreria *dsk5510_aic23.h*, inclusa in ogni progetto che opera sul codec, da anche la possibilità di modificare la frequenza di funzionamento e il guadagno in uscita di quest'ultimo mediante il comando *void DSK5510_AIC23_setFreq(DSK5510_AIC23_CodecHandle hCodec, Uint32 freq)* e *void*

DSK5510_AIC23_outGain(DSK5510_AIC23_CodecHandle hCodec, Uint16 outGain). Di seguito è stato riportato il programma realizzato

CODICE C : TEST AIC23

```
/* Length of sine wave table */
#define SINE_TABLE_SIZE  48
/* Number of elements for DMA and McBSP loopback tests */
#define N               16
/* Pre-generated sine wave data, 16-bit signed samples */
int sinetable[SINE_TABLE_SIZE] = {
   0x0000, 0x10b4, 0x2120, 0x30fb, 0x3fff, 0x4dea, 0x5a81, 0x658b,
   0x6ed8, 0x763f, 0x7ba1, 0x7ee5, 0x7ffd, 0x7ee5, 0x7ba1, 0x76ef,
   0x6ed8, 0x658b, 0x5a81, 0x4dea, 0x3fff, 0x30fb, 0x2120, 0x10b4,
   0x0000, 0xef4c, 0xdee0, 0xcf06, 0xc002, 0xb216, 0xa57f, 0x9a75,
   0x9128, 0x89c1, 0x845f, 0x811b, 0x8002, 0x811b, 0x845f, 0x89c1,
   0x9128, 0x9a76, 0xa57f, 0xb216, 0xc002, 0xcf06, 0xdee0, 0xef4c
};
/* Codec configuration settings */
DSK5510_AIC23_Config config = {
   0x0017,  /* 0 DSK5510_AIC23_LEFTINVOL  Left line input channel volume */ \
   0x0017,  /* 1 DSK5510_AIC23_RIGHTINVOL Right line input channel volume */\
   0x01f9,  /* 2 DSK5510_AIC23_LEFTHPVOL  Left channel headphone volume */  \
   0x01f9,  /* 3 DSK5510_AIC23_RIGHTHPVOL Right channel headphone volume */ \
   0x0015,  /* 4 DSK5510_AIC23_ANAPATH    Analog audio path control */     \
   0x0000,  /* 5 DSK5510_AIC23_DIGPATH    Digital audio path control */    \
   0x0000,  /* 6 DSK5510_AIC23_POWERDOWN  Power down control */           \
   0x0043,  /* 7 DSK5510_AIC23_DIGIF      Digital audio interface format */ \
   0x0081,  /* 8 DSK5510_AIC23_SAMPLERATE Sample rate control */          \
   0x0001   /* 9 DSK5510_AIC23_DIGACT     Digital interface activation */  \
};
/************************************************************/
 *  main()                                                  */
/************************************************************/
void main()
{
   DSK5510_AIC23_CodecHandle hCodec;
   Int16 i, j;
   /* Initialize the board support library, must be first BSL call */
   DSK5510_init();
   /* Configure McBSPs for normal codec operation */
   MCBSP_config(DSK5510_AIC23_CONTROLHANDLE, &mcbspCfg1);
   MCBSP_config(DSK5510_AIC23_DATAHANDLE, &mcbspCfg2);
   /* Start the codec */
   hCodec = DSK5510_AIC23_openCodec(0, &config);
   /* Generate a 1KHz sine wave for 1 second */
   for (i = 0; i < 1000; i++)
   {
      for (j = 0; j < SINE_TABLE_SIZE; j++)
      {    /* Send a sample to the left channel */
         while (!DSK5510_AIC23_write16(hCodec, sinetable[j]));
         /* Send a sample to the right channel */
         while (!DSK5510_AIC23_write16(hCodec, sinetable[j]));
      }
    }
     // fine test su iac23
    void DSK5510_AIC23_closeCodec( hCodec);
}
```

Di seguito sono stati analizzati i programmi realizzati e stato riportato il listato del relativo linguaggio C.

Progetti led.c e switch.c

Il DSK include 4 led e 4 switch accessibili via software tramite il registro CPLD_USER_REG. Nei file dsk5510_dip.h e dsk5510_led.h (inclusi nei nostri programmi) sono specificate tutte le funzioni e i relativi parametri che si possono eseguire sull'hardware, e cioè:

- void DSK5510_DIP_init() e void DSK5510_LED_init ; sono necessarie per inizializzare i DIP switch e i led
- Uint32 DSK5510_DIP_get(Uint32 dipNum) ; consente di ricavare lo stato dello switch indicato dal valore dipNum
- void DSK5510_LED_on(Uint32 ledNum) e void DSK5510_LED_off(Uint32 ledNum) ; servono rispettivamente per spegnere o accendere il led indicato dal valore ledNum
- void DSK5510_LED_toggle(Uint32 ledNum) ; attivazione del led denominato ledNUm

Il programma led.c riportato di seguito si limiga a ripetere un ciclo infinito nel quale attiva tutti i led simultaneamente per poi disattivarli in successione (ad intervalli di tempo di 300ms).

CODICE C : LED

```
/************************************************************************/
/* main()                                                               */
/************************************************************************/
 void main()
{
   Uint32 delay;
   /* Initialize the board support library, must be first BSL call */
   DSK5510_init();
   /* Initialize the LED module of the BSL */
   DSK5510_LED_init();
   while(1)
   {
         /* Toggle LEDs  */
         DSK5510_LED_toggle(0);
         DSK5510_LED_toggle(1);
         DSK5510_LED_toggle(2);
         DSK5510_LED_toggle(3);
   for (delay = 0; delay < 3000000; delay++);
   DSK5510_LED_off(0);
   /* Spin in a software delay loop for about 300ms */
   for (delay = 0; delay < 3000000; delay++);
   DSK5510_LED_off(1);
   /* Spin in a software delay loop */
   for (delay = 0; delay < 3000000; delay++);
   DSK5510_LED_off(2);
   /* Spin in a software delay loop */
   for (delay = 0; delay < 3000000; delay++);
   DSK5510_LED_off(3);
   /* Spin in a software delay loop */
   for (delay = 0; delay < 3000000; delay++);
   }
}
```

Il programma switch.c utilizza invece la routine user_switch_read() per leggere dal CPLD i tasti premuti dall'utente mediante la funzione DSK5510_DIP_get() presente nella libreria dsk5510_dip.h e li somma logicamente per ottenere un valore corrispondente compreso tra 0 e 15. Con l'utilizzo di un counter 10 volte al secondi il valore appena descritto viene richiamato nel main() e solo in caso di una sua variazione verrà visualizzato un messaggio in uscita. Tale controllo viene effettuato memorizzando gli ultimi due stati degli switch in un opportuno array.
Inoltre il led corrispondente allo switch premuto verrà attivato.

CODICE C : SWITCH

```
/*****************************************************************************/
/* user_switches_read()                                                      */
/*****************************************************************************/
static unsigned user_switch_read (void)
{
        unsigned int temp;
        temp = 0;

        if ( DSK5510_DIP_get(0) )
        {
        /* Switch 0 is on */
        temp = 0x0001;
        }
        If ( DSK5510_DIP_get(1) )
        {
        /* Switch 1 is on */
        temp |= 0x0002;
        }
        If ( DSK5510_DIP_get(2) )
        {
        /* Switch 2 is on */
        temp |= 0x0004;
        }
        if ( DSK5510_DIP_get(3) )
        {
        /* Switch 3 is on */
        temp |= 0x0008;
        }

        /* Return value built up in temp */

  return( temp );
}
/*****************************************************************************/
/* main().                                                                   */
/*****************************************************************************/
unsigned int  main()
{
  Uint32 delay;
   static unsigned int last_switch_values[2] = {99,99};
   //static unsigned int current_switch_value = 0;
   static unsigned int counter = 0;
   unsigned int temp;
   /* Initialize the board support library, must be first BSL call */
   DSK5510_init();
   /* Initialize the LED and DIP switch modules of the BSL */
   DSK5510_LED_init();
   DSK5510_DIP_init();

        DSK5510_LED_toggle(0);
        DSK5510_LED_toggle(1);
        DSK5510_LED_toggle(2);
        DSK5510_LED_toggle(3);

while(1)
{
 /* The variable 'counter' reaches zero 10 times per second. */
 /* Check the switch reading. */
 if ( 0 == counter)
   {
    /* Read latest switch settings. */
```

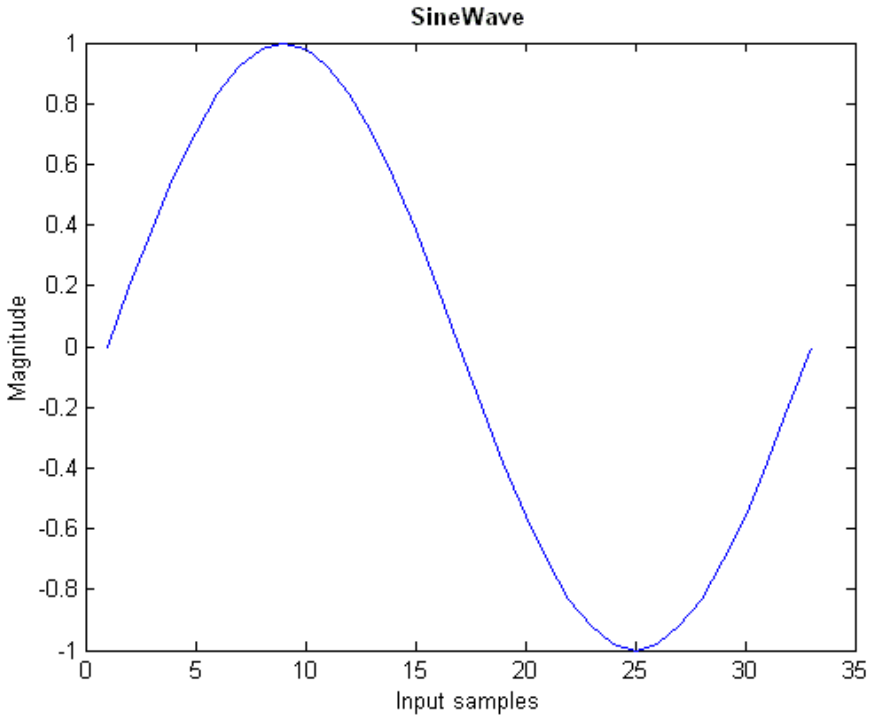
```
temp = user_switch_read();
/* Test if switch reading is the same as the last one. */
if ( temp == last_switch_values[0] )
  {
   /* Same reading as last time. Two identical values received */
   //current_switch_value = temp; /* Use new value */
   /* Only update display on Stdout when switch has just changed */
   if ( temp != last_switch_values[1] )
     {
      /* Reading at switches has changed. */
      /* Display new switch value.        */
     if ( 0 == temp )
        {
         puts("User switches = 0\n");
        }
      else if ( 1 == temp)
        {
         puts("User switches = 1\n");
        }
      else if ( 2 == temp )
        {
         puts("User switches = 2\n");
        }
      else if ( 3 == temp )
        {
         puts("User switches = 3\n");
        }
      else if ( 4 == temp )
        {
         puts("User switches = 4\n");
        }
      else if ( 5 == temp)
        {
         puts("User switches = 5\n");
        }
      else if ( 6 == temp )
        {
        puts("User switches = 6\n");
        }
      else if ( 7 == temp)
        {
         puts("User switches = 7\n");
        }
      else if ( 8 == temp )
        {
        puts("User switches = 8\n");
        }
      else if ( 9 == temp)
        {
         puts("User switches = 9\n");
        }
      else if ( 10 == temp )
        {
         puts("User switches = 10\n");
        }
      else if ( 11 == temp )
        {
        puts("User switches = 11\n");
        }
      else if ( 12 == temp)
        {
         puts("User switches = 12\n");
```

```
            }
          else if ( 13 == temp )
            {
             puts("User switches = 13\n");
            }
          else if ( 14 == temp )
            {
             puts("User switches = 14\n");
            }
          else if ( 15 == temp )
            {
             puts ("User switches = 15\n");
            }
          else
            {
             /* User switches outside the allowed range 0 - 15 */
             puts ("User switches out of range\n");
            }

         }
       }
/* Toggle LED */
/* Check DIP switch #3 and light LED #3 accordingly, 0 = switch pressed */
if (DSK5510_DIP_get(3) == 0)      /* Switch pressed, turn LED #3 on */
          DSK5510_LED_on(3);
       else                               /* Switch not pressed, turn LED #3 off */
          DSK5510_LED_off(3);

/* Check DIP switch #2 and light LED #2 accordingly, 0 = switch pressed */
if (DSK5510_DIP_get(2) == 0)      /* Switch pressed, turn LED #2 on */
          DSK5510_LED_on(2);
       else                               /* Switch not pressed, turn LED #2 off */
          DSK5510_LED_off(2);
/* Check DIP switch #1 and light LED #1 accordingly, 0 = switch pressed */
if (DSK5510_DIP_get(1) == 0)      /* Switch pressed, turn LED #1 on */
          DSK5510_LED_on(1);
       else                               /* Switch not pressed, turn LED #1 off */
          DSK5510_LED_off(1);
/* Check DIP switch #3 and light LED #0 accordingly, 0 = switch pressed */
if (DSK5510_DIP_get(0) == 0)      /* Switch pressed, turn LED #0 on */
          DSK5510_LED_on(0);
       else                               /* Switch not pressed, turn LED #0 off */
          DSK5510_LED_off(0);
 /* Shuffle switch readings along one place ready for next time. */
 /* Then read in current switch status.                          */
    last_switch_values[1] = last_switch_values[0];
    last_switch_values[0] = temp;
  }
  /* Increment counter. If at maximum or over-range, go back to zero. */
  if ( counter < 4800)
   {
   counter++;
   }
  else
   {
   /* Counter is at maximum value of 4800 or out of range. */
   /* Go back to the beginning.                            */
   counter = 0;
   }
  }
}
```

Progetto generate_sinewave.c

Simulazione con Matlab

L'obiettivo del programma era fornire in uscita un segnale seno , cioè un onda caratterizzata da una sola frequenza. È stato inizialmente creato un programma Matlab che, ricorrendo alla funzione sin() disponibile in libreria, fornisse in uscita una sinusoide nell'intervallo [0;2π].



Programma Matlab:

```
x = 0 : pi/16 : 2*pi;
waveform1 = sin(x);
plot (x);
title ('\bfSineWave');
xlabel ('Input samples');
ylabel ('Magnitude');
```

Implementazione

Il lavoro eseguito in ambiente Matlab è stato convertito in modo tale da poter esser utilizzabile da un DSK. Matlab utilizza numeri in floating point e genera una sinusoide con valori compresi tra +1 e -1, inoltre gli angoli sono espressi in radianti. Al contrario i processori DSP usano fixed-point (valori compresi tra -32767 e +32767) e le frequenze non sono più espresse in Hz ma come multipli della frequenza di campionamento F_s (48000 Hz). Per generare un'onda sinusoidale di ampiezza unitaria con un TMS320C5510 è possibile usare la funzione sine() della libreria dsplib.h che deve dunque essere inclusa nel codice C. Questa funzione prende 3 valore in ingresso:

```
sine(&input,&output,size);
```

dove:

- &input : corrisponde all'indirizzo della variabile rappresentante la frequenza desiderata, il suo valore deve esser compreso tra -32767 (corrispondente l'angolo -π) e 32767 (corrispondente all'angolo + π). Per generare una sinusoide di frequenza F questa deve esser scalata di un fattore 32767/48000 e provvedere a una sua saturazione software nel caso si ottengano valori non ammissibili.
- &output : è l'indirizzo dell'uscita dei valori della funzione sine().
- size: rappresenta il numero di sinusoidi calcolate, quindi per ricavare una sola funzione seno si avrà "sine(&input,&output,1)"

In generate_sinewave.c si è provveduto anche ad operare una saturazione software dell'ampiezza della sinusoide fornita in ingresso dall'utente qualora questa assuma valori superiori a 32767. Si ottiene dunque il programma:

CODICE C : GENERATE_SINEWAVES

```
/*****************************************************************************/
/* generate_sinewave_1()                                                     */
/* PARAMETER 1: The frequency of the sinewave between 10 Hz and 16000 Hz.    */
/* PARAMETER 2: The maximum amplitude of the sinewave between 1 to 32767.    */
/*****************************************************************************/
signed int generate_sinewave_1( signed short int frequency, signed short int amplitude)
{
 short int sinusoid;
 signed long result;
 static short int count = 0;
/* Multiply frequency by scaling factor of 32767 / 48000 */
 result = ( (long)frequency * 22368 ) >> 14 ;
if ( result > 32767)
   {
    result = 32767; /* Maximum value for highest frequency */
   }
 else if ( 0 == result)
   {
    result = 1;     /* Minimum value for lowest fequency */
   }
 else if ( result < -32767)
   {
    result = -32767;
   }
 count += (short int) result;
sine ( &count, &sinusoid, 1);
if ( amplitude > 32767 )
   {
    amplitude = 32767; /* Range limit amplitude */
   }
/* Scale sine wave to have maximum value set by amplitude */
result =  ( (long) sinusoid * amplitude ) >> 15;          /*2^15=32768*/
return ( (signed int ) result );
}
```

Progetto FIR.c

Simulazione con Matlab

Un filtro FIR presenta una relazione ingresso uscita esprimibile dalla seguente espressione:

$$y[n] = \sum_{k=0}^{N-1} h[k] \times x[n - k]$$

Più il numero dei suoi coefficienti sarà elevato maggiori saranno le sue prestazioni.
Parametri che definiscono un filtro:

- Frequenza di taglio o banda passante
- Frequenza di stop o banda di transizione
- Oscillazione massima in banda passante (*pass-band ripple*)
- Attenuazione minima in banda proibita

Nel mondo digitale è possibile realizzare un filtro ideale ma bisogna sempre tenere conto degli errori di quantizzazione, dei vincoli sulla velocità di elaborazione e dei limiti forniti dal teorema del campionamento di *Nyquist*. I coefficienti del filtro sono memorizzati nel vettore di lunghezza N e sono stati ricavati mediante la funzione:

```
hd = fir1 ( N-1, Fc, 'low', wn, 'noscale');
```

dove:
N-1 è l'ordine del filtro passa basso digitale.

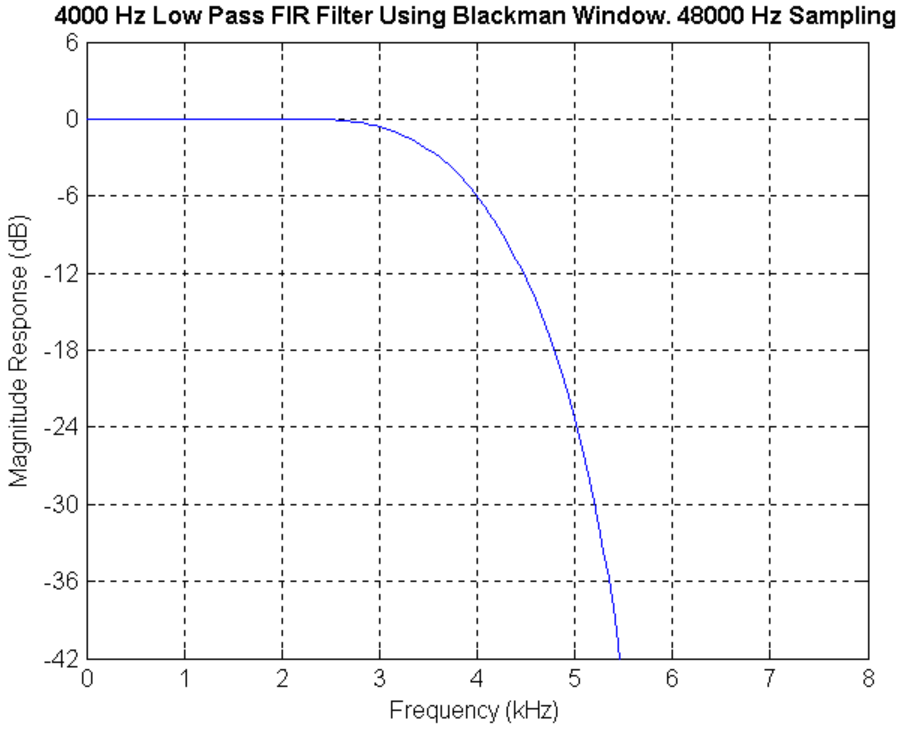
- Fc è la frequenza di taglio e deve assumare valori compresi tra 0 e 1.

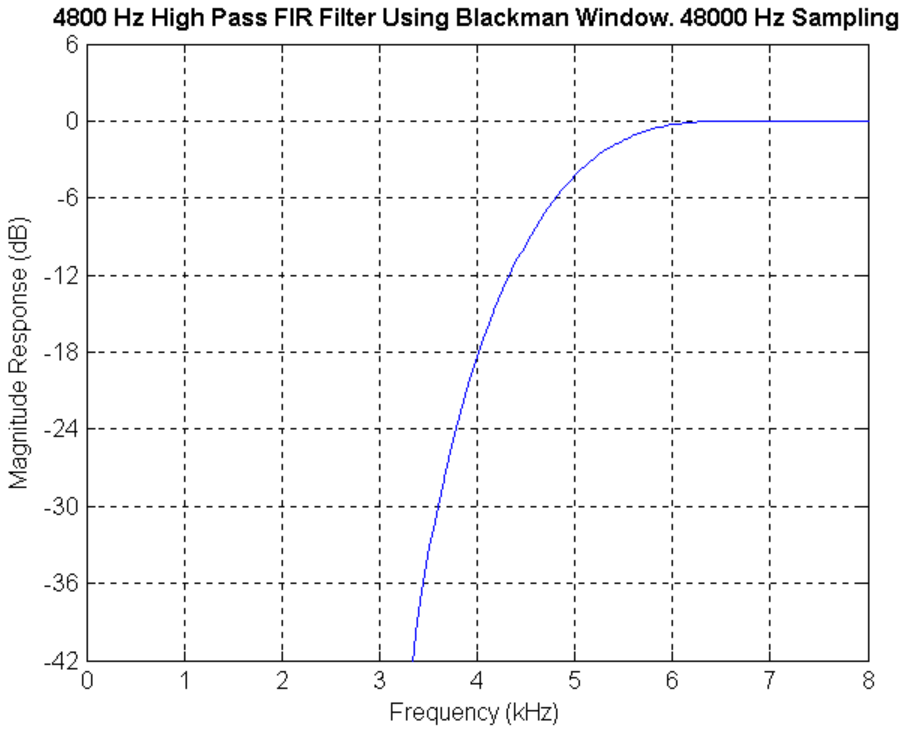
- ‘low’ indica la tipologia di filtro richiesta.
- wn è il vettore *win* di infinestramento della risposta impulsiva
- di default è opportunamente scalato in modo da assumere valore unitario dopo l’infinestramento. Per evitare ciò basta inserire 'noscale'.

La risposta in frequenza del filtro così ricavato è stata poi visualizzata e i parametri del grafico sono stati impostati mediante la funzione *set( )*. Il programma impiegato è il seguente:

```
% fir_.m
% Calculating FIR coefficients
Fs = 48000;          % Sampling frequency
FN = Fs/2;           % Nyquist frequency
Fc = 4000/FN;        % Cutoff frequency normalised to FN
N = 51;              % 51 coefficients
wn = hamming(N);
hn = fir1 ( N-1, Fc, 'low', wn, 'noscale' ); % Calculate blackman window coefficients
[H,f] = freqz ( hn, 1, 512, Fs);
magnitude = 20 * log10(abs(H));
plot ( f, magnitude), grid on
xlabel ('Frequency (kHz)');
ylabel ('Magnitude Response (dB)');
title ('4000 Hz Low Pass FIR Filter Using Blackman Window. 48000 Hz Sampling');
set (gca, 'xlim', [0, 8000])
set (gca, 'XTick', [0, 1000, 2000, 3000, 4000 5000 6000 7000 8000])
set (gca, 'XTickLabel', [0 1 2 3 4 5 6 7 8])
set (gca, 'ylim', [-42, 6])
set (gca, 'YTick', [-42, -36, -30, -24, -18, -12, -6, 0, 6])
set (gca, 'YTickLabel', [-42, -36, -30, -24, -18, -12, -6, 0, 6])
```

I coefficienti di un analogo filtro passa alto sono facilmente ricavabili con opportune modificazioni del programma sopra riportato. Di seguito sono stati visualizzati gli andamenti in frequenza dei filtri ottenuti.





Implementazione

Una volta determinati i coefficienti del filtro passa basso desiderato si è proceduto alla sua implementazione. Ogni volta che la funzione FIR_filter( ) viene chiamata dal main il dato in ingresso viene memorizzato in un array esterno e non elaborato immediatamente. La sommatoria viene eseguita in un ciclo *for* e scalata di un fattore 2^{15} per evitare possibili *overflow*. I valori dei dati in ingresso sono stati memorizzati in un array definiti esternamente insieme al suo indice e a quello dei coefficienti del filtro.
L'indirizzamento circolare negli ingressi del filtro è garantito dal primo *if* presente nel listato e inglobato nel ciclo *for*. Il secondo *if* si occupa invece della corretta memorizzazione dei nuovi dati in ingresso.

CODICE C : FIR_FILTER

```
/**************************************************************************/
#define FIR_FILTER_SIZE 64
static int value[FIR_FILTER_SIZE];        /* Retain value between calls */
signed int i;                             /* Index into filter constants */
static signed int j = 0;                  /* Index into input array */
long product;
/**************************************************************************/
/* FIR_filter()                                                           */
/* INPUTS:      1st parameter. Address of array of filter constants.      */
/*              2nd parameter. Latest input to filter                     */
/**************************************************************************/

short int FIR_filter(signed int * filter_constants, signed int input)
{
 product = 0;
 for ( i = 0 ; i < FIR_FILTER_SIZE ; i++)
 {
  /* Generate sum of products. Shift right to prevent overflow */
  /* This is first part of divide by 32767 */
  product +=  ( ((long) value[j] * filter_constants[i]) >> 5);
  if ( j < FIR_FILTER_SIZE-1)      /* Next item in circular buffer */
    {
```

```
    j++;
    }
  else
    {
    j = 0;                  /* Go back to beginning of buffer */
    }
 }

if ( j < FIR_FILTER_SIZE -1) /* Loop done. Last filter element */
    {
    j++;
    }
else
    {
    j = 0;   /* Point to new value */
    }
 product >>= 10;   /* Second part of divide by 32768 */
 value[j] = input; /* Read in new value for next time. */
 return( (signed int) product);
}
```

Conclusioni

Tutti i programmi sono risultati funzionare correttamente e costituiscono una buona base di partenza per la realizzazione di nuovi progetti sul processore TMS320C5510.

Bibliografia

1. TMS320C55x DSP CPU Reference Guide (SPRU371F), TI, Feb. 2004.
2. TMS320VC5501/…/5510 DSP Multichannel Buffered Serial Port (McBSP) Reference Guide (SPRU592E), TI, Apr. 2005.
3. TLV320AIC23 Stereo Adio CODEC Data Manual, TI, July 2001.
4. TMS320VC5510 DSK Technical Reference, Spectrum Digital, 2002.
5. TMS320C55x Chip Support Library API Reference Guide (SPRU433J), TI, Sept. 2004.