Lorenzo Orsingher **Matteo Amorth** **Mihaela Mereuta**

ANALIZZATORE DI SPETTRO

INTRODUZIONE

L'interazione dei segnali reali (analogici e continui) con le tecnologie informatiche ed elettroniche (digitali e discrete) rappresenta da sempre una sfida per ogni progettista che vi si cimenta.Nonostante ciò, questa interazione si trova alla base di innumerevoli tecnologie che spaziano dall'utilizzo quotidiano a complessi sistemi di elaborazione dati.
Con questo progetto si desidera elaborare un segnale acustico analogico in ingresso e fornire in uscita un segnale video, attraverso una porta VGA, e un segnale luminoso, attraverso una striscia LED. Il risultato finale sarà quindi rappresentare l'analisi dello spettro delle frequenze per quanto riguarda il numero e l'intensità delle stesse, a partire dal segnale in ingresso adeguatamente acquisito ed elaborato. Lo sviluppo del sistema è stato suddiviso in diverse componenti principali: acquisizione del segnale, elaborazione dei campioni e rappresentazione dei valori, nonché il relativo coordinamento attraverso la creazione di segnali specifici e implementazioni sviluppate matematiche ad hoc.

MICROFONO

La scheda si serve di un MEMs digital microphone che digitalizza l'audio dandolo in output in formato PDM (cioè con una sequenza di 0 e 1).
Lo scopo di questo modulo è quello di creare dei sample audio partendo dai dati che arrivano in ingresso dal microfono. Lo facciamo con una conversione dal formato PDM al formato PCM il quale di fatto è un intero che rappresenta il numero di uni presenti all'interno di una finestra di 128 elementi.
Con unico scopo quello del debugging, verrà anche mandato in output l'audio attraverso il on-board audio jack (J8). Per farlo basterà convertire i samples da noi creati dal formato PCM nel formato PWM e così potremo controllare che il campionamento sia avvenuto in maniera corretta.

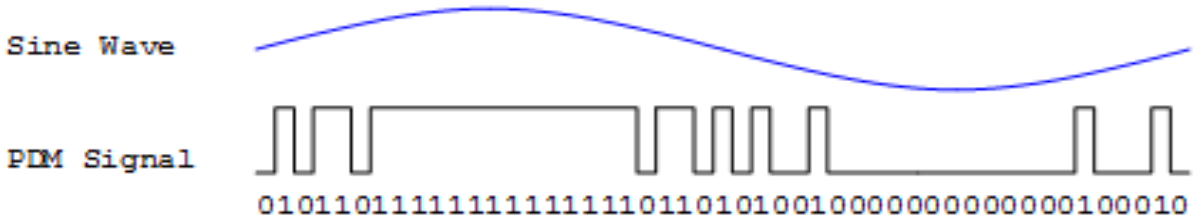


Figure 25. PDM representation of a sine wave.

```
entity mic is
    Port (

      clk_6_144MHz: in std_logic;

      -- Microphone PDM signals
      M_CLK     : out   std_logic; -- clk al microfono
      M_DATA    : in    std_logic; -- pdm data dal microfono
      M_LRSEL   : out   std_logic; -- 0-> i dati sono letti on the rising edge


      PCM_DATA :out  STD_LOGIC_VECTOR (6 downto 0);
      data_valid: out std_logic;

       -- Audio output signals
      AUD_PWM     : out   std_logic; -- Output Audio data to the lowpass filters
      AUD_SD  : out   std_logic -- Output Audio enable

    );
end mic;
```

L'entità "mic" è composta da:

- ingresso clk_6_144MHz - un clock da 6.144MHz generato grazie all'IP Core Clocking Wizard. In questo modulo avremo processi sensibili ad un clock di 3.072Mhz, non essendo però possibile generare con l'IP core tale frequenza, la creeremo manualmente dividendo quella da 6.144MHz
- ingresso M_DATA - sono i dati in formato PDM che ci arrivano dal microfono (sono una sequenza di 0 e 1)
- uscita M_CLK - clock da 3.072Mhz che viene dato in uscita al microfono
- uscita M_LRSEL - messa a 0 per indicare che i dati vengono letti sul rising edge
- uscita PCM_DATA - è il sample audio creato in formato PCM, verrà poi usato come input negli altri moduli; è un vettore di 7 bit che può avere valore massimo ->128
- uscita data_valid - comunica agli altri moduli quando un sample è pronto
- uscite AUD_PWM e AUD_SD (=1) - vengono usate per mandare l'audio in output attraverso il on-board audio jack (J8)

|| CONVERSIONE PDM -> PCM ||

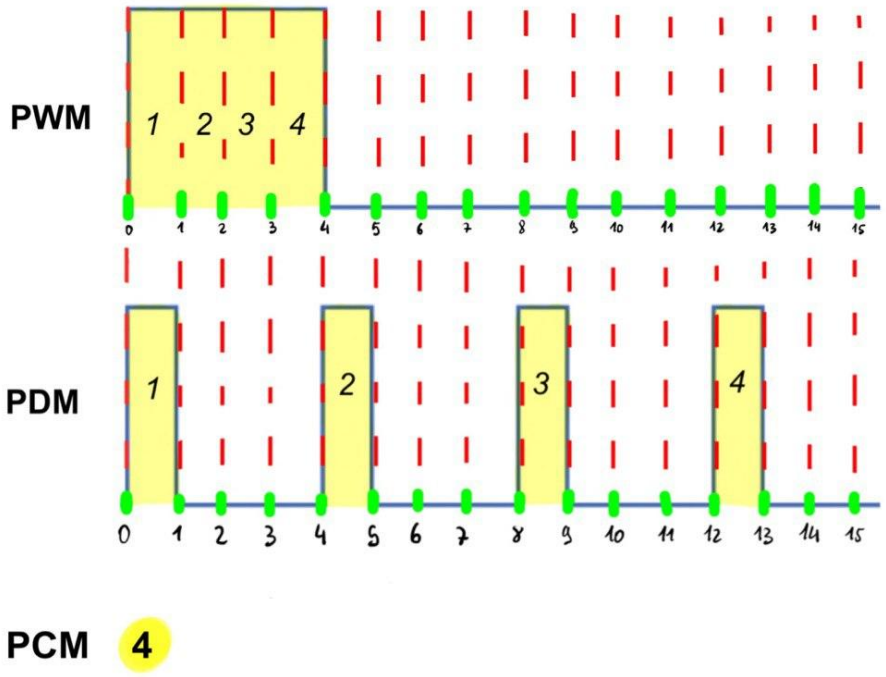
Questo processo serve per creare samples a partire dallo stream di ‘1’ e ‘0’ che ci arriva dal microfono. Ogni 3.072 MHz andiamo a controllare il valore di M_DATA in arrivo dal microfono. Se M_DATA=1 allora andremo ad incrementare il valore del nostro sample, se invece M_DATA=0 allora il valore del sample non verrà incrementato. Il sample sarà pronto dopo aver controllato il valore di 128 elementi in arrivo dal microfono e la sua validità verrà comunicata agli altri moduli mettendo ad ‘1’ il segnale data_valid che resterà alto per un solo colpo di clk_3_072MHz. In seguito il ciclo potrà ripartire da capo per creare il prossimo sample a partire dai prossimi 128 elementi.

```
pdm_pcm : process(clk_3_072MHz)
begin
  if(rising_edge(clk_3_072Mhz))then
    counterPCM <= counterPCM + 1;

    if(M_DATA = '1') then
      sample <= sample+1;
      data_valid<='0';
    end if;

    if(counterPCM= dim-1) then
        counterPCM <= 0;
        soglia<=sample;
        data_valid<='1';
        sample<=0;

        if(M_DATA = '1') then
            sample<=1;
        end if;
     end if;

  end if;
end process pdm_pcm;
```

|| CONVERSIONE PCM -> PWM ||

Per controllare che i sample creati siano validi, li convertiamo in formato PWM per poterli mandare in output. Per farlo, creiamo un’ uscita AUD_PWM che, in una finestra di 128 elementi, assume il valore ‘Z’ n volte (con n=sample) e poi assume il valore ‘0’ per il resto della finestra. Una volta completata la finestra di 128 elementi si può procedere con la conversione di un nuovo sample. In questo modo andiamo a "scomporre" i sample appena creati ricomponendoli in finestre di 128 elementi composte ciascuna da due blocchi distinti di ‘1’ e di ‘0’.

```
pcm_pwm : process(clk_3_072MHz, soglia)
begin
    if(rising_edge(clk_3_072MHz))then
        if(contPWM < soglia) then
             AUD_PWM  <='Z';
        elsif contPWM >= soglia then
             AUD_PWM <= '0';
        end if;

        contPWM<=contPWM+1;
        if(contPWM=dim-1) then
            contPWM<=0;
        end if;
    end if;
end process pcm_pwm;
```



In questa immagine possiamo vedere un esempio di conversione di un sample da PDM a PCM e a PWM
In questo caso la finestra considerata è di 16 elementi mentre la nostra è di 128.

FFT

|| INTUIZIONE e BASE ||

Questo blocco rappresenta il cuore dell'analisi dei segnali, che ne permette la manipolazione ed elaborazione in maniera semplice.
Innanzitutto, la trasformata di Fourier è uno strumento matematico utile per il passaggio dal dominio del tempo a quello delle frequenze e viceversa; in altre parole, permette di cambiare una funzione in un'altra, mantenendo i risultati della stessa consistenti e analizzabili. Essa si compone dall'integrazione della funzione iniziale sul proprio dominio e dei fattori moltiplicativi che permettono questa trasformazione. Essendo che nelle architetture elettroniche digitali (e quindi nel nostro caso) non è possibile introdurre un segnale qualsiasi in maniera continua, risulta necessario adeguare modificare la trasformata di Fourier da continua, lungo l'intero dominio dei valori, a discreta, introducendo quindi la cosiddetta DFT (discrete Fourier Transform):

$$F_k = \sum_{n=0}^{N} x_n \cdot e^{-j\frac{2\pi \cdot k \cdot n}{N}}$$

dove N è il numero di elementi in cui è suddiviso un certo intervallo continuo, n l'elemento corrente (campione) nel dominio del tempo e Fk il fattore corrispondente nel dominio delle frequenze. Volendo rappresentare il tutto sotto forma di matrice in maniera esplicita si ottiene:

$$\begin{bmatrix} \widehat{f}_0 \\ \widehat{f}_1 \\ \widehat{f}_2 \\ \widehat{f}3 \end{bmatrix} = \begin{bmatrix} e^{-j\frac{2\pi\cdot 0\cdot 0}{N}} & e^{-j\frac{2\pi\cdot 0\cdot 0}{N}} & e^{-j\frac{2\pi\cdot 0\cdot 0}{N}} & e^{-j\frac{2\pi\cdot 0\cdot 0}{N}} \\ e^{-j\frac{2\pi\cdot 1\cdot 0}{N}} & e^{-j\frac{2\pi\cdot 1\cdot 1}{N}} & e^{-j\frac{2\pi\cdot 1\cdot 2}{N}} & e^{-j\frac{2\pi\cdot 1\cdot 3}{N}} \\ e^{-j\frac{2\pi\cdot 2\cdot 0}{N}} & e^{-j\frac{2\pi\cdot 2\cdot 1}{N}} & e^{-j\frac{2\pi\cdot 2\cdot 2}{N}} & e^{-j\frac{2\pi\cdot 2\cdot 3}{N}} \\ e^{-j\frac{2\pi\cdot 3\cdot 0}{N}} & e^{-j\frac{2\pi\cdot 3\cdot 1}{N}} & e^{-j\frac{2\pi\cdot 3\cdot 2}{N}} & e^{-j\frac{2\pi\cdot 3\cdot 3}{N}} \end{bmatrix} \times \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

Utilizzando il fattore Ck semplificativo risulta:

$$\begin{bmatrix} \widehat{f}_0 \\ \widehat{f}_1 \\ \widehat{f}_2 \\ \widehat{f}3 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & C_{k=1} & {C_{k=1}}^2 & {C_{k=1}}^3 \\ 1 & C_{k=1} & {C_{k=1}}^4 & {C_{k=1}}^6 \\ 1 & C_{k=1} & {C_{k=1}}^6 & {C_{k=1}}^9 \end{bmatrix} \times \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad dove\ C_k = e^{-j\frac{2\pi\cdot k}{N}}$$

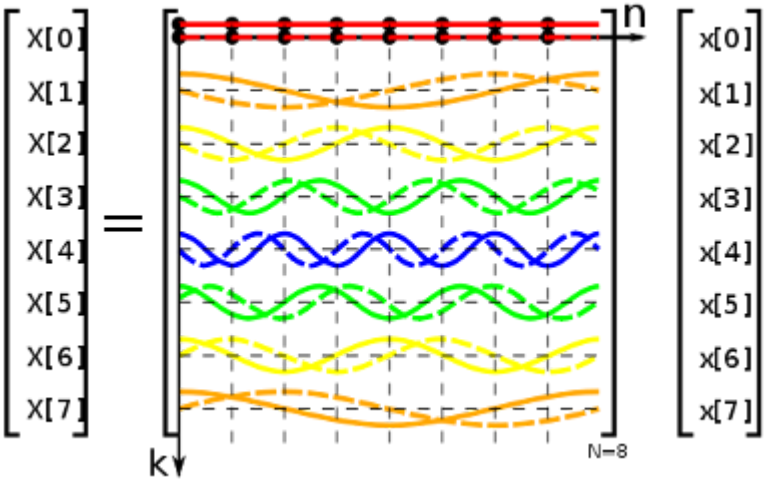
NB! Alcune considerazioni sulla matrice:

- l’indice k (= 0 ,1,2,3) rappresenta le righe: per questo motivo la prima riga è 1 (x^0 = 1);
- l’indice n (= 0 ,1,2,3) rappresenta le colonne: per questo motivo la prima colonna è 1 (x^ 0 = 1);

Come si può notare, per ogni fattore k-esimo della trasformata sono necessari n^2 calcoli per determinarlo, rendendo l’intera funzione estremamente onerosa in termini di risorse e tempo.In aggiunta, per via della teoria del campionamento, non è possibile misurare frequenze superiori alla metà della frequenza di campionamento, rendendo quindi ancora più inefficiente l’algoritmo. Da qui, risulta necessaria una semplificazione per la determinazione dei risultati, sfruttando le caratteristiche e periodicità delle funzioni sin e cos.
Una prima semplificazione possibile è grazie alla proprietà di identicità periodica delle funzioni sin e cos. Tralasciando la dimostrazione, si riscontra che superata la colonna N/2-esima il fattore Ck si ripete secondo la seguente modalità:

- se l’indice della riga è pari, il fattore Ck mantiene il segno (il fattore k-esimo al numeratore permette la ripetizione ogni 2π);
- se l’indice della riga è dispari, il fattore Ck cambia il segno (il fattore k-esimo al numeratore permette la ripetizione ogni π, alternando quindi il segno.

Questa considerazione permette di calcolare due fattori Ck per ogni k-esimo elemento in contemporanea, dimezzando quindi le operazioni per ogni riga (virtualmente, calcoliamo i fattori con la metà delle colonne).
Una seconda semplificazione possibile analizzando la logica dietro alla funzione. Osservando la matrice DFT si può notare come le righe siano “simmetriche” rispetto a quella centrale (K = N/2). Questo risulta veritiero in quanto abbiamo un’analisi delle stesse frequenze nel domino positivo e negativo. Essendo che non esistono frequenze negative, la seconda metà delle righe sta in realtà misurando anch’essa le frequenze positive semplicemente con segno invertito. Questo accorgimento permette quindi di calcolare in contemporanea due righe, dimezzando i calcoli necessari per determinarle (eccezion fatta per la componente continua della prima riga e la riga corrispondente alla metà con frequenza massima, non simmetriche).



In conclusione, grazie a questi accorgimenti è possibile implementare una versione altamente efficiente della trasformata di Fourier (FFT- fast fourier transform), diminuendo notevolmente le operazioni necessarie per il calcolo dei fattori.
Di seguito, una possibile implementazione della FFT in c++ con le semplificazioni sopra citate:

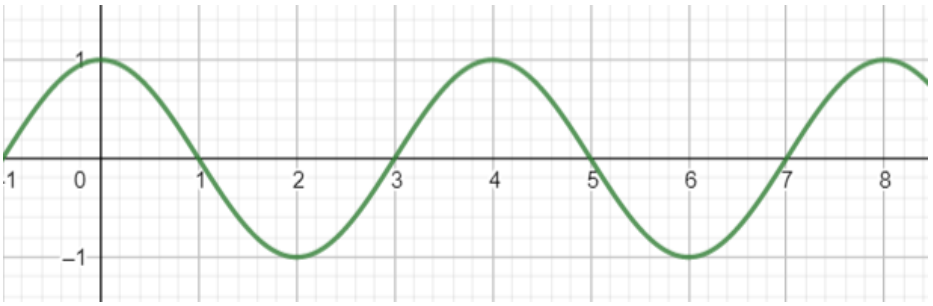
```
vector<complex<double> > FFT(vector<complex<double> > &samples) {
   complex<double> Ck;
   complex<double> CkNeg;
   vector <complex<double> > complexValues(N, 0);
   for (int k = 0; k <= N/2; k++){
       for (int n = 0; n < N/2; n++){
           //creazione elemento Ck
           Ck= polar(1.0,-2*M_PI*n*k/N);
           //assegnazione segno considerando la riga
           //e moltiplica con i campioni corrispondenti
           if (k%2==0)
               CkNeg = Ck * samples[n+N/2];
           else
               CkNeg = -Ck * samples[n+N/2];

           Ck *= samples[n];
           complexValues[k] += Ck + CkNeg;
       }
       //filtro per componente continua e massima frequenza (non sono specchiati)
        if((k != 0) && (k !=N/2))
          complexValues[N -k] = -complexValues[k];
   }
   return complexValues;
}
```

|| IMPLEMENTAZIONE ||

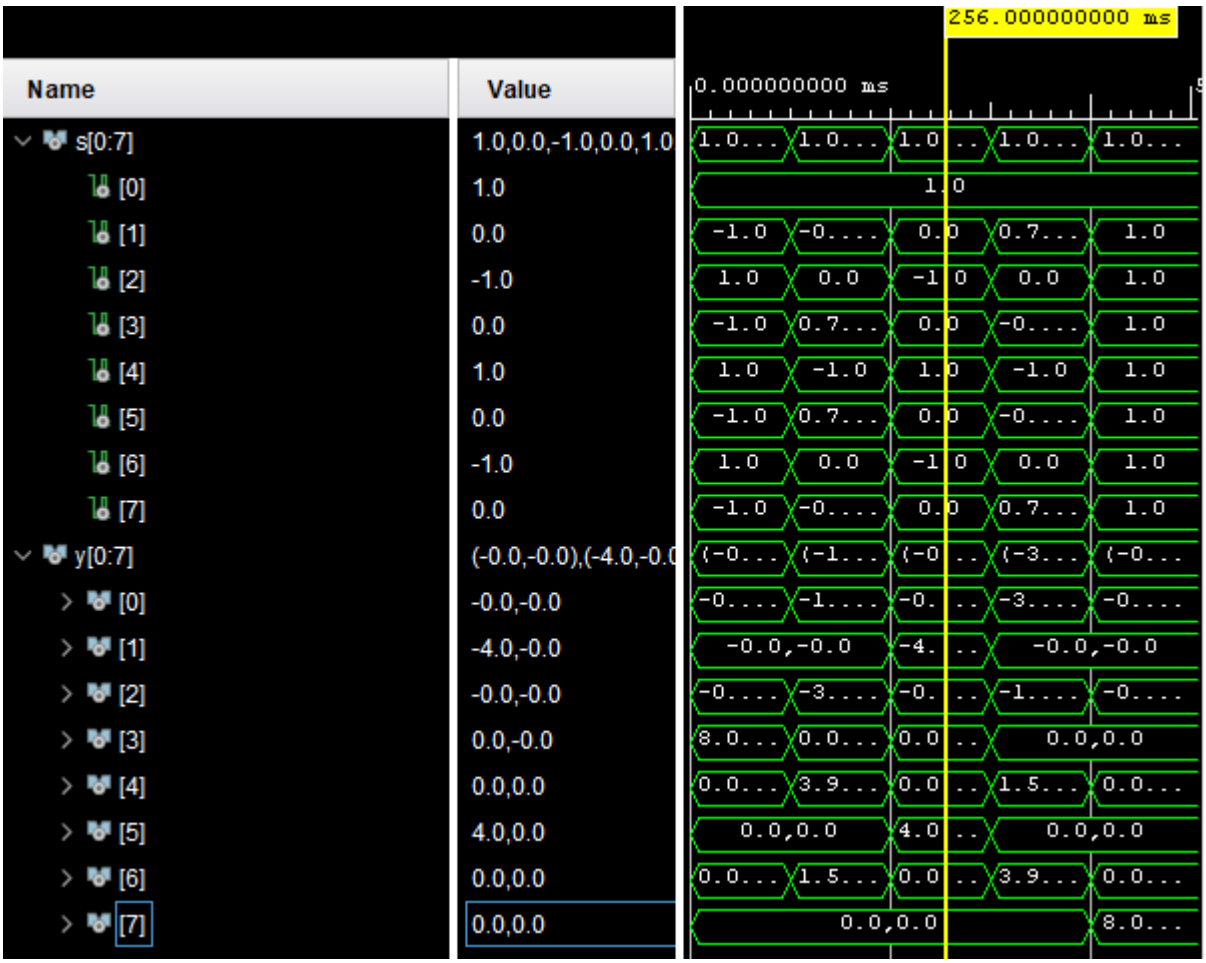
Una volta testato l'algoritmo in C++ abbiamo provato una prima implementazione in VHDL, il che risulta facile a dirsi ma ha portato ad alcuni problemi non da poco. Come prima cosa è stato necessario trovare un modo per gestire numeri complessi, fondamentali per i nostri calcoli. Vivado contiene nativamente la libreria **IEEE.math_real** che all'interno contiene tipi e funzioni per lavorare con numeri reali e con le funzioni seno e coseno, anch'esse importanti per i calcoli. Tuttavia non presenta i tool per utilizzare numeri complessi, abbiamo quindi creato una nuova libreria **MATH_COMPLEX** che funziona come una sorta di wrapper intorno a **IEEE.math_real** con l'aggiunta di vari tipi:

```
type COMPLEX         is record RE, IM: real; end record;
type COMPLEX_VECTOR  is array (integer range <>) of COMPLEX;
type COMPLEX_POLAR   is record MAG: real; ARG: real; end record;
type COMPLEX_VECTOR_FIXED is array (0 to 15) of COMPLEX;
type REAL_VECTOR is array (integer range <>) of REAL;
```

e delle relative funzioni aritmetiche specifiche per interagire correttamente.
Una volta tradotto il codice da C++ a VHDL e creato un testbench è stato possibile verificare il funzionamento teorico del programma.



Quello sopra è il segnale presente nella simulazione seguente. L'asse x indica i campioni e siccome abbiamo fatto la prova con 8 campioni, il coseno compie due periodi completi corrispondendo quindi ad un segnale reale a 2Hz. Il vettore **s** contiene le misurazioni degli 8 campioni mentre **y** gli output della parte reale della trasformata (in questo esempio siccome si tratta di un coseno ci saranno variazioni SOLO nella parte reale della trasformata).



Partendo dal fondo del vettore **y** si può notare come y[0], corrispondente alla componente continua, sia pari a 0 mentre y[5] corrispondente all'armonica di 2Hz ha correttamente valore 4. Ipotizzando che la frequenza base sia 1Hz e avendo campionato 8 volte (quindi a 8Hz) possiamo misurare massimo fino a 4Hz, motivo per cui dopo la metà la trasformata risulta specchiata.

Giunti a questo punto abbiamo trovato un altro ostacolo imprevisto: funzioni e tipi di **IEEE.math_real** non sono sintetizzabili, potevamo quindi verificare in simulazione il funzionamento teorico del programma senza essere in grado di portarlo sulla scheda.
E' quindi risultato necessario trovare un modo di codificare in modo sintetizzabile numeri complessi, numeri reali e le funzioni seno e coseno.

Vivado non supporta nativamente l'implementazione sintetizzabile di numeri reali, dopo un po' di ricerca, l'unica soluzione veramente funzionante è stata di utilizzare la libreria modificata **fixed_generic_pkg_mod** trovata in un vecchio thread sul forum di supporto Xilinx e di sfruttare la presenza di numeri a virgola fissa con segno **sfixed** (la decisione di utilizzare numeri a virgola fissa invece che a virgola mobile è dovuta al fatto che i secondi sono estremamente più difficili da gestire e debuggare). Una volta importata la libreria è stata modificata aggiungendo i tipi complessi, i vettori e l'implementazione delle funzioni aritmetiche con i nuovi tipi.

```
constant nsamples_const     : integer           := 32;
constant BITS_H     : integer                   := 8;
constant BITS_L     : integer                   := 8;

type complex is record RE, IM: sfixed(BITS_H - 1 downto -BITS_L); end record;
type fixed_vector is array(0 to nsamples_const-1) of sfixed(BITS_H - 1 downto -BITS_L);
type complex_vector is array(0 to nsamples_const-1) of complex;
```

BITS_H e **BITS_L** => il numero di bit dedicati rispettivamente alla parte intera e alla parte decimale in ogni variabile a virgola fissa
nsamples_const => numero di sample con cui viene effettuata la FFT

Un grosso inconveniente portato da questa libreria è che è compatibile solamente con VHDL 2008, il file va settato in quel formato che purtroppo in Vivado 2020.2 non supporta la simulazione, per il debug di ogni codice contenente la libreria è stato quindi necessario lavorare direttamente dalla scheda, il che ha allungato i tempi e aumentato la complessità.

A questo punto rimaneva l'implementazione delle funzioni seno e coseno, ci siamo trovati davanti a due strade, o utilizzare le serie di Taylor oppure creare una lookup table e abbiamo scelto la seconda opzione poiché molto più facile da gestire. Per risparmiare calcoli nella LUT sono stati messi direttamente i coefficienti **Ck** spiegati all'inizio del capitolo e per la generazione abbiamo usato del codice C++

```
#define N 32
#define MAX N*N+1
using namespace std;
int main(){
    int elementi [MAX];
    double real;
    double img;

    for (int i = 0; i < MAX; i++)
        elementi[i]= 0;

    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            elementi[i*j]++;

    FILE *file;
    file = fopen("LUT.txt", "w");
    if(file==NULL)
              printf("Errore in apertura file R");
    for (int i = 0; i < MAX; i++)
       {
              fprintf(file, "WHEN %d =>\n\tresult := COMPLEX'(to_sfixed(%f,BITS_H-1,
-BITS_L),to_sfixed(%f,BITS_H-1, -BITS_L));\n", i, cos(-2*M_PI*i/N),sin(-2*M_PI*i/N));
        }
    fclose(file);
    return 0;
}
```

e la LUT costruita con uno switch:

```
Package LUT is
function getCk(index: integer) return complex;
end  LUT;
Package body LUT is
function getCk(index: integer) return complex is
        variable result: complex;
        BEGIN
            CASE index IS

WHEN 0 =>
    result :=
COMPLEX'(to_sfixed(1.000000,BITS_H-1,-BITS_L),to_sfixed(-0.000000,BITS_H-1, -BITS_L));
WHEN 1 =>
    result :=
COMPLEX'(to_sfixed(0.980785,BITS_H-1,-BITS_L),to_sfixed(-0.195090,BITS_H-1, -BITS_L));
```

l'ultimo blocco presente in **IEEE.math_real** da implementare era quindi quello corrispondente al modulo e la radice quadrata

|| MODULO e RADICE ||

Una parte importante della FFT è il calcolo del modulo per poter ricostruire fedelmente la parte reale della trasformata discreta. Qui è sorto un altro problema: sebbene sia un operazione triviale con linguaggi di programmazione al alto livello, non si è rivelata altrettanto semplice in VHDL e siccome non è presente una funzione aritmetica nativa nelle librerie a nostra disposizione abbiamo deciso di procedere scrivendo i nostri moduli di per effettuare il calcolo del modulo e della radice quadrata.

modulo2.vhd

```
entity modulo2 is
  Port (

    clk : in STD_LOGIC;
    val1 : in sfixed(BITS_H-1 downto -BITS_L); --primo valore
    val2 : in sfixed(BITS_H-1 downto -BITS_L); --secondo valore
    res : out sfixed(BITS_H-1 downto -BITS_L); --risultato
    ack : out std_logic;
    rst : in std_logic  --reset asincrono

   );
end modulo2;
```

clk => clock a 100MHz
val1 => primo valore per il modulo
val2 => secondo valore per il modulo
res => risultato del modulo
ack => segnale di ACK. Rimane alto per un colpo di clock quando è pronto il modulo dei valori che erano in ingresso al modulo nell'istante del colpo di ACK precedente. Questo vuol dire che se i valori in ingresso cambiano mentre è in corso il calcolo di un modulo, quando si riceverà il segnale di ACK in uscita si avrà il risultato corrispondente ai valori che erano in entrata prima della modifica e il risultato dei valori correntemente inseriti sarà pronto all'ACK successivo
rst => segnale di reset asincrono, finchè rst è alto, modulo restituirà 0 e non arriverà nessun segnale di ACK, una volta abbassato riprenderà la normale operatività del codice

La prima parte del calcolo è semplice, si tratta di una somma di quadrati, è stato però necessario fare molta attenzione in ogni parte di codice a utilizzare variabili della corretta dimensione di bit per non incorrere nel rischio di overflow (o underflow siccome stiamo lavorando con numeri a virgola fissa con segno). Per ovviare a questo problema ogni dichiarazione di dimensione è stata modificata, da intervalli hard-coded siamo passati a range che fanno vece alle dimensioni di variabile "globali" dichiarate nella libreria modificata **fixed_generic_pkg_mod.vhd**. La variabile **BITS_H** il numero di bit dedicati a indicare la parte intera mentre **BITS_L** il numero di bit per la parte decimale.

```
val1s <= val1 * val1;
val2s <= val2 * val2;
sum <= val1s + val2s;
x <= resize(sum, (2*BITS_H)-1,-(2*BITS_L));
```

L'ultimo resize è necessario per riportare a variabile alla dimensione di lavoro, **sum** infatti è dimensione doppia (per via della moltiplicazione sopra dove vengono raddoppiati i bit) più uno (il bit dell'eventuale riporto nella somma). A questo punto è necessario fare la radice quadrata di **x** che, ricordiamo è un signed fixed point, nel caso specifico della nostra implementazione a 8 bit interi e 8 bit decimali

```
x40 <= to_slv(x);
c <= to_sfixed(signed(sqout2),c);
res <= resize(c srl (BITS_L), BITS_H-1, -BITS_L);
```

x40 è l'input del modulo della radice mentre **sqout2** è l'output. L'algoritmo da noi scelto non è in grado di fare la radice quadrata di numeri con la virgola, è quindi necessario fare un paio di conti per lavorare con numeri interi. Un escamotage è moltiplicare il numero fino ad avere una quantità significativa di bit interi poi effettuare la radice sulla parte intera e infine dividere, il tutto utilizzando potenze di due seguendo questa formula:

$$\frac{\sqrt{x\times 2^{y}}}{2^{\frac{y}{2}}}=\frac{\left(x\times 2^{y}\right)^{\frac{1}{2}}}{2^{\frac{y}{2}}}=\frac{x^{\frac{1}{2}}\times 2^{\frac{y}{2}}}{2^{\frac{y}{2}}}=x^{\frac{1}{2}}=\sqrt{x}$$

Siccome ogni moltiplicazione e in particolare ogni divisione è per potenze di 2, non è necessario

utilizzare nessuna funziona particolare ma è sufficiente effettuare prima uno shift a sinistra di **y** (a patto che y sia a sua volta un multiplo di 2) e poi uno shift a destra di **y** mezzi.

La strategia adottata è quindi shiftare il numero a sinistra finché tutti i bit decimali non sono oltre alla virgola, effettuare la radice, dopodichè shiftare il tutto a destra per metà dei bit del primo shift.

esempio: modulo tra 9 e 14, ogni comando è seguito dal risultato dell'operazione in decimale e binario

valori di input:

val1 = 0000 1001.0000 0000 9

val2 = 0000 1110.0000 0000 14

*val1s <= val1 * val1*

0000 0000 0101 0001.0000 0000 0000 0000 81

*val2s <= val2 * val2*

0000 0000 1100 0100.0000 0000 0000 0000 196

sum <= val1s + val2s 277

0 0000 0001 0001 0101.0000 0000 0000 0000

il numero a cui dev'essere fatta la radice è quindi 277

siccome effettuiamo la radice a 32 bit il numero va copiato in una variabile a 32 bit

*x <= resize(sum, (2*BITS_H)-1,-(2*BITS_L))*

0000 0001 0001 0101.0000 0000 0000 0000 277

Questo comando trasforma il numero con segno in uno standard vector perchè il modulo radice ha bisogno di un unsigned in entrata, viene anche "tolta" la virgola, l'operazione è quindi equivalente ad uno shift a sinistra di 2***BITS_L**, nel caso specifico uno shift di 16 bit a sinistra ovvero ad una moltiplicazione per 65536

x40 <= to_slv(x)

0000 0001 0001 0101 0000 0000 0000 0000 18153472

qui viene effettuata la radice quadrata

ora il risultato è presente in **sqout2** e viene copiato in **c**

c <= to_sfixed(signed(sqout2),c)

0001 0000 1010 0100. sqrt(18153472) = 4260

qui finalmente avviene lo shift a destra di **BITS_L** (8) bit e viene salvato nella

variabile res
```
res <= resize(c srl (BITS_L), BITS_H-1, -BITS_L)
```
0001 0000. 1010 0100 16.640625

andando a controllare:
sqrt(9^2 + 14^2) = 16.64332 => 16.640625

il risultato è corretto fino alla seconda cifra decimale, è una buona approssimazione.

sqrt2.vhd

Per quanto riguarda il calcolo effettivo della radice quadrata ci siamo affidati ad un algoritmo ispirato al paper *"A New Non-Restoring Square Root Algorithm and Its VLSI Implementations"*, tradotto in VHDL da *www.vhdlguru.blogspot.com* e adattato alle nostre esigenze.

```
entity sqrt2 is
    generic(N : integer := 2* (BITS_H + BITS_L));
    port (
        Clk : in std_logic;
        rst : in std_logic;
        input : in unsigned(N-1 downto 0);
        done : out std_logic := '0';
        sq_root : out unsigned(N/2-1 downto 0)
    );
end sqrt2;
```

Clk => clock a 100MHz
rst => segnale di reset asincrono (collegato a **rst** di modulo)
input => valore su cui effettuare la radice
done => segnale di fine (collegato ad **ack** di modulo)
sq_root => output con il risultato

La peculiarità di questo algoritmo rispetto ad altre implementazioni real time è che è clockato, questo ci permette di risparmiare molte risorse hardware riutilizzandole ad ogni ciclo di clock. Ciò non vuol dire che sia un algoritmo lento, anzi, ci permette di effettuare la radice quadrata di N bit in N/2 cicli, il che vuol dire che in appena 16 cicli di clock a 100MHz (0.00016 millisecondi) porta a termine il calcolo.

Siccome la FFT produce tutti i campioni complessi allo stesso momento è stato possibile parallelizzare il processo di calcolo del modulo quindi tutti i campioni sono processati insieme e in 16 colpi di clock, a prescindere dal numero di campioni, il risultato è già pronto.

```
    gen_mod:
       for j in 0 to nsamples_const-1 generate
          modulo : modulo2 port map(
            clk => clk,
            val1 => complexResult(j).RE,
            val2 => complexResult(j).IM,
            res => realTransform(j),
            ack => acks(j),
            rst => '0'
            );
       end generate gen_mod;
```

normalizer.vhd

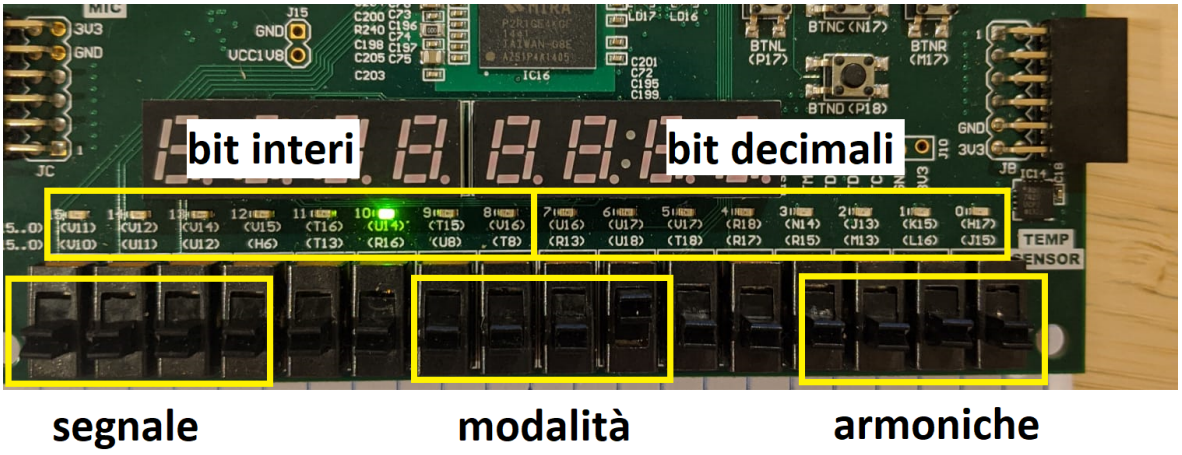
Un altro modulo importante attinente alla FFT è normalizer, seppure semplice esso si occupa di normalizzare i valori provenienti dal microfono che oscillano tra [0, 128] in un intervallo [-1, 1] che è quello su cui lavora la FFT e ci permette di effettuare tutti i calcoli senza incorrere in problemi di overflow e risparmiando una quantità considerevole di risorse.

```
entity normalizer is
  Port (

    samplesRealIn : in sfixed(BITS_H-1 downto -BITS_L);
    samplesRealNormOut : out sfixed(BITS_H-1 downto -BITS_L)

  );
end normalizer;
```

|| DEBUG FFT ||

Durante tutto l'iter di implementazione di questi moduli è stato necessario sviluppare un modello per permettere il debug real time del programma, siccome, come citato in precedenza, non era possibile eseguire simulazioni, la soluzione più diretta è stata quella di utilizzare i led integrati nella scheda. Il lato positivo di questo tipo di debug è stata la possibilità di verificare nell'ambiente di esecuzione vero e proprio il funzionamento del codice.

Per permettere di controllare più aspetti del programma senza dover ri-sintetizzare tutto il programma ogni tentativo (il processo richiede più di 20 minuti) sono stati utilizzati vari switch per interagire dinamicamente.



segnale => con questi 4 switch è possibile scegliere quale segnale analizzare. I segnali erano generati a parte e inseriti nel vettore di input in modo che imitassero il comportamento del segnale proveniente dal microfono
modalità => questi switch permettono di scegliere cosa visualizzare sui led, ci sono 3 modalità principali: una per visualizzare la parte reale dell'armonica selezionata, una per la parte immaginaria e una per il modulo. La quarta modalità viene cambiata a seconda delle esigenze e può contenere sequenze di test per verificare l'effettivo funzionamento del modulo, come anche una visualizzazione di vari segnali interni come ACK oppure i vari clock a diverse frequenze.
armoniche => la posizione di questi quattro switch viene tradotta da binario a decimale e poi utilizzata per selezionare quale armonica utilizzare. 4 sono sufficienti poichè con 32 campioni abbiamo 16 armoniche rilevanti, in fase di test utilizzando numeri maggiori di armoniche anche gli altri 2 switch non utilizzati a sinistra erano inclusi nel calcolo

I numeri binari a virgola fissa in complemento a due erano poi convertiti e comparati con i calcoli effettuati in precedenza per verificare il corretto funzionamento delle varie parti.

Per generare i segnali "hardcoded" ci siamo avvalsi di un codice C++ molto simile a quello utilizzato per la LUT

```
#define N 32
#define MAX N*N+1 //N*N+1

#define co   cos(2*M_PI*i/N)
#define co2  cos(2*2*M_PI*i/N)
#define co3  cos(3*2*M_PI*i/N)
#define co4  cos(4*2*M_PI*i/N)
#define co10 cos(10*2*M_PI*i/N)
#define co15 cos(15*2*M_PI*i/N)
#define co20 cos(20*2*M_PI*i/N)

#define si   sin(2*M_PI*i/N)
#define si2  sin(2*2*M_PI*i/N)
#define si3  sin(3*2*M_PI*i/N)
#define si4  sin(4*2*M_PI*i/N)
#define si10 sin(10*2*M_PI*i/N)
#define si15 sin(15*2*M_PI*i/N)

#define sig0 co
#define sig1 si10 + co10
#define sig2 si15 + si15 + co15
#define sig3 co2 + co3 + si3 + co3
#define sig4 co + co2 + co2 + co2 + si2

 using namespace std;
int main(){


    FILE *file;
    file = fopen("sig32_64.txt", "w");
    if(file==NULL)
             printf("Errore in apertura file R");


    for (int i = 0; i < MAX; i++)
      {
             fprintf(file, "realSignal(%d) <= to_sfixed(%f, BITS_H-1, -BITS_L)
when sw_2(0) = '0' else to_sfixed(%f, BITS_H-1, -BITS_L) when sw_2(1) = '0' else
to_sfixed(%f, BITS_H-1, -BITS_L) when sw_2(2) = '0' else to_sfixed(%f, BITS_H-1,
-BITS_L) when sw_2(3) = '0' else to_sfixed(%f, BITS_H-1, -BITS_L);\n", i,
(sig0)*63 + 64 , (sig1)*63/2+ 64, (sig2)*63/3+ 64, (sig3)*63/4+ 64, (sig4)*63/5
+ 64);

      }
    fclose(file);
    return 0;
}
```

Le varie direttive chiamate coX e siX corrispondono a segnali di seno e coseno con frequenza X Hertz su N campioni dove N è il numero si sample su cui si vuole effettuare il test.

I vari segnali erano combinati nelle 5 direttive sigX dove la prima corrisponde a un solo segnale, la seconda alla somma di due e così via. I segnali sono poi normalizzati tra (0, 128) per simulare l'input del microfono e tutto l'output è già formattato come uno switch e stampato su un file txt. E' sufficiente copiare l'output nell'architettura del modulo FFT e eseguire il test, gli output hanno questa forma (per motivi di spazio solo le prime due colonne sono riportate):

```
realSignal(0) <= to_sfixed(3.000000, BITS_H-1, -BITS_L) when sw_2(0) = '0' else to_sfixed(1.000000, BITS_H-1, -BITS_L)
realSignal(1) <= to_sfixed(2.561842, BITS_H-1, -BITS_L) when sw_2(0) = '0' else to_sfixed(0.541196, BITS_H-1, -BITS_L)
realSignal(2) <= to_sfixed(-1.197441, BITS_H-1, -BITS_L) when sw_2(0) = '0' else to_sfixed(-1.414214, BITS_H-1, -BITS_L)
realSignal(3) <= to_sfixed(2.445861, BITS_H-1, -BITS_L) when sw_2(0) = '0' else to_sfixed(0.541196, BITS_H-1, -BITS_L)
realSignal(4) <= to_sfixed(2.707107, BITS_H-1, -BITS_L) when sw_2(0) = '0' else to_sfixed(1.000000, BITS_H-1, -BITS_L)
realSignal(5) <= to_sfixed(-2.800719, BITS_H-1, -BITS_L) when sw_2(0) = '0' else to_sfixed(-1.306563, BITS_H-1, -BITS_L)
realSignal(6) <= to_sfixed(-1.089790, BITS_H-1, -BITS_L) when sw_2(0) = '0' else to_sfixed(-0.000000, BITS_H-1, -BITS_L)
realSignal(7) <= to_sfixed(0.857777, BITS_H-1, -BITS_L) when sw_2(0) = '0' else to_sfixed(1.306563, BITS_H-1, -BITS_L)
realSignal(8) <= to_sfixed(-4.000000, BITS_H-1, -BITS_L) when sw_2(0) = '0' else to_sfixed(-1.000000, BITS_H-1, -BITS_L)
realSignal(9) <= to_sfixed(-2.837741, BITS_H-1, -BITS_L) when sw_2(0) = '0' else to_sfixed(-0.541196, BITS_H-1, -BITS_L)
realSignal(10) <= to_sfixed(1.738637, BITS_H-1, -BITS_L) when sw_2(0) = '0' else to_sfixed(1.414214, BITS_H-1, -BITS_L)
realSignal(11) <= to_sfixed(-1.269985, BITS_H-1, -BITS_L) when sw_2(0) = '0' else to_sfixed(-0.541196, BITS_H-1, -BITS_L)
realSignal(12) <= to_sfixed(-1.292893, BITS_H-1, -BITS_L) when sw_2(0) = '0' else to_sfixed(-1.000000, BITS_H-1, -BITS_L)
realSignal(13) <= to_sfixed(3.976595, BITS_H-1, -BITS_L) when sw_2(0) = '0' else to_sfixed(1.306563, BITS_H-1, -BITS_L)
realSignal(14) <= to_sfixed(1.630986, BITS_H-1, -BITS_L) when sw_2(0) = '0' else to_sfixed(0.000000, BITS_H-1, -BITS_L)
realSignal(15) <= to_sfixed(-1.133676, BITS_H-1, -BITS_L) when sw_2(0) = '0' else to_sfixed(-1.306563, BITS_H-1, -BITS_L)
realSignal(16) <= to_sfixed(3.000000, BITS_H-1, -BITS_L) when sw_2(0) = '0' else to_sfixed(1.000000, BITS_H-1, -BITS_L)
realSignal(17) <= to_sfixed(1.450701, BITS_H-1, -BITS_L) when sw_2(0) = '0' else to_sfixed(0.541196, BITS_H-1, -BITS_L)
realSignal(18) <= to_sfixed(-3.045200, BITS_H-1, -BITS_L) when sw_2(0) = '0' else to_sfixed(-1.414214, BITS_H-1, -BITS_L)
realSignal(19) <= to_sfixed(0.484290, BITS_H-1, -BITS_L) when sw_2(0) = '0' else to_sfixed(0.541196, BITS_H-1, -BITS_L)
realSignal(20) <= to_sfixed(1.292893, BITS_H-1, -BITS_L) when sw_2(0) = '0' else to_sfixed(1.000000, BITS_H-1, -BITS_L)
realSignal(21) <= to_sfixed(-3.190900, BITS_H-1, -BITS_L) when sw_2(0) = '0' else to_sfixed(-1.306563, BITS_H-1, -BITS_L)
realSignal(22) <= to_sfixed(-0.324423, BITS_H-1, -BITS_L) when sw_2(0) = '0' else to_sfixed(0.000000, BITS_H-1, -BITS_L)
realSignal(23) <= to_sfixed(2.520716, BITS_H-1, -BITS_L) when sw_2(0) = '0' else to_sfixed(1.306563, BITS_H-1, -BITS_L)
realSignal(24) <= to_sfixed(-2.000000, BITS_H-1, -BITS_L) when sw_2(0) = '0' else to_sfixed(-1.000000, BITS_H-1, -BITS_L)
realSignal(25) <= to_sfixed(-1.174802, BITS_H-1, -BITS_L) when sw_2(0) = '0' else to_sfixed(-0.541196, BITS_H-1, -BITS_L)
realSignal(26) <= to_sfixed(2.504004, BITS_H-1, -BITS_L) when sw_2(0) = '0' else to_sfixed(1.414214, BITS_H-1, -BITS_L)
realSignal(27) <= to_sfixed(-1.660166, BITS_H-1, -BITS_L) when sw_2(0) = '0' else to_sfixed(-0.541196, BITS_H-1, -BITS_L)
realSignal(28) <= to_sfixed(-2.707107, BITS_H-1, -BITS_L) when sw_2(0) = '0' else to_sfixed(-1.000000, BITS_H-1, -BITS_L)
realSignal(29) <= to_sfixed(2.015024, BITS_H-1, -BITS_L) when sw_2(0) = '0' else to_sfixed(1.306563, BITS_H-1, -BITS_L)
realSignal(30) <= to_sfixed(-0.216773, BITS_H-1, -BITS_L) when sw_2(0) = '0' else to_sfixed(0.000000, BITS_H-1, -BITS_L)
realSignal(31) <= to_sfixed(-2.244817, BITS_H-1, -BITS_L) when sw_2(0) = '0' else to_sfixed(-1.306563, BITS_H-1, -BITS_L)
```

Questi corrispondono a due segnali composti a 32 campioni.

|| CONCLUSIONI FFT ||

La nostra scelta di utilizzare solo 32 campioni non è solo dovuta al fatto che siano più semplici da gestire e debuggare ma anche a problemi di risorse: per via di come è stato implementato l'algoritmo esso non è un processo clockato, quindi nonostante sulla carta sia iterativo quello che fa vivado è srotolare i cicli e crea un'unica cascata di circuiteria che effettua tutti i calcoli. Sebbene ciò renda il calcolo della trasformata *estremamente* veloce, la scheda ne risente parecchio per quanto riguarda l'utilizzo delle risorse, infatti con 64 campioni Vivado riporta over utilization di CARRY4 con conseguente errore di implementazione. Questo è uno dei motivi per cui abbiamo scelto di optare per un processo clockato ma parallelizzato sulla radice quadrata che risulta quindi essere il vero collo di bottiglia, sebbene completando il conto in appena 16 cicli rimane molto veloce, decisamente più di quanto fosse per noi necessario.
In ogni caso, al di là di qualsiasi problema legato alle performance, 32 campioni si sono rivelati più che sufficienti per lo scopo del nostro progetto permettendoci di analizzare senza problemi e in tempo reale segnali reali registrati dal microfono.

|| SINCRONIZZAZIONE MODULI MICROFONO-VGA ||

```
COMPONENT blk_mem_gen_0 is
  Port (
    clka : in STD_LOGIC;
    ena : in STD_LOGIC;
    wea : in STD_LOGIC_VECTOR ( 0 to 0 );
    addra : in STD_LOGIC_VECTOR ( 9 downto 0 );
    dina : in STD_LOGIC_VECTOR ( 6 downto 0 );
    clkb : in STD_LOGIC;
    enb : in STD_LOGIC;
    addrb : in STD_LOGIC_VECTOR ( 9 downto 0 );
    doutb : out STD_LOGIC_VECTOR ( 6 downto 0 )
  );

end COMPONENT;
```

Questa componente è una dual port RAM con depth 1024 e width 7 bit generata usando il Block Memory Generator IP Core.
--port A
clka – clock con il quale andiamo a scrivere nella porta A
ena - enable lettura /scrittura/ reset porta A
wea - enable della scrittura della porta A
addra - indirizzo della ram nel quale si sta scrivendo
dina - dato che si sta scrivendo attraverso la porta A
– port B
clkb - clock con il quale si legge dalla porta B
enb -enable lettura /scrittura/ reset porta B
addrb- indirizzo della ram dal quale si legge attraverso la porta B
doutb - dato letto attraverso la porta B

Per riuscire a sincronizzare tutti i moduli del nostro progetto, dobbiamo assicurarci di andare a scrivere e leggere da queste dual port ram con la giusta frequenza e quindi dobbiamo calcolarci i giusti valori dei segnali di sincronizzazione ena, wea e addra.

```
addraTime <= intAddraTime(9 downto 0);
intEnaTime <= not intAddraTime(10);
enaTime <= intEnaTime;
```

intAddraTime è un vettore di 11 elementi che corrisponde all'indirizzo al quale andremo a scrivere nella RAM. Il nostro indirizzo è un vettore di 10 elementi (2^10=1024 indirizzi) e, infatti, nel momento in cui andiamo ad assegnare intAddraTime all'uscita addra, andiamo a passare solo gli ultimi 10 elementi.

L'undicesimo elemento sta solo ad indicare quando si va in overflow e quindi quando si cerca di accedere al 1024° elemento che di fatto non esiste nella RAM da noi creata. In questo caso avremo intAddraTime(10)=1 -> `intEnaTime <= not 1`. E così, mettendo IntEnaTime=0, andiamo a togliere i permessi di scrittura nella porta A poiché la RAM è già stata riempita alla sua massima capienza.

```
syncTime: process(M_CLK)
   begin
      if rising_edge(M_CLK) then
        contatore <= contatore +1;
         if contatore >= 307200 then--devo resettare la RAM
              intAddraTime <= (others => '0');
              contatore <= 0;
         end if;

         if weaTime = '1' then --un dato valido
            if intEnaTime = '0' then -- blocking condition
               null;
            else
               data_out_vga <= PCM_DATA;
               intAddraTime <= intAddraTime + '1';
            end if;
         end if;
      end if;
   end process;
```

Con questo processo siamo riusciti a calcolare il giusto valore di addra in ogni istante tenendo conto di:

- l'ingresso wea che arriva dal modulo del microfono e sta a segnalarci quando un sample è appena stato creato ed è pronto a essere memorizzato nella RAM (frequenza di 48KHz)
- il segnale intEnaTime che ci dice quando abbiamo riempito un blocco da 1024
- l'ingresso M_CLK che è un clock da 3.072MHz

Come prima cosa andiamo a creare la condizione di reset dell'indirizzo al quale stiamo scrivendo. Volendo infatti poter rappresentare a video questi sample, dobbiamo assicurarci che essi non varino con una velocità troppo elevata per poter essere apprezzati dall'occhio umano. Andiamo quindi ad assicurarci che i dati all'interno della RAM rimarranno gli stessi per circa 100 ms e quindi che l'indirizzo di scrittura addra torni a 0 (reset) con una frequenza di soli 10 HZ (=3.072MHz/307200).

Andremo invece ad incrementare l'indirizzo di scrittura solo quando:

1. wea=1 -> un nuovo sample è pronto
2. ena=1 -> non c'è la condizione di blocco e quindi non abbiamo ancora raggiunto l'ultimo indirizzo della RAM

Quando wea=1 ma ena=0 allora continuiamo a ciclare senza cambiare addra finché non saranno passati 100 ms e addra sarà stato azzerato (ena=1).

Nel nostro codice clka è 100MHz ma in realtà possiamo scrivere un dato nella RAM solo quando abbiamo il permesso di farlo (ena=1 e wea=1) e solo all'indirizzo da noi calcolato.

|| SINCRONIZZAZIONE MODULI FFT-VGA ||

```
COMPONENT blk_mem_gen_1
  PORT (
    clka : IN STD_LOGIC;
    ena : IN STD_LOGIC;
    wea : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
    addra : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
    dina : IN STD_LOGIC_VECTOR(17 DOWNTO 0);
    clkb : IN STD_LOGIC;
    enb : IN STD_LOGIC;
    addrb : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
    doutb : OUT STD_LOGIC_VECTOR(17 DOWNTO 0)
  );
END COMPONENT;
```

Questa componente è una dual port RAM con depth 16 e width 18 bit generata usando il Block Memory Generator IP Core.
Ha lo stesso funzionamento descritto precedentemente per `blk_mem_gen_1`, cambiano soltanto le dimensioni della RAM. Anche in questo caso infatti dobbiamo calcolarci i giusti valori dei segnali di sincronizzazione ena, wea e addra.

```
syncFreqOut: process (ck100MHz)
 begin
  if rising_edge(ck100MHz)then
       contatoreF <= contatoreF +1;

       if contatoreF >= 10000000 and indice >= 16 then --devo resettare RAM
           indice <= 0;
           contatoreF <= 0;
       end if;

       if(freqReady = '1' or indice > 0)then
            if(indice >= 16)then
                enaFreq <= '0';
                weaFreq <= '0';
            else
                enaFreq <= '1';
                weaFreq <= '1';
                data_freq <= vectorOfFreq(indice);
                addraFreq <= std_logic_vector(to_unsigned(indice, addraFreq'length));
                indice <= indice +1;
            end if;
        end if;
   end if;
end process;
```

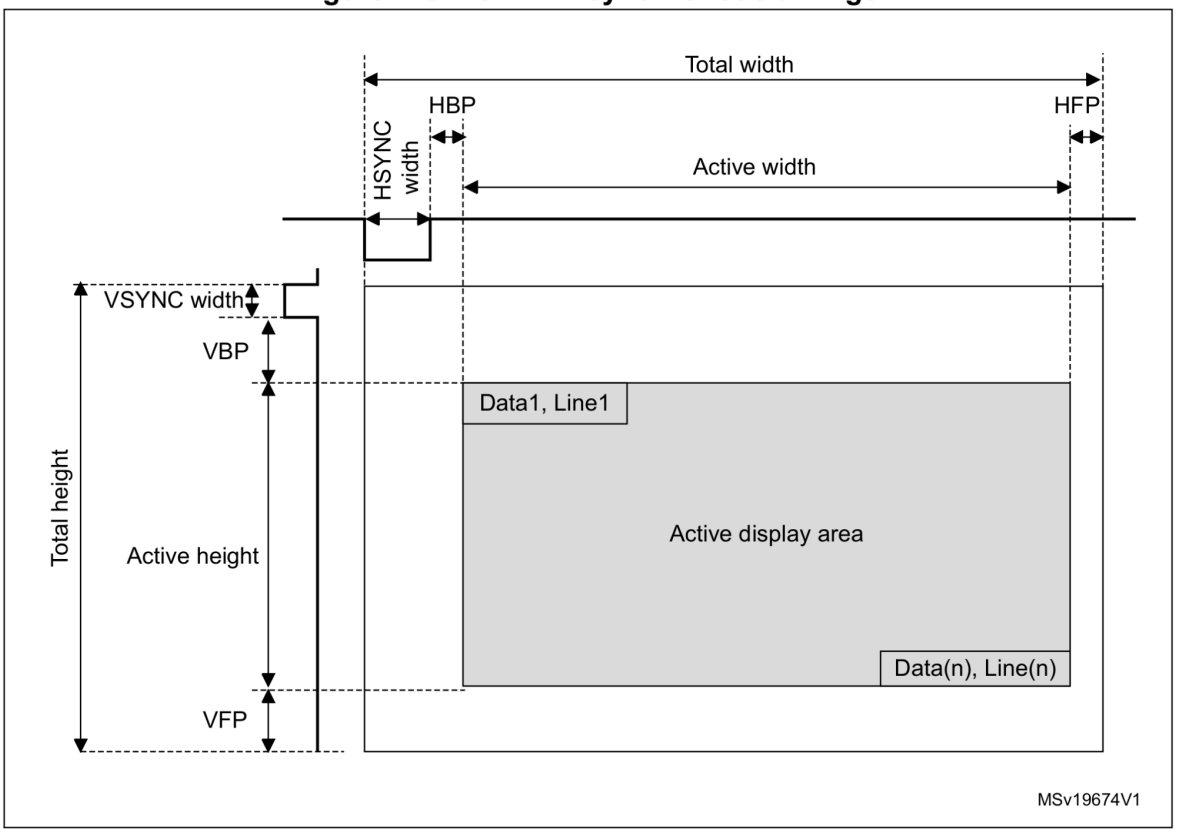
Con questo processo andiamo a determinare il giusto valore di data_freq e dei segnali di sincronizzazione ad ogni colpo di ck100MHz in modo da salvare correttamente in RAM ognuna delle 16 armoniche.
`vectorOfFreq` è il vettore contenente le 16 armoniche calcolate dalla FFT ognuna delle quali è rappresentata da un vettore a 18 bit. Ad ogni colpo di ck100MHz andiamo a mettere in RAM un'armonica diversa e una volta finite aspettiamo 100 ms. Nel mentre, poniamo enaFreq ed weaFreq a 0 in modo da non poter più scrivere niente in RAM per quei 100 ms, solo una volta terminati quelli possiamo ricominciare il ciclo e memorizzare il prossimo vettore da rappresentare.

VGA

Per quanto riguarda il modulo VGA, abbiamo una prima parte che gestisce i segnali VGA e una seconda che si occupa della creazione dell'immagine rappresentata nel nostro caso specifico.
La prima delle due ha un'implementazione molto standard e quindi non verrà trattata durante questa relazione. Ci limitiamo ad inserire la seguente immagine che aiuta a capirne il funzionamento.

Risoluzione: 640x480@60Hz
pixel clock = 25MHz – frequenza con cui si passa da un pixel all'altro

Figure 115. LCD-TFT synchronous timings



The HBP and HFP are respectively the horizontal back porch and front porch period.
The VBP and the VFP are respectively the vertical back porch and front porch period.

|| DOMINIO DEL TEMPO ||

La rappresentazione dell'audio nel dominio del tempo è visualizzabile in viola nella metà superiore dello schermo.
Considerando le dimensioni scelte, riusciremo a rappresentare 640 samples contemporaneamente. Ogni batch di 640 samples verrà mostrata per 100 ms così da rendere apprezzabile all'occhio umano l'immagine.

```
timeInVGA: blk_mem_gen_0
  PORT MAP (
    clka => ck100MHz,
    ena =>enaTime , -- active while counting
    wea(0) => weaTime,  -- wea is std_logic_vector(0 downto 0) ...
    addra => addraTime,
    dina => data_in_vga,
    clkb => ckVideo,  -- Video clock
    enb => '1',
    addrb => vecadrHor,
    doutb => TimeSample
  );
```

timeInVga è una dual port RAM (con depth 1024 e width 7 bit) nella quale, attraverso la porta A, andiamo a scrivere 1024 samples temporali uno per volta (data_in_vga) usando gli ingressi di sincronizzazione enaTime, weaTime e addraTime calcolati nel modulo FFT. Dalla porta B invece andiamo a leggere questi dati.
Ad ogni colpo di ckVideo (25MHz -> pixel clock) andiamo a prendere l'indirizzo della RAM che corrisponde alla coordinata orizzontale del pixel che stiamo attualmente aggiornando e mettiamo il suo contenuto in TimeSample.
In questo modo all'interno dell'arco temporale di 100 ms andremo a rappresentare sempre e solo i primi 640 samples.

|| DOMINIO DELLA FREQUENZA ||

La rappresentazione dell'audio nel dominio del tempo è visualizzabile in giallo nella metà superiore dello schermo.
Considerando le dimensioni scelte, riusciremo a rappresentare 16 armoniche contemporaneamente. Ogni batch di 16 armoniche verrà mostrata per 100 ms così da rendere apprezzabile all'occhio umano l'immagine.

```
freqInVGA: blk_mem_gen_1
 PORT MAP (
   clka => ck100MHz,
   ena =>enaFreq ,
   wea(0) => weaFreq,
   addra => addraFreq,
   dina =>  data_freq,
   clkb => ckVideo,
   enb => '1',
   addrb => vecadrHorDivided,
   doutb => freqSample
 );
```

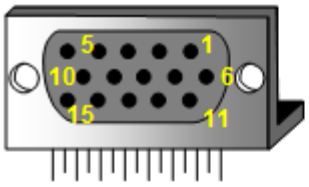
```
horRamClock: process(ckVideo)
begin
    if(rising_edge(ckVideo))then
        if(cntH <= 40 or cntH = 799)then
            vecadrHorDivided <= "0000";
        elsif(cntH <= 80)then
            vecadrHorDivided <= "0001";
        elsif(cntH <= 120)then
            vecadrHorDivided <= "0010";
        elsif(cntH <= 160)then
            vecadrHorDivided <= "0011";
        elsif(cntH <= 200)then
            vecadrHorDivided <= "0100";
        elsif(cntH <= 240)then
            vecadrHorDivided <= "0101";
        elsif(cntH <= 280)then
            vecadrHorDivided <= "0110";
        elsif(cntH <= 320)then
            vecadrHorDivided <= "0111";
        elsif(cntH <= 360)then
            vecadrHorDivided <= "1000";
        elsif(cntH <= 400)then
            vecadrHorDivided <= "1001";
        elsif(cntH <= 440)then
            vecadrHorDivided <= "1010";
        elsif(cntH <= 480)then
            vecadrHorDivided <= "1011";
        elsif(cntH <= 520)then
```

```
            vecadrHorDivided <= "1100";
        elsif(cntH <= 560)then
            vecadrHorDivided <= "1101";
        elsif(cntH <= 600)then
            vecadrHorDivided <= "1110";
        else
            vecadrHorDivided <= "1111";
        end if;
    end if;
end process;
```

freqInVga è una dual port RAM (con depth 16 e width 18 bit) nella quale, attraverso la porta A, andiamo a scrivere 16 armoniche (data_freq) usando gli ingressi di sincronizzazione enaFreq, weaFreq e addraFreq calcolati nel modulo FFT. Dalla porta B invece andiamo a leggere questi dati.
Ad ogni colpo di ckVideo (25MHz -> pixel clock) andiamo a prendere l'indirizzo della RAM che corrisponde a `vecadrHorDivided` il quale viene ricalcolato ad ogni colpo di ckVideo all'interno del processo horRamClock.
In questo modo all'interno dell'arco temporale di 100 ms andremo a rappresentare le stesse 16 armoniche.

|| RAPPRESENTAZIONE ||

La scheda Nexys4 DDR usa 14 segnali per creare una porta VGA con 4 bit per ogni canale RGB e 2 segnali standard di sync (HS- Horizontal Sync e VS- Vertical Sync). Ciò vuol dire che ogni colore verrà rappresentato da 12-bit.



Pin 1: Red	Pin 5: GND
Pin 2: Grn	Pin 6: Red GND
Pin 3: Blue	Pin 7: Grn GND
Pin 13: HS	Pin 8: Blu GND
Pin 14: VS	Pin 10: Sync GND

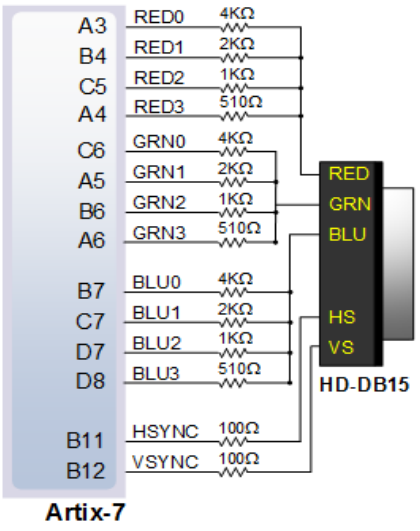
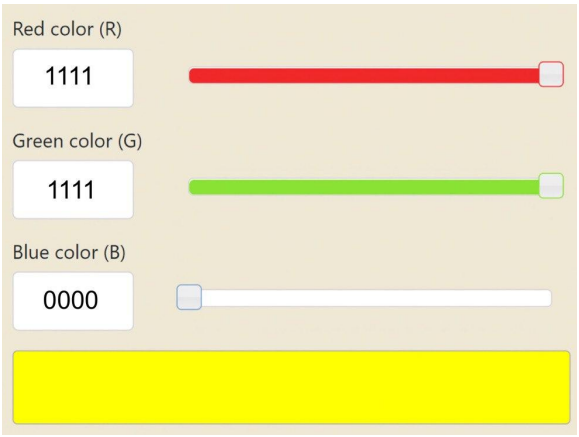
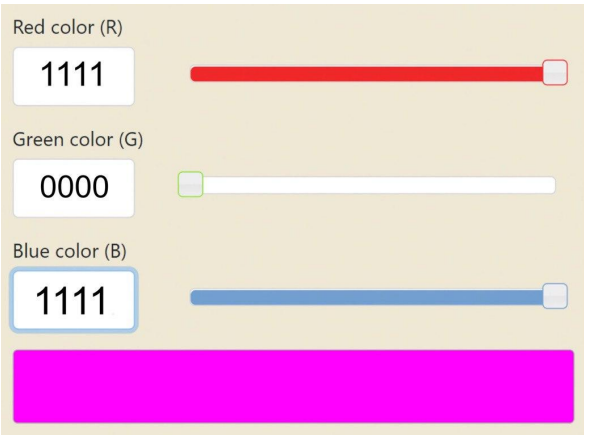


Figure 11. Nexys4 DDR VGA interface.

```
colora: process(ckVideo)
begin
  if(rising_edge(ckvideo)) then
     red <= "0000";
     green <= "0000";
     blue <= "0000";
     if(flgActiveVideo = '1') then
        if(adrVer <= V_DISPLAY/2 and
                        adrVer >= (V_DISPLAY/2 -to_integer(unsigned(timeSample)))) then
           red <= "1111";
           blue <="1111";
         end if;
         if(adrVer > V_DISPLAY/2 and
                       adrVer >= (V_DISPLAY -( to_integer(unsigned(freqSample))*6)))then
           green <= "1111";
           red <= "1111";
          end if;
       end if;
     end if;
end process;
```

Con questo ultimo processo, infine, andiamo a rappresentare i sample nel tempo e quelli in frequenza.





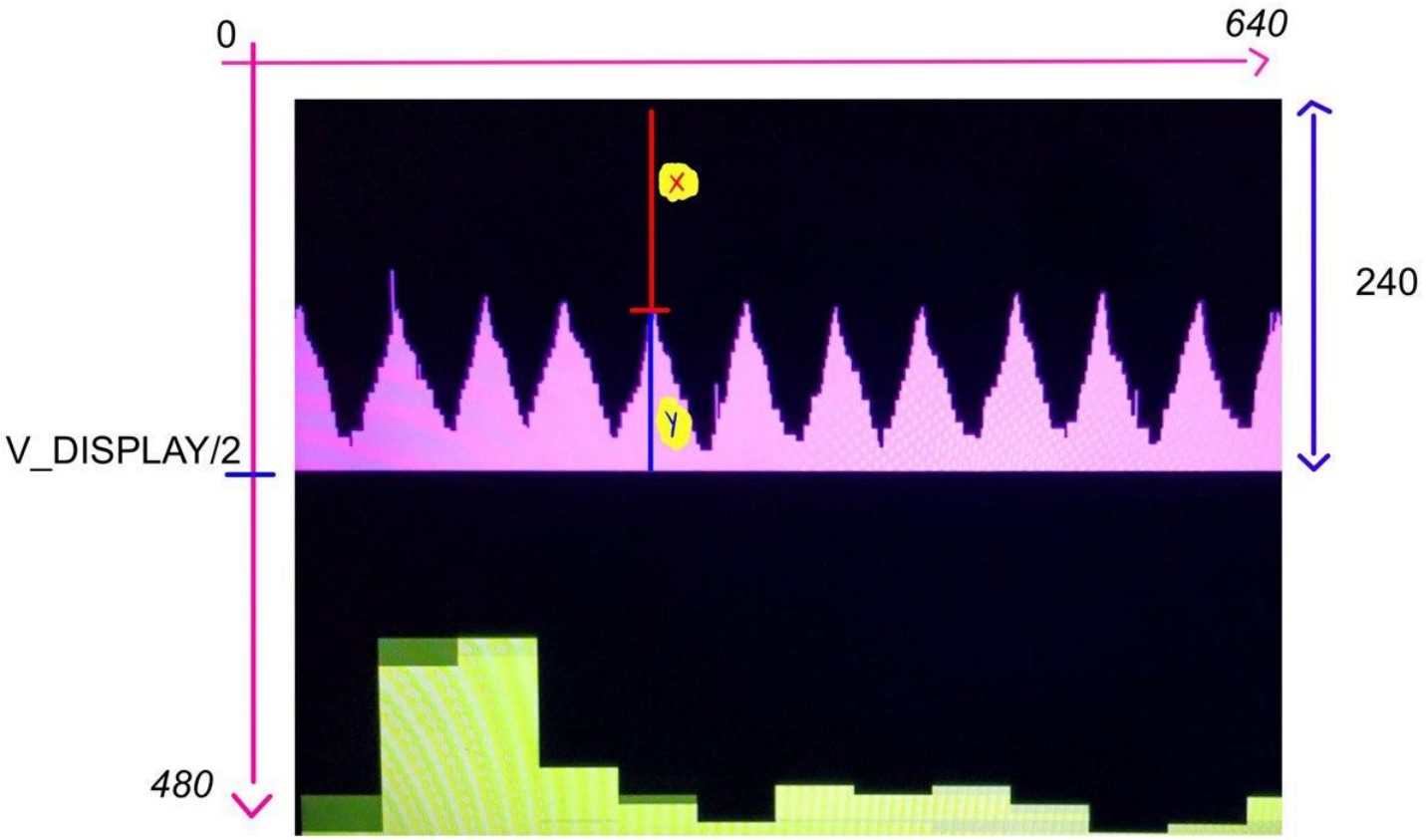
Mettiamo come default RED, GREEN e BLUE a 0 in modo da avere uno schermo nero come sfondo.

TEMPO

Mettiamo RED=1111 e BLUE=1111 (viola) solo se siamo nella metà superiore dello schermo e se l'attuale coordinata verticale è maggiore del risultato della sottrazione V_DISPALY/2 (metà schermo=240) sottratto il valore del sample temporale che nella RAM è all'indirizzo dell'attuale coordinata orizzontale (il risultato della sottrazione è nell'immagine rappresentato da **X**). La porzione che risulterà viola a seguito di questa operazione è quella appunto compresa tra 240 e l'attuale valore del sample (<128), nell'immagine è rappresentata da **Y**.

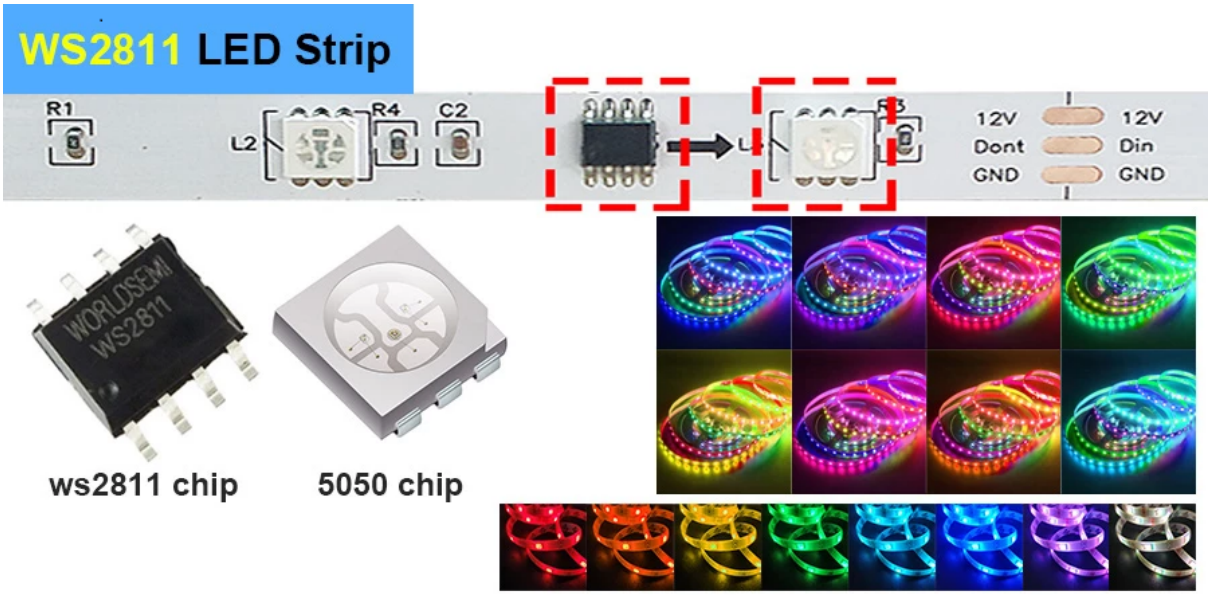
FREQUENZA

Seguendo lo stesso ragionamento mettiamo RED=1111 e GREEN=1111 (giallo) solo se siamo nella metà inferiore dello schermo e se l'attuale coordinata verticale è maggiore del risultato della sottrazione V_DISPALY (480) sottratto il valore dell'armonica che è attualmente estratta dalla RAM moltiplicata per 6 (normalizzazione).



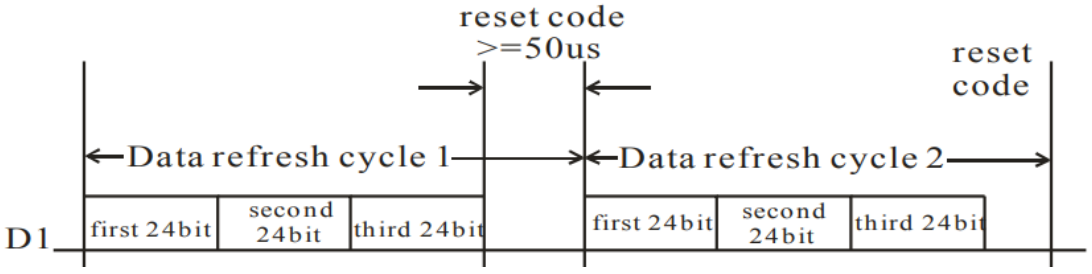
LED

Questa parte del progetto è dedicata alla creazione di un'interfaccia per poter mostrare in tempo reale il livello del volume registrato dal microfono attraverso una striscia di led.
I led in questione sono stati acquistati separatamente da Aliexpress e si tratta di questo modello:



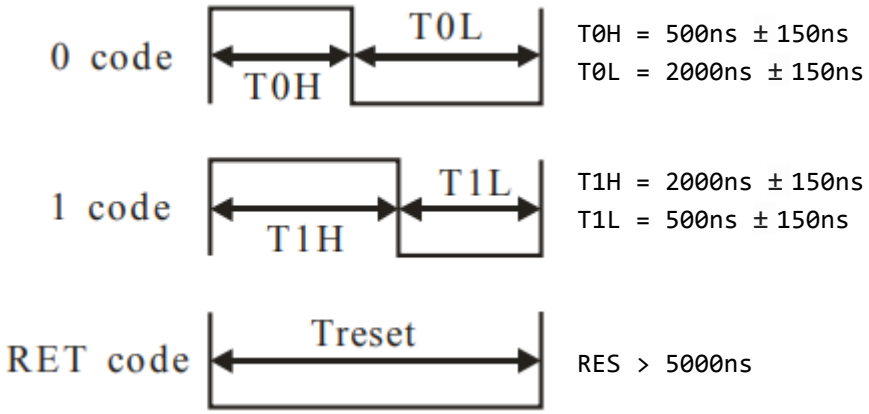
La striscia è costituita da una serie di chip WS2811 a cascata ognuno dei quali controlla gruppi di 3 LED, la striscia non è quindi indirizzabile a singolo led ma solo tre a tre, in ogni caso per lo scopo del nostro progetto non si è rivelato un problema.

Il primo passo è stato trovare un datasheet affidabile per capire il funzionamento e le tempistiche del protocollo. Qui si è presentato il primo problema perché nonostante sul sito il chip fosse pubblicizzato come WS2811 dopo un esame più attento ci siamo accorti si trattasse invece di un GS1903 di cui non si trovava quasi nulla online. Dopo un po' di ricerca si è rivelato essere un clone del circuito integrato UCS1903 su cui abbiamo basato il resto del lavoro



Il protocollo è relativamente semplice: siccome i GS1903 sono a cascata vanno inviate tante sequenze di 24 bit quanti i chip che si vogliono raggiungere seguite poi da un segnale di reset. Ogni circuito GS1903 leggerà i primi 24 bit in entrata dopodichè se non riceverà il segnale di reset (linea bassa per almeno 50us) allora inoltrerà i 24 bit al chip successivo memorizzando i nuovi 24 in arrivo.

Una volta ricevuto il segnale di reset il chip tratterà i 24 bit come una sequenza 8 + 8 + 8 in ordine BRG e illuminerà i 3 LED 5050 di conseguenza.
Per distinguere tra bit 1 e bit 0 il chip usa la modulazione PWM e si affida a questo standard:



levelsBRG.vhd

```
entity levelsBRG is
    Port (
        clk_out1 : in  std_logic;
        zout : out std_logic;
        DATA :in  STD_LOGIC_VECTOR (6 downto 0);
        volume : out std_logic_vector (12 downto 0)
    );
end levelsBRG;
```

clk_out1 = clock a 100Mhz
zout = questo è il segnale che poi arriverà direttamente al bus dei led
DATA = in questo vettore di 7 bit passa il valore PDM istantaneo preso direttamente dal microfono
volume = questo vettore di 12 bit veniva utilizzato in fase di debug per visualizzare sui led della board il volume allo stesso modo di come sarebbe stato poi visibile sulla striscia

Il processo **bit_clk_gen** serve a generare **bit_clk**, ovvero il clock a 400KHz che è la frequenza operativa della striscia led, la velocità con cui vanno inviati bit ai chip.
Il secondo processo, **createGreen** (chiamato così perché inizialmente generava solo il colore verde, poi sono stati aggiunti gli altri), si occupa di indicare al modulo **ledStrip** quali bit generare a seconda del livello del volume.
Nella variabile **b** viene salvato il valore del bit che deve essere generato mentre **r** viene portata alta se si è concluso di inviare i bit ed è il momento di inviare il segnale di reset.
Il processo è sincronizzato con i 400KHz della striscia led quindi ad ogni ciclo

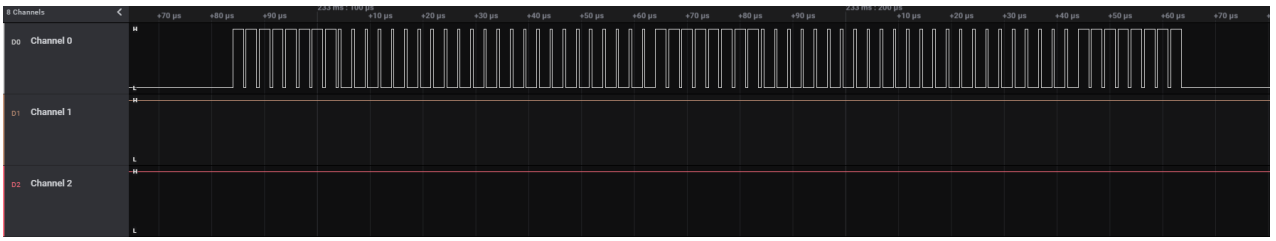
viene aumentato un contatore corrispondente a quale bit è arrivato. In tutto vengono inviati 120 bit corrispondenti a 5 livelli da 24 bit più 40 di reset signal.
I vari colori sono generati “a mano” con una cascata di if in cui viene controllato a quale livello si è arrivati e, a seconda del livello del volume, se attivare i led o meno.

ledStrip.vhd

```
entity ledStrip is
    Port (
        clk : in  std_logic;    -- 100MHz clock
        bit_clk : in std_logic; --bit clock 400KHz
        b : in std_logic; -- 0 or 1
        r : in std_logic; -- reset signal
        z : out std_logic := '0' -- signal goes to the LED strip

    );
end ledStrip;
```

clk = clock a 100MHz
bit_clk = clock a 400KHz per sincronizzarsi con la striscia led
b = segnale in entrata indica che bit va generato
r = segnale di reset
z = il segnale in uscita che arriva poi ai led

La variabile **bit_state** indica in che stato ci troviamo, ovvero se il modulo in uscita deve generare un 1, uno 0 oppure il segnale di reset. Esso viene aggiornato a seconda dei segnali **b** e **r** ed è sincronizzato col modulo **levelsBRG.**
Nel processo **pwm_modulation** il programma cicla 250 volte a 100MHz corrispondente ai 2500ns che occorrono per inviare il bit e utilizzando un contatore assegna il valore all'uscita a seconda del tipo di segnale richiesto.



Questa è una lettura eseguita con un analizzatore logico direttamente dal pin di uscita della scheda, nello specifico dalla porta 1 del connetore Pmod JA. Si possono notare che è una sequenza di 72 bit seguita e preceduta da due segnali di reset. Ogni sequenza di 24 bit corrisponde ad un colore “puro” e sono inviati in ordine, nella striscia led si illumineranno 3 gruppi di tre led, in ordine blu, rossi e verdi.
Interfacciarsi con i led ci ha portato non pochi problemi perchè nonostante il codice relativamente compatto gli errori si possono celare ovunque, da chip non

genuini a datasheet errati a errori di codice, ma in questo caso la complicazione si è rivelata essere una porta non funzionante nella scheda che quindi inviava spazzatura ai led che ovviamente non funzionavano. A seguito una lettura effettuata direttamente dalla suddetta porta (connettore Pmod JD) utilizzando lo stesso codice della letture corretta.

I led sono alimentati con un alimentatore esterno da 12V e i ground sono tutti collegati in comune per garantire stabilità