UNIVERSITÀ DEGLI STUDI DI PADOVA

Misuratore di frequenza

Tesi di laurea triennale in ingegneria dell'Informazione

Antonio Rizzo
Matricola: 610143
28/09/2012

Relatore: Daniele Vogrig

Realizzazione di un misuratore di frequenze digitali tramite scheda FPGA Xilinx.

Indice

Sommario

Tramite l'utilizzo di una scheda montante un FPGA è stato realizzato un progetto che ha lo scopo di calcolare e visualizzare sui display a 7 segmenti della board la frequenza di un'onda digitale in ingresso alla scheda.

Il clock che viene analizzato viene fatto entrare in ingresso da uno dei pin di segnale di I/O presenti sulla board.

L'obbiettivo è quello di riuscire a minimizzare errori dovuti a ritardi e componentistica tramite blocchi semplici dal punto di vista della logica interna.

Tali blocchi sono poi collegati tra loro in modo tale da permettere il funzionamento complessivo del progetto.

La scheda utilizzata monta un FPGA di marca ***Xilinx*** [5] e la PCB è stata realizzata dalla ***Diligent*** [6]. Il software in uso è provveduto dalla ***Xilinx*** [5] ed in particolare è stato usato il pacchetto "***ISE Design Suite V14.2***" [5].

Capitolo 1

Introduzione di base all'FPGA [1]

Cos'è un FPGA

Un FPGA (*Field Programmable Gate Array - "Matrice di porte logiche programmabili sul campo")* è un circuito digitale integrato (IC) contenente blocchi logici programmabili e interconnessioni configurabili colleganti questi blocchi. Questi dispositivi consentono l'implementazione di un numero vasto di dispositivi, a seconda di come viene programmato un FPGA è infatti possibile riprodurre diverse tipologie di circuiti logici svolgenti molteplici funzioni. Il nome *"Field Programmable" – "Programmabile sul campo"* è riferito al fatto che la programmazione di questi dispositivi avviene a cura dell'utente che ne farà uso e non è configurata durante la creazione del chip. Un FPGA può essere configurato durante l'applicazione ad uno specifico circuito o può anche essere riprogrammato in seguito ad una precedente installazione in un sistema che offre la possibilità di molteplici utilizzi (quale ad esempio la scheda usata per questo progetto). Se un dispositivo è in grado di essere programmato mentre si trova su una scheda, si parla di dispositivo *"In System Programmable" – "Programmabile nel sistema"* o ISP.

FPGAs fecero la loro comparsa alla metà degli anni '80. All'inizio venivano perlopiù usate per implementare macchine di stato, o processi abbastanza limitati. In seguito nei primi anni '90 le dimensioni e le capacità di calcolo implementati su un FPGA iniziarono ad aumentare, permettendone un largo utilizzo nel campo delle telecomunicazioni dove si rendeva necessario elaborare grossi blocchi di dati. Verso gli ultimi anni '90 vennero poi utilizzati anche a livello industriale per le più svariate applicazioni.

La tecnologia offerta da un FPGA viene usata per la realizzazione di prototipi di chip ASIC o per verificare il funzionamento di un nuovo algoritmo a livello hardware. Punto di forza di schede basate su FPGA è la possibilità di implementare hardware personalizzato a bassi costi e di immetterlo sul mercato in tempi brevi. Inoltre grazie alle prestazioni elevate raggiunte (questi dispositivi ormai integrano spesso

microprocessori o interfacce ad alta velocità per I/O) oggi un FPGA permette la realizzazione di pressoché qualunque circuito per qualunque tipo di applicazione.

Tecnologia alla base

Il primo dispositivo FPGA fu reso disponibile da ***Xilinx*** [5] nel 1984, i primi FPGA erano basati sulla tecnologia CMOS e usavano SRAM come celle di memoria per la configurazione delle connessioni. Sebbene si parli di una tecnologia agli esordi e pertanto il numero di porte logiche fosse ridotto, essa può essere usata per spiegare bene come funzioni un FPGA in quanto contiene tutti gli aspetti principali alla base dell'architettura che tutt'oggi viene usata.

Un FPGA è basata sull'uso di blocchi logici programmabili, ognuno dei quali contiene principalmente una LUT (*"LookUp Table"* – *"Tabella di riferimento"*), un registro che può fungere sia da flip-flop che da latch ed infine un multiplexer.

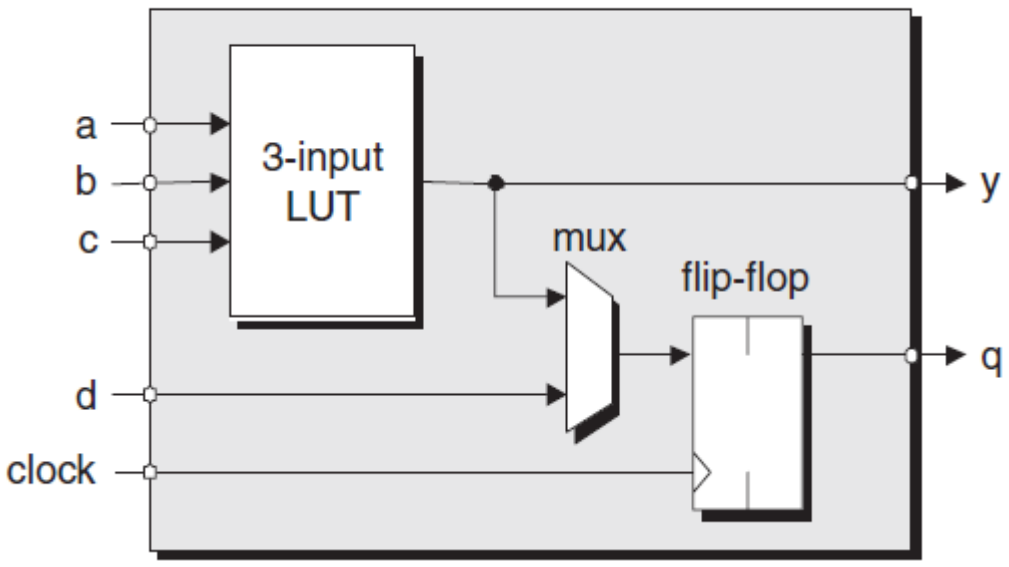


Figura 1-1

Un FPGA contiene un gran numero di questi blocchi i quali, a seconda di come viene programmata la SRAM, svolgono diverse funzioni. I registri possono essere usati come flip-flop a campionamento, sia su rami in salita

che in discesa, che come latch sia positivo o negativo. Il multiplexer serve per decidere qual è l'ingresso che andrà ad alimentare il flip-flop. Nel blocco logico rappresentato in figura, ogni LUT può realizzare una qualsiasi funzione logica a 3 ingressi e 1 uscita. La caratteristica di questi blocchi logiche sono di avere solitamente pochi I/O a differenza delle CPLD[1].

Nella *figura 1-2* viene mostrato come questi blocchi logici interagiscono tra loro tramite l'uso di un "mare " di interconnessioni programmabili.

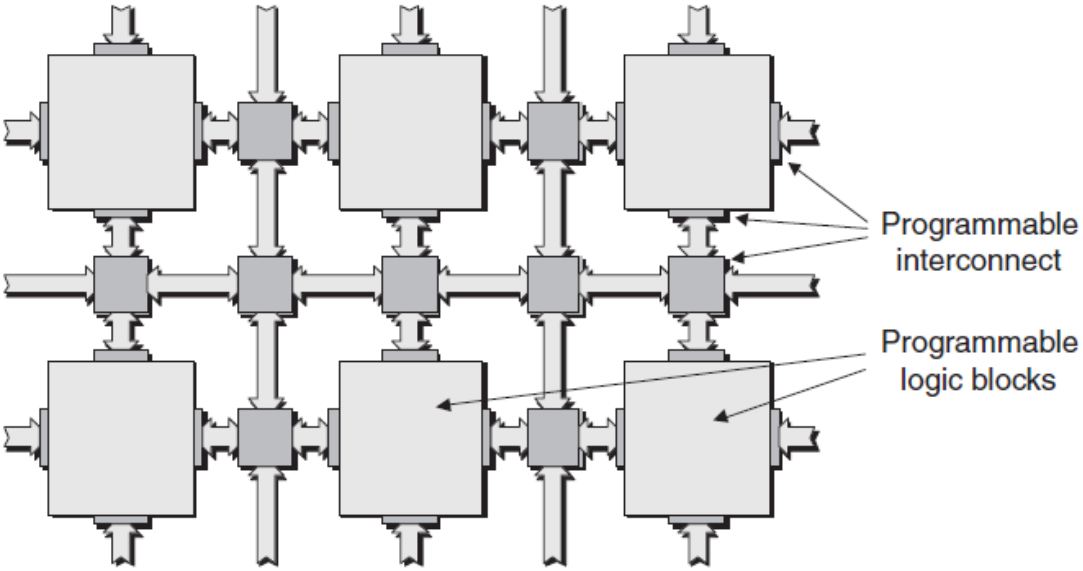


Figura 1-2

Oltre all'uso di queste interconnessioni solitamente sono predisposte anche connessioni ad alta velocità utili soprattutto a trasportare segnali da una parte all'altra del chip senza dover attraversare troppi multiplexer o elementi di scambio. Tramite un'appropriata configurazione è possibile indirizzare pin di I/O in modo da collegarli direttamente ai blocchi logici, così com'è possibile collegare l'uscita di un blocco logico all'ingresso di un'altro.

Nella *figura 1-3* è riportato l'esempio schematico di come un LUT implementi una data funzione logica. Il contenuto delle celle SRAM determina la funzione logica che viene realizzata nella LUT, riscrivendone il contenuto è quindi possibile cambiare la funzione logica.

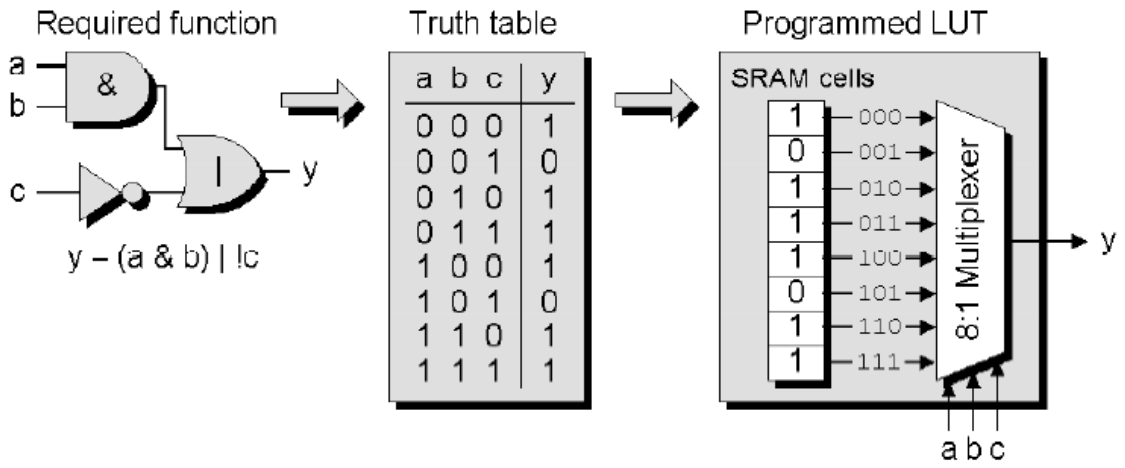


Figura 1-3

FPGA: una visione più moderna

Nel paragrafo precedente è stato descritto il funzionamento del blocco logico di base di un FPGA e come questi blocchi vengano connessi.

Negli FPGA recenti spesso sono inclusi nel package anche altri moduli come sommatori, moltiplicatori, unità MAC (*"Multiply and Accumulate"* – *"Moltiplica e Accumula"*), memorie RAM, nuclei di microprocessori o interfacce veloci di I/O. Questi componenti integrati negli FPGA non sono riprogrammabili (se non le connessioni a cui sono collegati) e svolgono funzioni specifiche, permettendo di alleggerire il carico delle funzioni logiche a cui dovranno andare ad assolvere i blocchi logici riprogrammabili.

Nell'esempio in figura 1-4 è stata riportata l'architettura di un FPGA della Xilinx, la Virtex-4 [2].

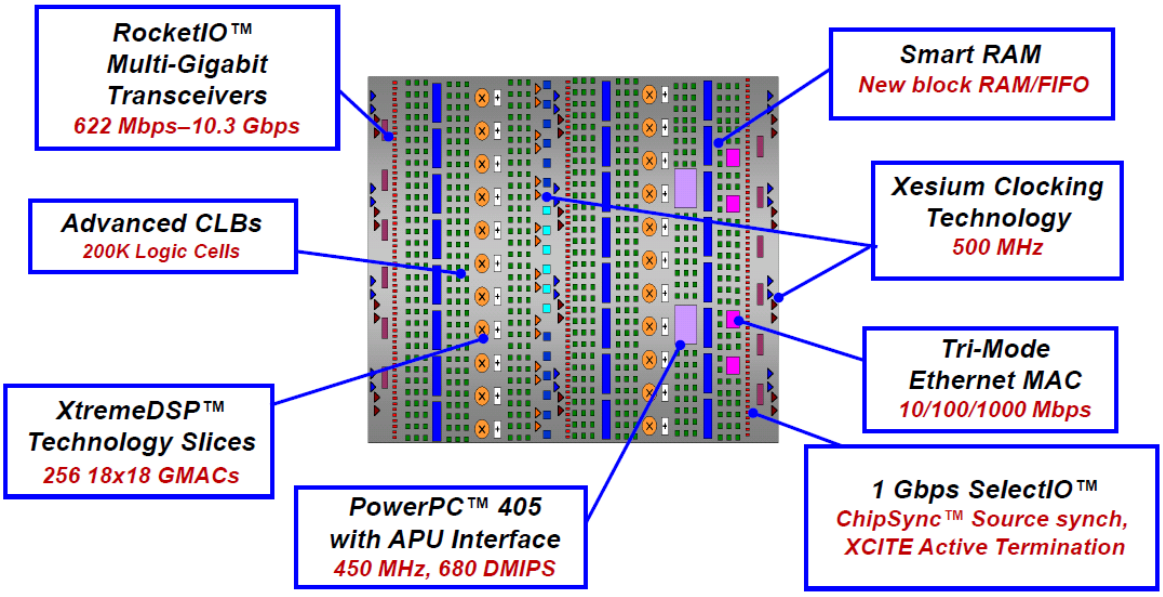


Figura 1-4

Linguaggio HDL

Il linguaggio HDL (*"Hardware Description Language"* – *"Linguaggio Descrittivo per Componenti Fisiche"*) fu introdotto a partire dalla fine degli anni '80 quando la visualizzazione, la comprensione e la correzione dei progetti iniziò a divenire inefficiente e inapplicabile, se basata solo su schemi delle porte logiche, a causa della crescente complessità dei progetti da realizzare.

L'idea del linguaggio HDL è quella di riuscire a descrivere l'Hardware, inteso solo come circuiti e componenti degli IC, tramite un linguaggio descrittivo. Agli albori del linguaggio HDL, questo era usato sia per descrivere circuiti analogici che digitali tuttavia nella sezione successiva si farà riferimento solo a descrizione di circuiti digitali. Nella *figura 1-5* sono rappresentati i diversi livelli con cui è possibile descrivere un circuito digitale tramite linguaggio HDL.

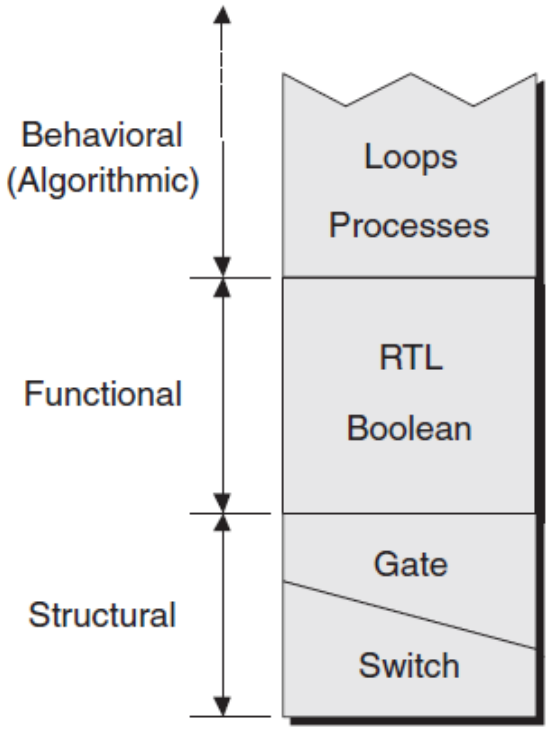


Figura 1-5

Il livello più basso di astrazione per un linguaggio HDL è rappresentato dallo *"Switch Level" – "Livello degli interruttori"*; a questo livello il linguaggio descrive il circuito come una rete di interruttori costituiti da transistor. Leggermente più su troviamo il *"Gate Level" – "Livello Porte"* in cui il linguaggio descrive una rete costituita da porte logiche. Entrambi questi livelli fanno parte di una descrizione strutturale di un circuito.

Il livello definito come *"Funzional" – "Funzionale"* è costituito da quei linguaggi HDL che supportano la rappresentazione della logica del circuito tramite una funzione booleana o una rappresentazione RTL (*"Register Transfer Level" – "Livello Connessioni tra Registri"*). A tale livello è possibile implementare connessioni tra registri sincrone o asincrone e funzioni logiche velocemente, senza soffermarsi nei dettagli implementativi delle porte logiche.

L'ultimo livello è quello *"Behavioral" – "Comportamentale"* nel quale è possibile descrivere il comportamento che deve assumere un determinato circuito usando costruttori astratti come cicli e processi. A questo livello è inoltre possibile utilizzare funzioni come sommatori e moltiplicatori in un algoritmo senza preoccuparsi di come questi siano implementati.

Il linguaggio HDL può essere usato per creare progetti sia per FPGA che per ASIC (*"Application Specific Integrated Circuit" – "Circuiti Integrati per Applicazioni Specifiche"*) in quanto descrive perfettamente, nelle diverse forme viste ai vari livelli, il circuito che si sta progettando.

Un componente ASIC è un circuito integrato disegnato per poter essere usato in una specifica applicazione. Tale circuito viene creato in fabbrica sotto le specifiche del richiedente e non ha la possibilità di essere programmato dall'utente finale.

Durante la realizzazione di un ASIC è possibile specificare qualsiasi parametro dei transistor al suo interno in modo tale da aumentarne le prestazioni che sono richieste dal cliente, tuttavia una volta implementato e lanciato in produzione, il costo per effettuare modifiche al disegno del progetto è enorme.

Per questo un FPGA è la soluzione ideale nel caso il progetto di un ASIC non sia conveniente, per via della produzione limitata che si intende fare.

Progettazione su FPGA

La fase di progettazione di un FPGA è molto simile a quella di un ASIC e fa uso pertanto di un linguaggio HDL e di strumenti CAD, la differenza principale si rileva a livello di implementazione fisica dove mentre per un ASIC si devono piazzare le celle della libreria collegandole "ad hoc" con connessioni che poi resteranno fisse, per un FPGA il piazzamento delle celle è dovuto ad una allocazione delle risorse disponibile nel componente in modo ottimale con relativa griglia di connessioni. Tale disposizione in un FPGA può essere modificata in modo da ottimizzare i tempi o lo spazio o altri fattori da considerare in fase di progettazione.

Nello schema a blocchi riportato in *figura 1-6* viene rappresentata la sequenza temporale e le varie fasi a cui va sottoposto il progetto prima di una effettiva implementazione sulla scheda.

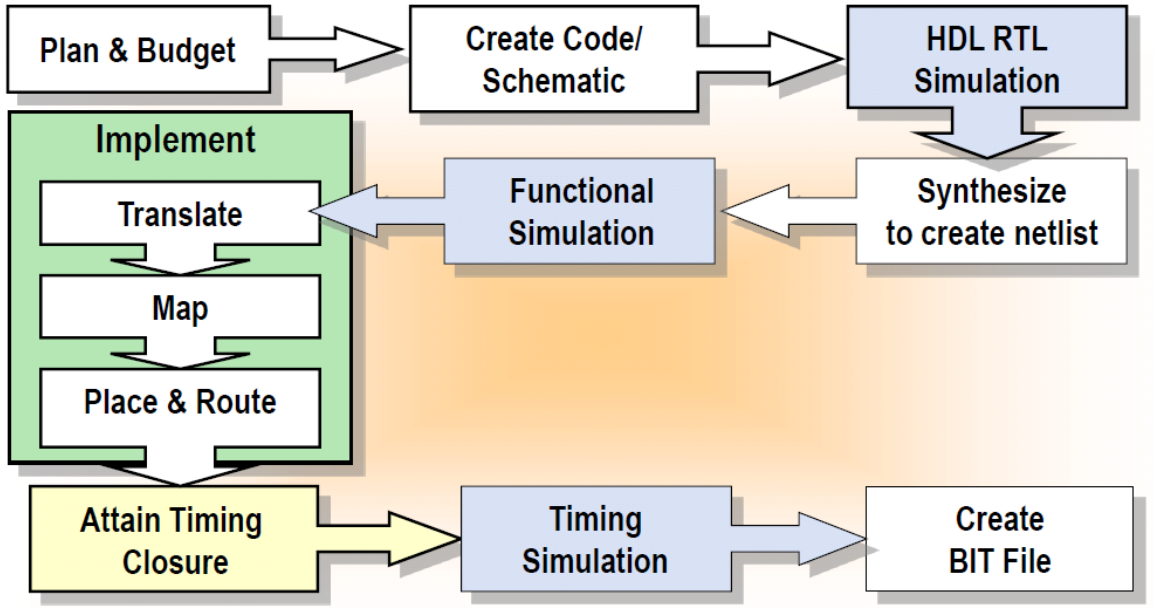


Figura 1-6

Partendo dalle specifiche che sono richieste e dal budget messo a disposizione, si decide che tipo di approccio avere e la complessità del progetto che si vuole creare. Poi si crea uno schema a blocchi, per capire quali sono le componenti principali che si dovranno andare a creare per implementare il progetto, e si scrive il codice che corrisponde ai vari blocchi connettendoli infine tra loro. Terminata la fase progettuale si passa alla sintesi e alla creazione di una prima rete di connessioni. Si simula quindi la logica del progetto per verificare che il comportamento che esso assume sia

corretto e si passa all'implementazione. L'implementazione viene fatta tenendo conto di eventuali parametri e costanti di tempo vincolate alla scheda utilizzata. Ora si ha un modello molto simile a quella che diventerà la reale implementazione sulla scheda. Con tale modello si può ora effettuare una simulazione considerando i ritardi che introduce l'hardware della scheda. Se tutte le specifiche sono rispettate si può quindi generare il file che servirà a programmare la scheda.

Al termine della fase di progetto si otterrà un file contenente uno *"stream"* - *"flusso"* di bit che andrà poi caricato nella SRAM o nell'EEPROM della scheda e che servirà a programmare tutte le strutture contenute nel FPGA.

FPGA Xilinx [1] [2] [6]

Descrizione di un FPGA Xilinx

Tutte le schede FPGA della ***Xilinx*** [5] presentano una struttura di base pressoché costante comprendente *"slices"* – *"fette"* raggruppate in CLB (*"Configurable Logic Block"* – *"Blocchi Logici Configurabili"*), IOB ("*Input-Output Block"* – *"Blocchi di Ingresso-Uscita"*), interconnessioni riprogrammabili, risorse di memoria, moltiplicatori, buffer di clock globali e logica per boundary scan. Vediamo ora un po' più approfonditamente i blocchi logici configurabili che sono il cuore di un FPGA.

Un CLB è composto da 2 slices (o più a seconda del FPGA) ognuna delle quali ha 2 celle logiche LC (*"Logic Cell"* – *"Celle Logiche"*) come quella rappresentata in *figura 1-7*.

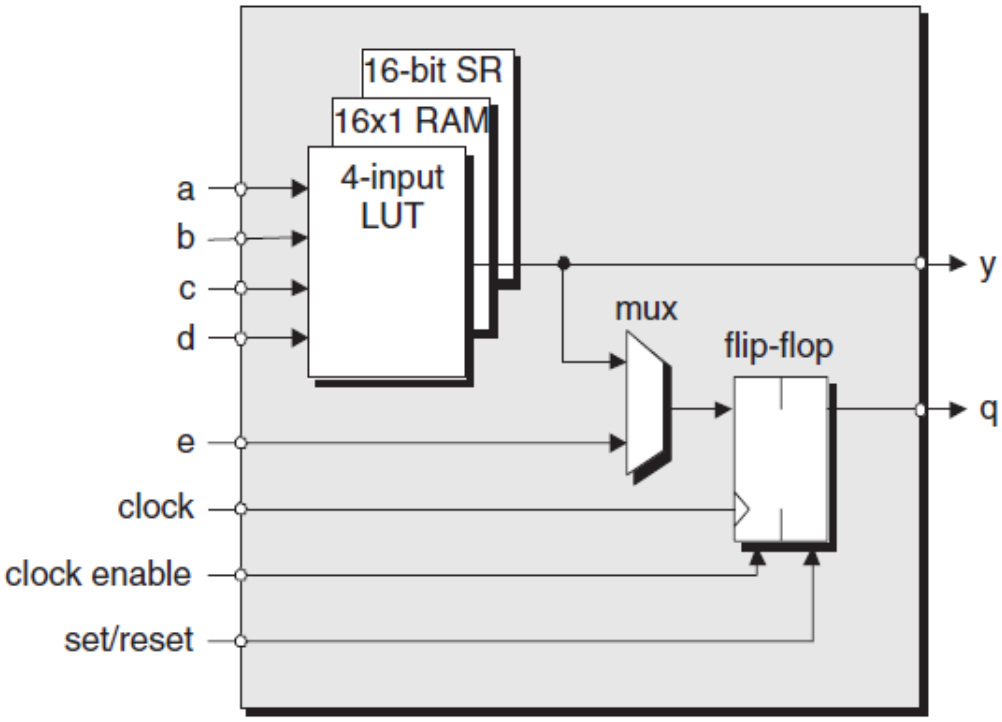


Figura 1-7

Una LC contiene come già visto una LUT, una logica di riporto e un registro.

Ogni slice ha 4 uscite, delle quali 2 sincrone e 2 asincrone, e ha 2 driver tri-state (BUFT) associati a ogni CLB i quali sono accessibili da tutte le 16 (o 8 a seconda di quante slice sono raggruppate in un CLB) uscite della CLB. Inoltre all'interno della CLB sono spesso presenti 2 catene indipendenti di "carry" – "riporto" le quali, associate a collegamenti dedicati, consentono di incrementare le performance di funzioni logiche come contatori o sommatori. Nella *figura 1-8* ne è riportato uno schema descrittivo.

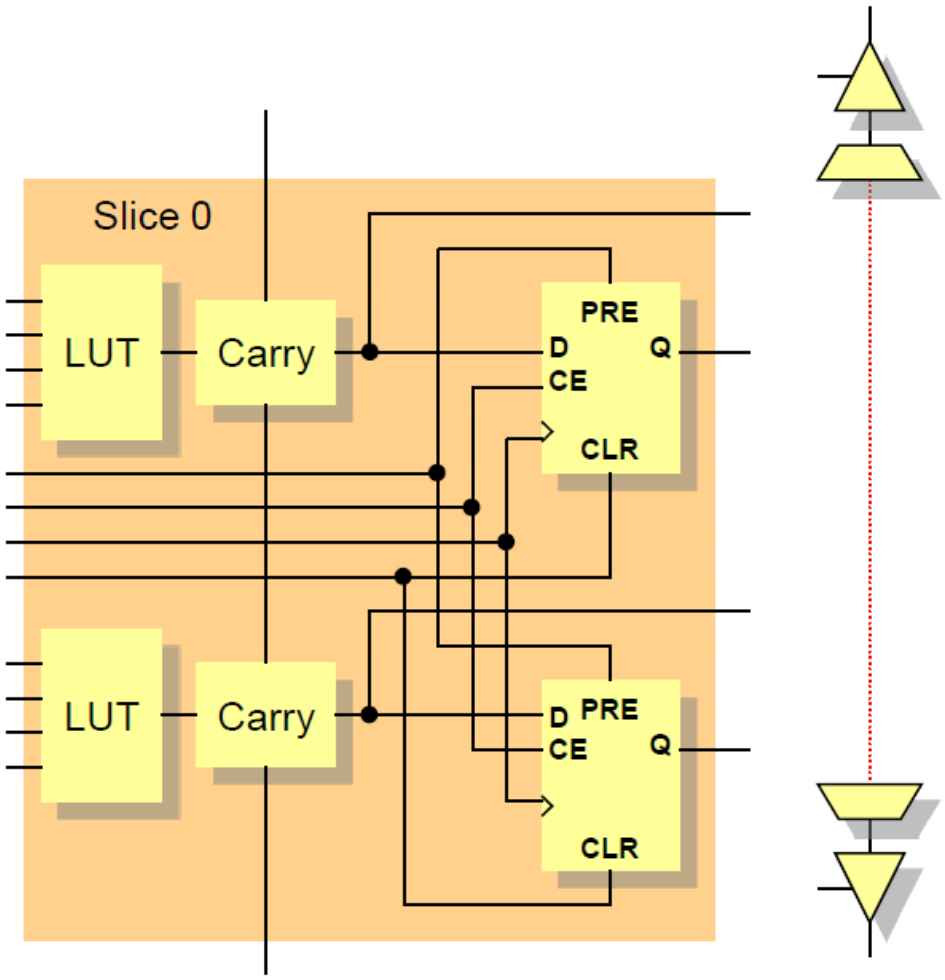


Figura 1-8

Alcune LUT possono essere usate come una RAM da 16-bit o come uno "*shif-register*" – *"registro a scorrimento"* da 16-bit. È importante notare che per combinare le funzioni delle 2 LC, ogni slice contiene un multiplexer e uno XOR per effettuare l'addizione. Infine per poter pilotare i bus all'interno del chip sono presenti dei driver tri-state o BUFT.

Specifiche della scheda Diligent e software usato

Dopo questa panoramica veloce sul funzionamento di un FPGA, passiamo ora ad analizzare nel dettaglio la scheda Diligent usata per la realizzazione di questo progetto.

La scheda chiamata "***NEXYX 3***" [6] è prodotta dalla "*Diligent*" [6] e monta un FPGA "Xilinx Spartan 6 - XC6SLX16" [5]. Le caratteristiche tecniche del FPGA montato in questa scheda sono:

- 2278 slices ognuna delle quali contenenti 4-input LUTs e 8 FlipFlops;
- 576Kbits di RAM veloce;
- 2 celle di clock (4 DCM-Digital Clock Manager e 2 PLL-Phase Locked Loop);
- 32 DSP-Digital Signal-Processing slices;
- <500MHz velocità di clock;

Nella *figura 1- 9* sono riportate tutte le proprietà della scheda con i diversi dispositivi I/O forniti come ingressi o uscite al FPGA.

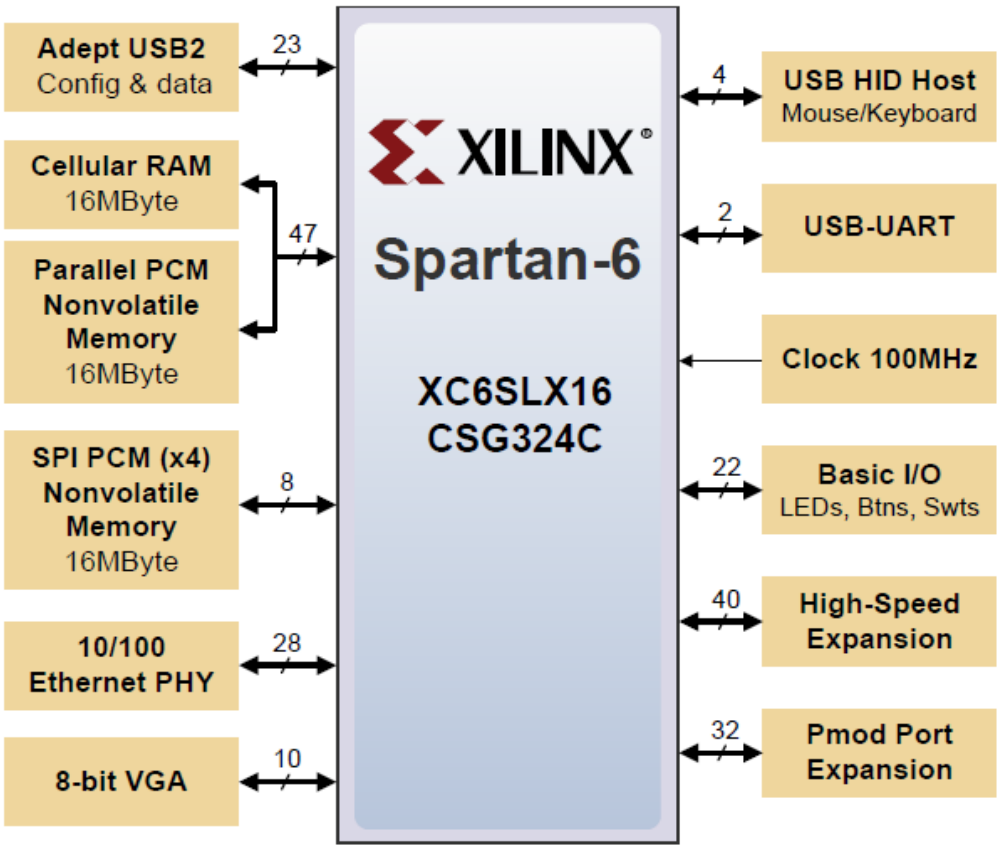


Figura 1-9

Tra questi sono messi in evidenza i dispositivi usati nel progetto che sono:

- 100MHz oscillatore CMOS;
- Basic I/O comprende le uscite collegate agli 8 LED, agli 8 SWITCH, ai 6 pulsanti e ai 4-display a 7 segmenti;
- High-Speed Expansion e Pmod Port Expansion forniscono 72 I/O connessioni ai pannelli di connessione esterna;

Nella *figura 1-10* sono riportate le connessioni I/O di base che sono state usate nel progetto e i relativi pin associati a tali I/O.

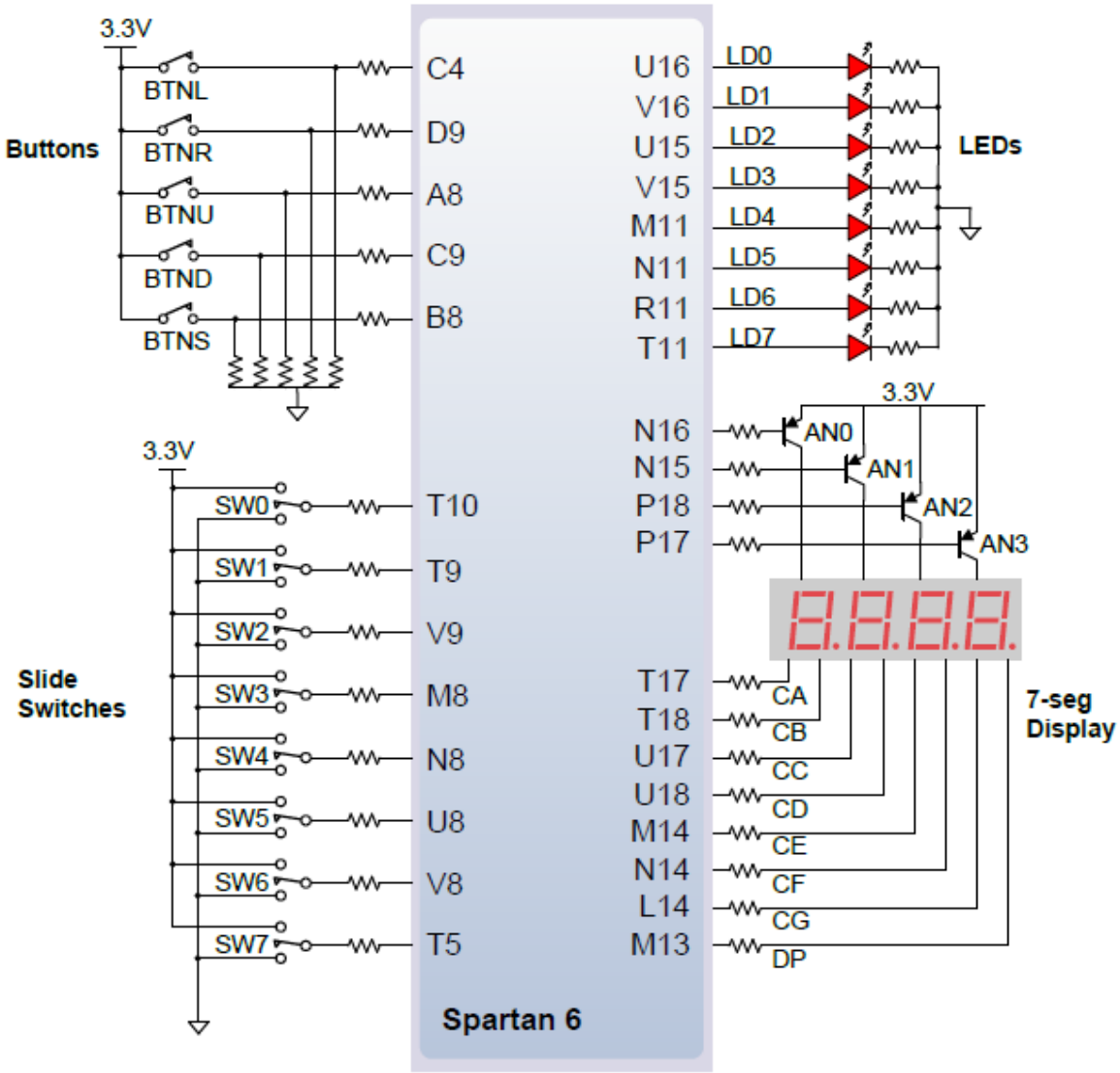


Figura 1-10

Nella *figura 1-11* è riportato un dettaglio di come è strutturato il display a 7-segmenti e sono riportati i segmenti che devono essere accesi per poter ottenere una determinata cifra. È possibile notare il collegamento ad anodo comune per ogni segmento ma individuale per ogni display e catodo individuale per segmento ma comune per tutti i 4 display.

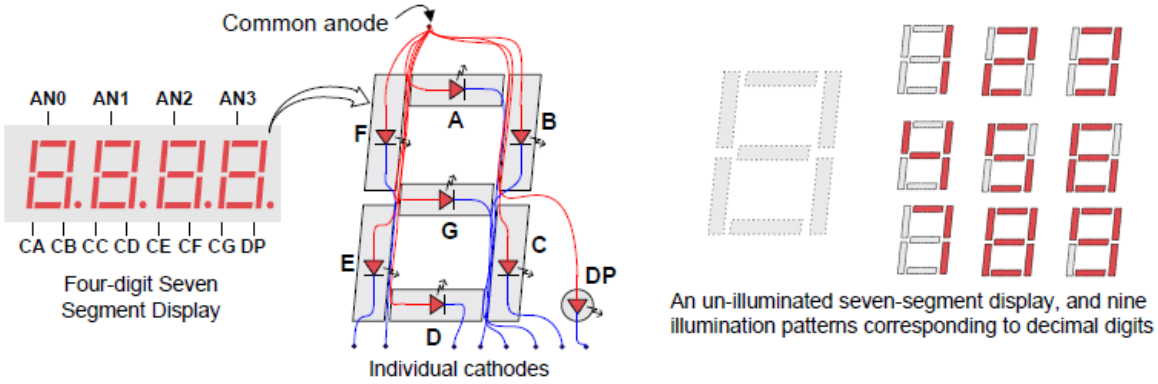


Figura 1-11

I banchi PMOD costituiscono i 32 ingressi dei quali solo uno è stato usato per leggere il clock esterno. Inoltre possono essere usati per fornire alimentazione a schede esterne, difatti ogni PMOD contiene 2 piedini di alimentazione VCC e 2 collegati a massa GND. La *figura 1-12* riporta lo schema dei banchi PMOD e i PINs associati.

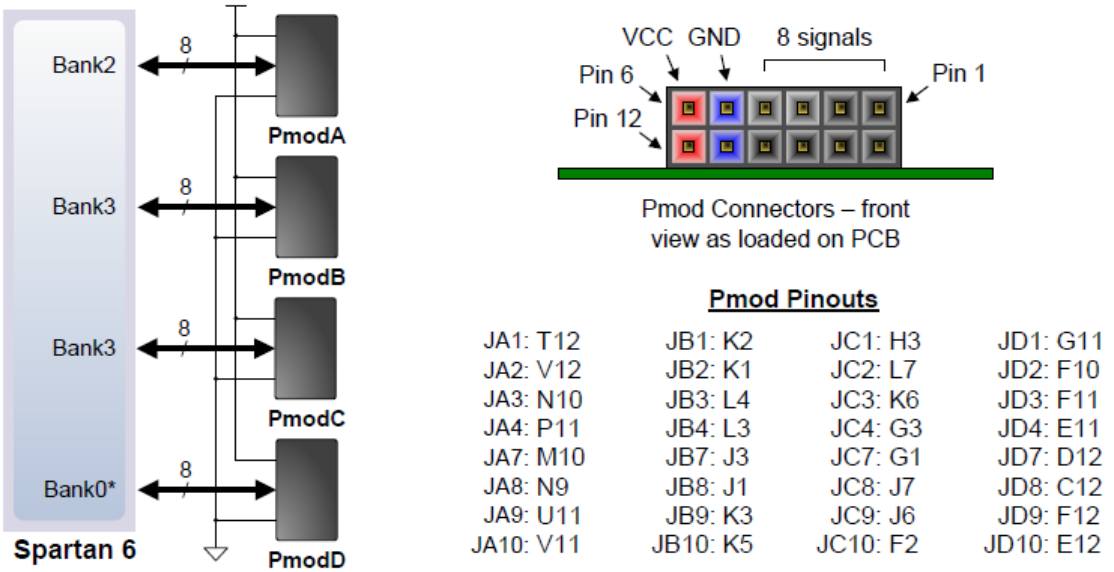


Figura 1-12

Nella *figura 1-13* è riportata un'immagine della board "NEXYS 3" usata nel progetto [6].

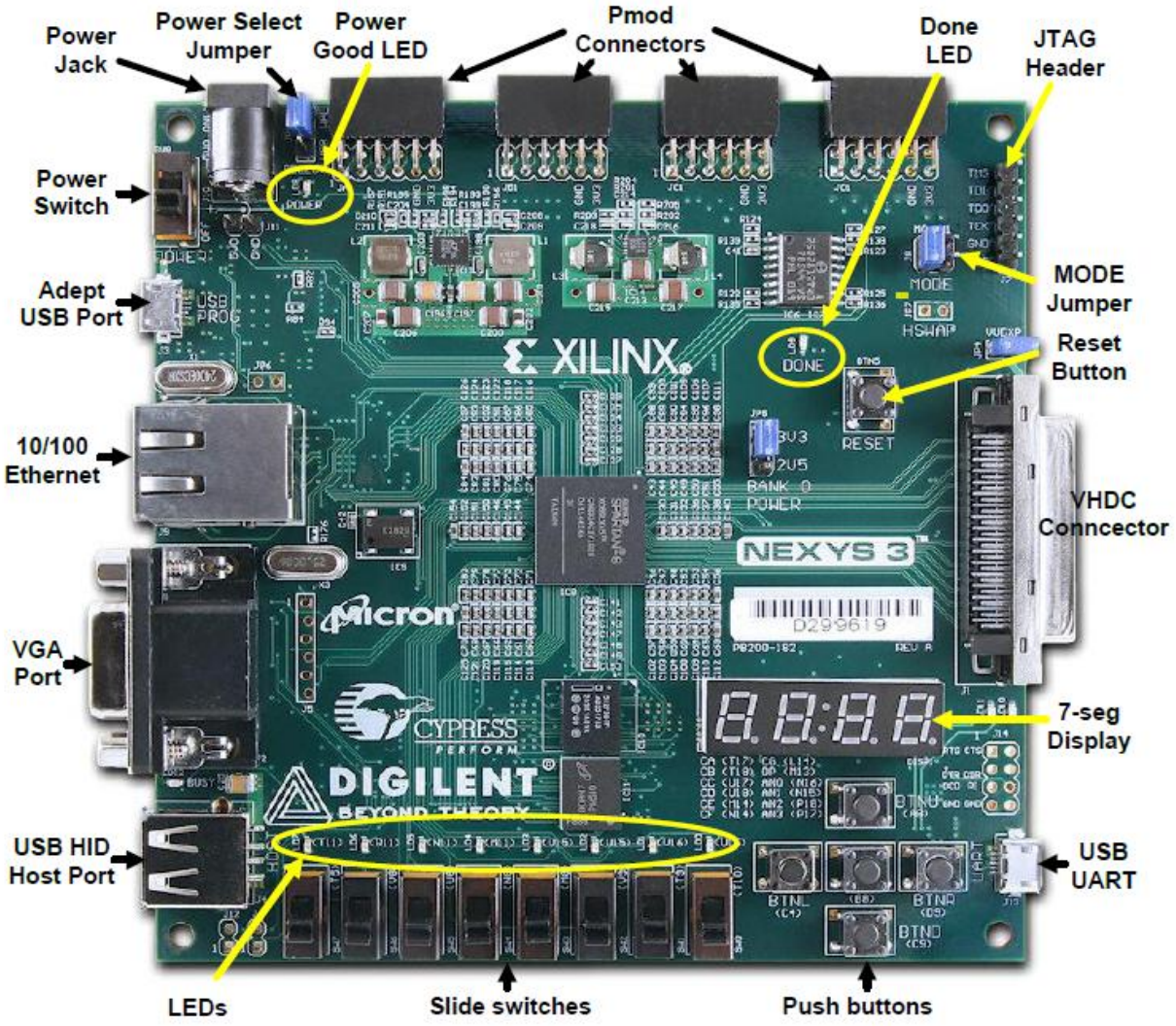


Figura 1-13

Il software usato durante la progettazione ed implementazione è fornito da Xilinx ed si chiama "***ISE Design Suites***" [5], la versione usata è la 14.2.

Capitolo 2

Idea alla base del progetto

Alla base dell'implementazione di questo misuratore di frequenza si trova un'idea molto semplice. La misurazione della frequenza di un clock esterno avviene semplicemente tramite il conteggio del numero di oscillazioni effettuate dal clock interno in un intero periodo del clock da analizzare.

Tramite la proporzione sottostante tra le frequenze viene poi visualizzato il risultato sui display a 7 segmenti.

$$\frac{f_{ext}}{n_{ext}} = \frac{f_{int}}{n_{int}}$$

f_{ext} = frequenza da calcolare.

n_{ext} = numero periodi contati clock esterno.

f_{int} = frequenza dell'oscillatore presente sulla scheda.

n_{int} = numero periodi contati clock interno.

Nel progetto ho deciso di considerare solo 1 periodo del clock esterno e all'interno di questa calcolare il numero di oscillazioni fatte dal clock interno. Nella *figura 2-1* è possibile visualizzare le forme d'onda di due clock simulati.

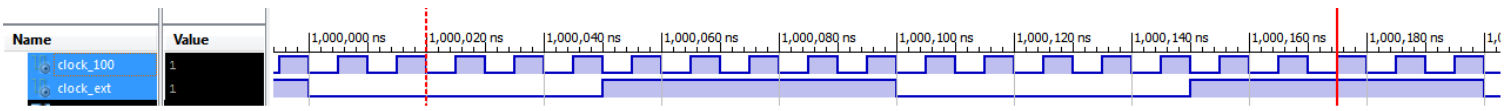


Figura 2-1

Il clock superiore rappresenta il clock interno a 100MHz, quello sottostante invece è il clock esterno da misurare.

Il misuratore di frequenza conta il numero n_{int} di periodi del clock interno in un periodo del clock esterno.

La scelta dell'utilizzo di un intero periodo di clock esterno, anziché del semi-periodo, permette di effettuare il calcolo della frequenza anche per clock asimmetrici come quello mostrato in *figura 2-2*.

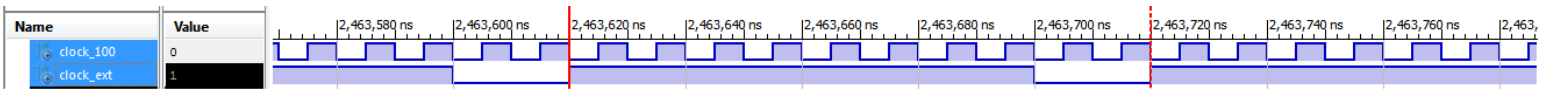


Figura 2-2

Specifiche di utilizzo

A causa dell'implementazione molto semplice, questo progetto presenta delle forti limitazioni nel suo utilizzo. Difatti nel caso di frequenze da analizzare molto alte, che si avvicinano come ordine di grandezza alla frequenza interna (quando si va oltre i 10MHz), il conteggio delle oscillazioni interne diventa impreciso e può portare a errori di misurazioni anche del 50%. Nella *figura 2-3* è possibile capire quanto risulti difficile ed impreciso il calcolo di frequenze elevate.

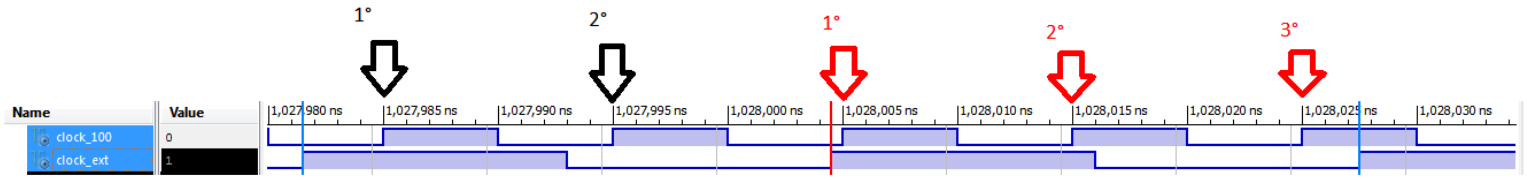


Figura 2-3

Nell'esempio la frequenza esterna (quella sotto) è di 43,478 MHz mentre l'interna è sempre di 100MHz (quella sopra). Come è possibile vedere, poiché il contatore passa al valore successivo all'apparire di un ramo in salita del clock interno durante un periodo di quello esterno, nel caso di frequenze molto elevate il contatore può ottenere risultati diversi a seconda della misurazione. Nel primo periodo infatti il contatore si fermerà a 2 prima di resettarsi, nel secondo periodo invece si fermerà a 3. Questo porta ad un calcolo di 2 frequenze che sono rispettivamente 50MHz e

33,33MHz. Si ha quindi un errore, in questo caso particolare, del 23% sul valore reale della misura, il che risulta ovviamente eccessivo!

Pertanto i limiti dovuti alla struttura del progetto entro i quali è bene attenersi per avere una buona misurazione della frequenza in ingresso sono delineati tra 1Hz e 1MHz. In tal modo l'errore sulla misura sarà al massimo dell'1%.

Capitolo 3

Modello a blocchi del progetto [3]

Nel seguente capitolo verrà presentata una visione generale a blocchi del progetto e in seguito verranno spiegati blocco per blocco le funzionalità dei diversi componenti e come sono stati costruiti. Nei blocchi presentati sono visibili solo Input/Output del blocco e come questi sono collegati tra loro nell'insieme.

Visione d'insieme

Nella *figura 3-1* è possibile visionare come si presenta il progetto sotto forma di *"black-box" – "scatola nera"*.

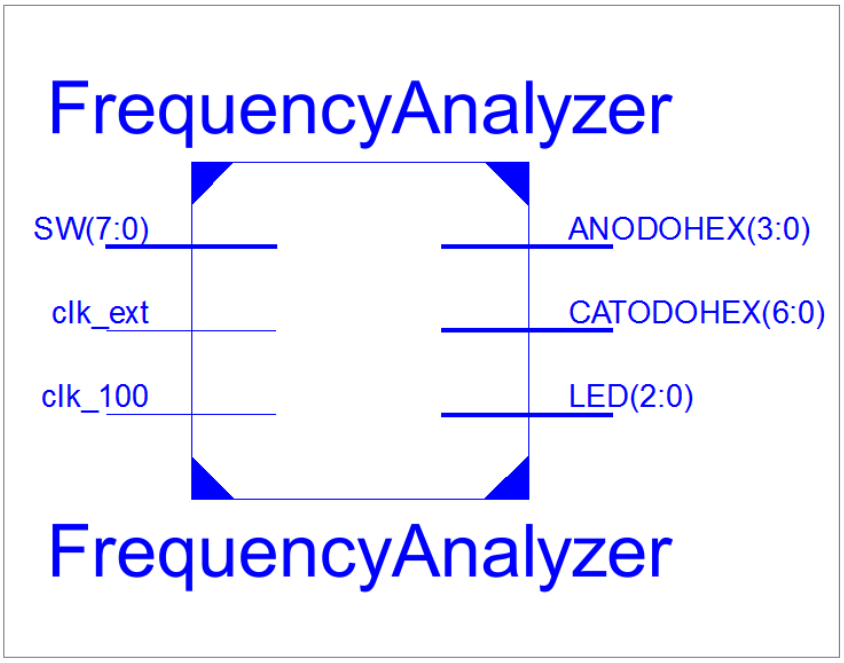


Figura 3-1

Analisi Input/Output :

- `Clk_ext, clk_100` : rappresentano rispettivamente il clock da calcolare, quello esterno che poi andrà a collegarsi ad un ingresso di I/O della scheda, e il clock interno usato come riferimento. In questo caso il clock interno si chiama clk_100 in quanto è un clock a 100 MHz (quella generata dalla scheda).

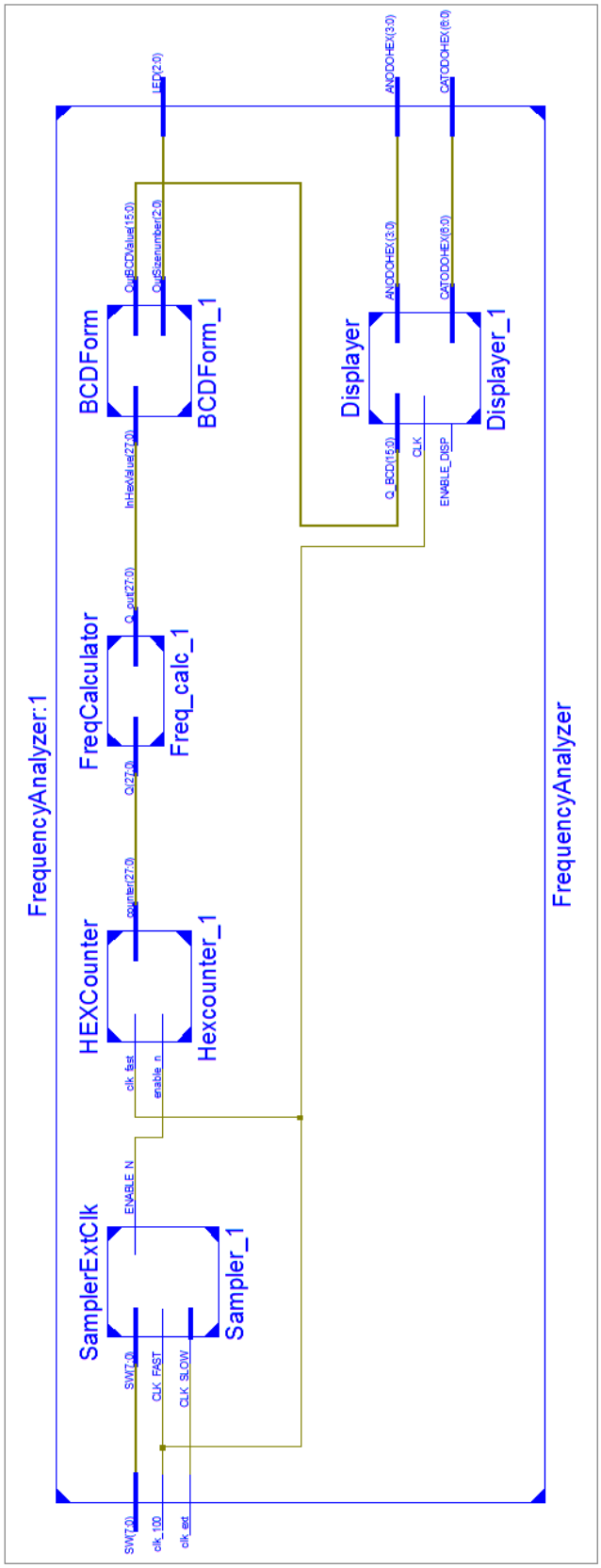
- `SW(7:0)`: rappresentano gli ingressi contenenti il valore da raggiungere da parte del contatore interno prima di aggiornare il risultato visualizzato sui display a 7-segmenti. Tale valore è espresso come numero (`SW` da 5 a 0) e unità di misura (`SW` da 7 a 6).
- `ANODOHEX, CATODOHEX`: sono i segnali che vanno a controllare il display a 7 segmenti della scheda. Sono poi collegati con i PINs dei display rispettivamente all'anodo dei led dei display ed al catodo del display.
- `LED`: questa bus contiene segnali collegati ai led della scheda, i led sono usati all'interno del progetto per permettere di visualizzare l'ordine di grandezza della misura riportata, cioè se la cifra visualizzata sul display sia in Hz, KHz o MHz. A seconda di quale di queste rappresenta avremo, a partire da destra, il primo led (Hz), il secondo (KHz) od il terzo (MHz) che si accenderanno.

Codice di implementazione

Il codice realizzante questo blocco è perlopiù costituito da inizializzazioni e dichiarazioni dei sottoblocchi di cui è composto. Al suo interno sono contenuti solo le definizioni blocchi da collegare e dei segnali di supporto per collegarli tra loro.

Andando leggermente più nel dettaglio vediamo ora come si presenta all'interno il progetto. Nello schema a blocchi in *figura 3-2* è possibile visualizzare come sono interconnessi tra loro i blocchi che compongono il misuratore di frequenza. Per una descrizione accurata dei blocchi, visualizzare i singoli blocchi nei paragrafi dedicati.

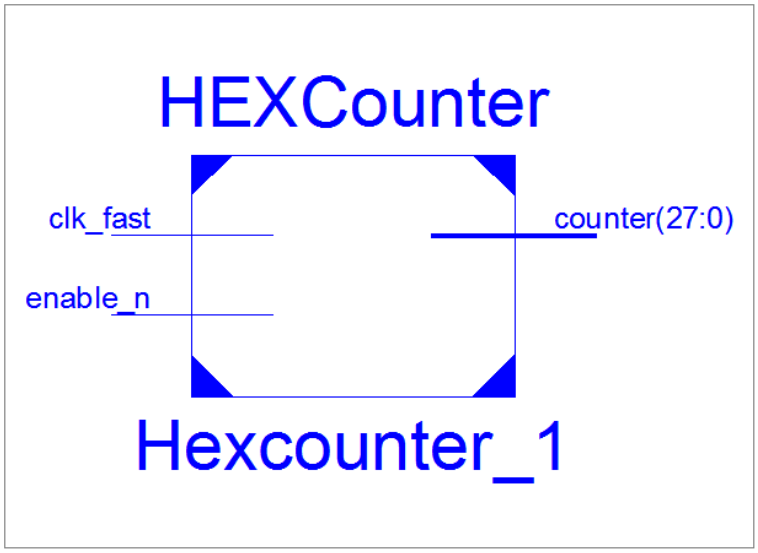
Figura 3-2



Passiamo dunque ad analizzare nel dettaglio i singoli blocchi.

Counter

Cuore del progetto è questo *"counter"* – *"contatore"* in binario a 28-bits. La scelta di un contatore a 28 bit è dovuta al fatto che il clock interno è di 100MHz e al funzionamento dell'analizzatore (che si basa sul conteggio del numero di oscillazioni del clock interno entro il periodo del clock esterno). Con 28-bits il massimo valore raggiungibile dal contatore è 268'435'456 che è più che sufficiente per calcolare frequenze dell'ordine dell'Hz. Il limite inferiore come già specificato è infatti di 1 Hz nel caso in cui il contatore raggiunga il valore di 100'000'000 oscillazioni del clock interno in un periodo di quella esterno.



Analisi Input/Output :

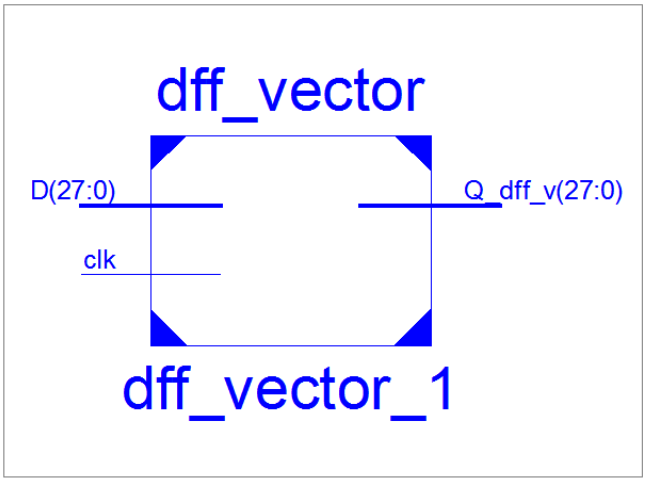
- `Clk_fast`: si tratta del clock usato per contare, come è possibile vedere dallo schema dei collegamenti tra blocchi, questo segnale di ingresso è collegato al clock interno della scheda (clk_100) a 100MHz.
- `Enable_n`: questo segnale *"active_low"* – *"attivo quando basso"* serve ad abilitare il conteggio. Quando viene abilitato il contatore parte a contare da 1 e termina quando `enable_n` torna ad un valore alto. Questo segnale viene prodotto da un altro blocco (vedi "*SamplerExternalClk*"). Una nota particolare deve essere fatta per notare che non esistono segnali di reset per azzerare il contatore ma questo avviene in automatico.
- `Counter` : è una bus in uscita che porta il segnale contenente il valore del contatore quando esso ha ultimato il conteggio. Lo standard usato è quello adottato per i numeri naturali.

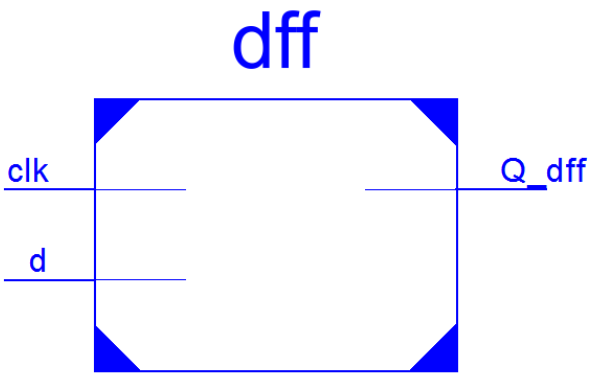
Codice di implementazione

Il codice riportato in *appendice 3-i* proviene dal blocco "*Counter"* e ne mostra la sezione più significativa, cioè come è stato implementato il reset automatico del contatore quando questo debba ripartire per calcolare il numero di oscillazioni del clock interno in un periodo del clock esterno. Il range del contatore viene impostato da una costante presente nel progetto principale; per questo progetto il contatore non serve superi l'ordine di grandezza di 10^8 (cioè avente una dimensione di 28-bit) corrispondente ad una frequenza esterna di 1Hz. Nel caso la frequenza esterna sia inferiore e quindi il contatore raggiunga valori maggiori il risultato mostrato risulta essere sbagliato in quanto dovuto ad un *"overflow"* – *"straripamento"* del contatore. Per evitare di saltare un clock nel conteggio, quando il contatore viene resettato parte da "1".

Sampler a FlipFlop

Questo sottoblocco presente all'interno di "*Counter*" costituisce un campionatore tramite un vettore D-FlipFlop a *"RisingEdge"* – *"ramo in salita"*, cioè il segnale è campionato quando il clock in ingresso presenta un ramo in salita. L'inserimento di questo blocco si è reso necessario in quanto l'uscita del contatore varia in base al segnale di `enable_n` e pertanto se collegata direttamente ai blocchi per la visualizzazione sul display comporterebbe una continua variazione del valore visualizzato. Per evitare ciò all'interno del blocco contatore è inserito questo blocco che consente di campionare il valore del contatore solo quando questo ha raggiunto il termine del conteggio e prima che questo riparta (e quindi venga azzerato).





Necessitando di un registro a 28-bit, il blocco è composto da 28 singoli D-FlipFlop.

Analisi Input/Output :

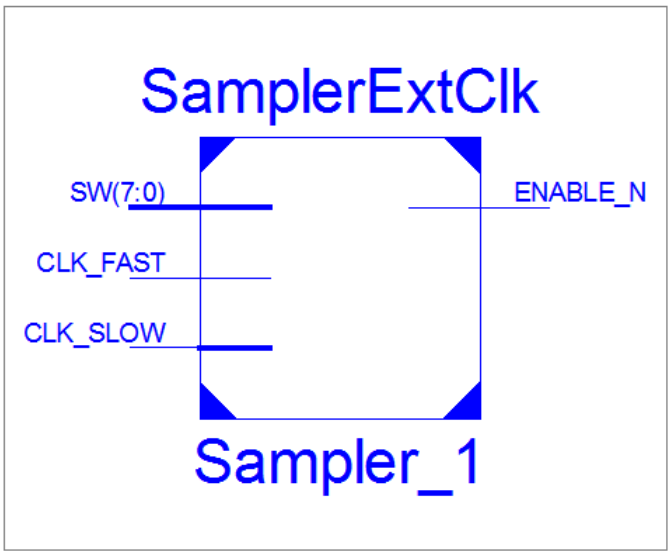
- `Clk`: è il segnale di clock, quando questo segnale si trova su un ramo in salita, il D-FlipFlop campiona l'ingresso.
- `D`, `d`: è il segnale in ingresso da campionare.
- `Q_dff`, `Q_dff_v`: è il segnale in uscita portante il valore campionato.

Codice di implementazione

Il codice utilizzato per il blocco unitario è molto semplice e descrive il funzionamento di un D-FlipFlop in modo comportamentale. Il blocco che crea un vettore D-FlipFlop invece contiene solamente un'inizializzazione tramite uso di un ciclo per collegare ingressi e uscite del vettore ad altrettanti sottoblocchi.

SamplerExternalClock

Il blocco "*SamplerExternalClock*" – "*Campionatore del clock esterno*" è un blocco molto importante in quanto permette di abilitare il conteggio per tutto un periodo del segnale in ingresso.



Questo blocco realizzato tramite una FSM ("*Finite State Machine*" – "*Macchina agli Stati Finiti*") abilita e disabilita il segnale `enable_n` ("*active_low*") a seconda del clock esterno e del valore imposto dagli `SW(7:0)` in ingresso alla scheda. Tramite questo blocco è possibile effettuare la misura della frequenza anche per segnali non simmetrici, cioè aventi all'interno del periodo una fase alta che dura più della fase bassa o viceversa, e con una frequenza di campionamento variabile.

Analisi Input/Output :

- `Clk_fast`: all'interno del campionatore viene usato solo al fine di permettere alla macchina di stato di aggiornare lo stato corrente con il successivo. Corrisponde al clock interno clk_100.
- `Clk_slow`: corrisponde al clock esterno ed è usato per determinare quando il periodo di clock inizia/termina. Inoltre su questo clock è basato il contatore che viene usato per determinare quando campionare il clock esterno.
- `SW(7:0)`: sono collegati direttamente agli ingressi di "FrequencyAnalyzer" e permettono di scegliere la frequenza di campionamento con in quale andare a valutare il valore della frequenza esterna.
- `Enable_n`: è l'unico segnale in uscita prodotto, molto importante in quanto pilota il contatore abilitandolo/disabilitandolo.

Codice di implementazione

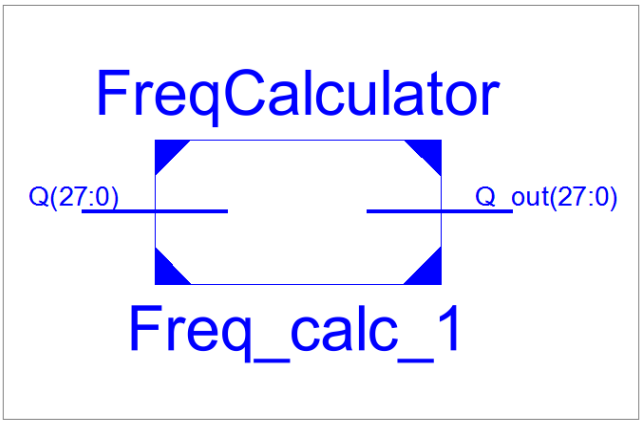
Per poter effettuare il campionamento sul periodo intero del clock esterno è stato necessario realizzare una macchina agli stati finiti (FSM) che disponesse di 3 stati, uno in cui far abilitare il conteggio, uno in cui far disabilitare il conteggio e prepararsi al reset ed uno di IDLE in cui si lascia disabilitato il conteggio per permettere di visualizzare il risultato. Nel codice in *appendice 3-ii* è riportata l'implementazione della FSM e l'aggiornamento dello stato.

Per determinare la frequenza con cui campionare il clock esterno sono stati usati gli 8 switch presenti sulla board. Di questi switch 2 (`SW7 e SW6`) sono usati per decidere l'ordine di grandezza del valore mentre gli altri 6 sono usati per determinare le cifre significative. Il valore `numbercicle` così calcolato costituisce il valore massimo a cui il contatore può arrivare prima di resettarsi.

È possibile osservare che mentre lo stato attuale è aggiornato ad ogni ramo di salita del clock interno (in questo caso chiamato `clk_fast`), lo stato successivo viene aggiornato solamente quando il clock esterno (`clk_slow`) cambia. In questo modo il segnale di uscita sarà sincrono con il clock interno. Anche il contatore dell'IDLE viene aggiornato quando si ha un ramo in salita nel clock esterno. Nei periodi di tempo in cui il contatore del clock interno è disabilitato si dà la possibilità di campionare il valore raggiunto e far visualizzare il risultato del conteggio.

Frequency Calculator

Il blocco qui presentato non è altro che un blocco contenente una serie di calcoli per poter ottenere, a partire dal valore campionato del contatore, il valore della frequenza del clock esterno.



L'ingresso come l'uscita sono composti da 28 bit, la stessa dimensione del contatore. Infatti poiché il limite superiore delle frequenze misurabili è di 100 MHz, non ha senso che l'uscita di questo blocco restituisca un valore maggiore, come ordine di grandezza, della massima frequenza misurabile.

Analisi Input/Output :

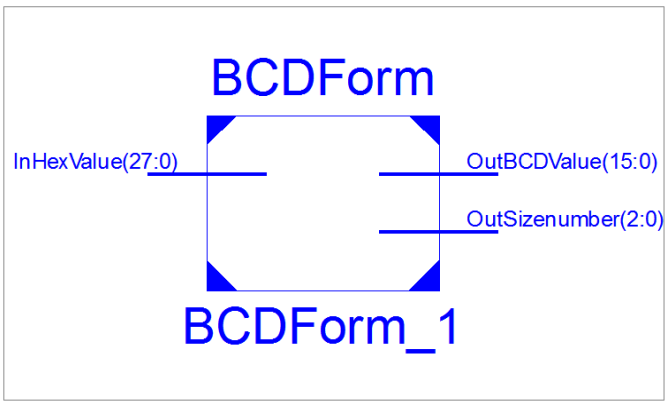
- `Q`: rappresenta l'ingresso del blocco, è collegato all'uscita del blocco *"Sampler a FlipFlop" – "Campionatore a FlipFlop"*.
- `Q_out`: costituisce l'uscita contenente il valore in binario rappresentante la frequenza del clock in ingresso.

Codice di implementazione

Il calcolo della frequenza viene effettuato in modo molto semplice facendo uso del pacchetto ***IEEE.NUMERIC_STD*** tramite il quale è possibile usare segnali ***unsigned*** e quindi effettuare calcoli come divisioni e moltiplicazioni. Il range dei segnali ***unsigned*** usati è lo stesso usato nel contatore in quanto il risultato che si otterrà ci aspettiamo abbia un valore inferiore a quello di riferimento della scheda.

BCDForm

"Binary-Coded Decimal" – "Decimale Codificato in Binario", questo blocco permette di convertire un numero binario in ingresso nel corrispondente numero rappresentato nella forma decimale, cioè dove ogni cifra viene rappresentata da 4-bit. Poiché inoltre le cifre visualizzabili sono solamente 4 sulla scheda, questo blocco provvede a mostrare le cifre più significative mostrandone l'ordine grandezza tramite l'uscita `OutSizenumber`.



Esempio: se l'ingresso è di 1,342 MHz, l'uscita `OutBCDValue` *rappresenta le 4 cifre "1342", che poi il blocco Displayer provvederà ad elaborare, mentre l'uscita* `OutSizeNumber` *indicherà che l'ordine di grandezza è quello dei KHz.*

Analisi Input/Output :

- `InHexValue`: è l'unico ingresso del blocco che porta all'interno il valore da trasformare in forma decimale.
- `OutBCDValue`: è l'uscita contenente, in 4 blocchi di 4-bit, le cifre più significative da visualizzare.
- `OutSizeNumber`: è l'uscita rappresentante l'ordine di grandezza delle cifre rappresentate da `OutBCDValue`.
-

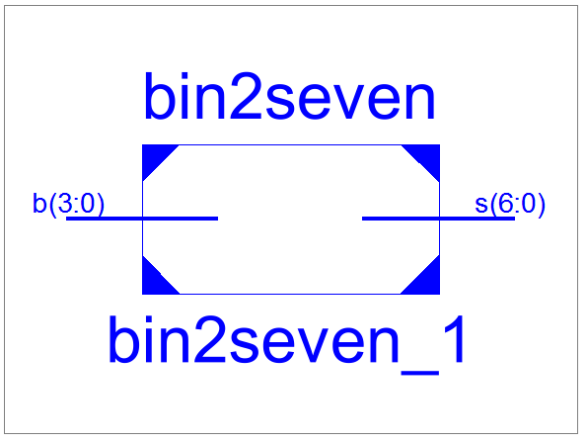
Codice di implementazione

La realizzazione del convertitore `BCD` è stata implementata con una FSM e tramite l'utilizzo di maschere per determinare l'ordine di grandezza del valore in ingresso. Il valore in ingresso rappresentante il valore della frequenza in binario, viene trasformato in un intero e poi elaborato per determinare, in base all'ordine di grandezza, le cifre più significative che lo compongono. In *appendice 3-iii* viene riportato il codice responsabile del calcolo del range della frequenza e delle cifre significative.

Displayer

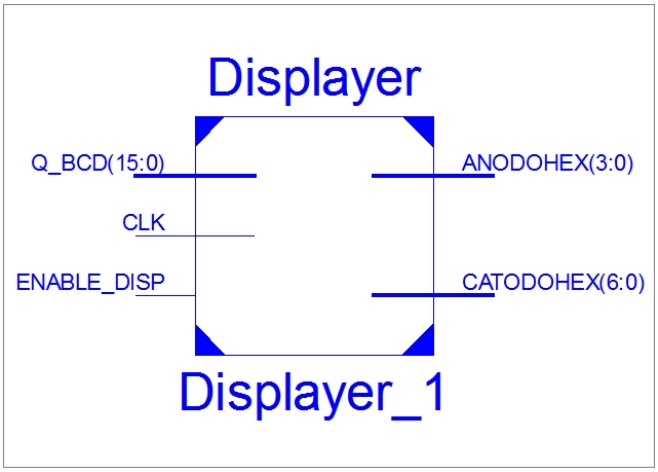
Si tratta di un blocco per la visualizzazione sui display a 7 segmenti di un numero decimale.

Il blocco contiene anche una sub componente chiamata *"bin2seven"* che provvede a codificare una stringa di 4-bit in una stringa di 7-bit che rappresenta quali sono i segmenti da accendere del display a 7 segmenti. Nell'ipotesi in cui l'ingresso non rappresenti un numero tra 0 e 9 l'uscita rappresenterà un segnale che una volta applicato spegnerà il display (nessun segmento acceso).



Per via della struttura della scheda, i segmenti dei display sono tutti controllabili tramite 7 pin, cioè quelli in uscita dal sotto-blocco *"bin2seven"*. Per decidere su quale dei 4 display visualizzare il numero invece, si deve settare a massa il pin del display desiderato lasciando alti gli altri. Per fare questo viene usata l'uscita ANODOHEX.

Il tutto funziona sfruttando la lentezza dell'occhio umano. Settando sufficientemente bassa la costante di aggiornamento in questo blocco è infatti possibile vedere come vengono aggiornate le varie cifre una dopo l'altra mentre se si vuole evitare lo "sfarfallio" del display la frequenza di aggiornamento è bene si trovi tra i 50Hz e 100Hz.



È bene notare che in caso di una frequenza di aggiornamento troppo alta, i led non riuscirebbero nemmeno ad accendersi e il risultato è una visualizzazione molto debole del numero.

Analisi Input/Output :

- `Q_BCD`: è l'ingresso contenente 4 cifre rappresentate nella forma decimale. È collegata, come è possibile vedere nella "Visione d'insieme" in *figura 3-2*, all'uscita del blocco "*BCDForm"*.
- `Clk`: rappresenta il clock in ingresso da usare per decidere quando accendere i display. Quando il blocco viene instanziato all'interno di un progetto di alto livello ("*FrequencyAnalyzer*" nel mio caso), deve essere specificata una costante contenente la frequenza del clock e anche la frequenza con la quale si vogliono aggiornare le cifre.
- `Enable_disp`: è un segnale in ingresso *"active_high" – "attivo quando alto"* per segnalare al blocco quando si vuole visualizzare il numero in ingresso. Nel caso questo segnale venga messo a massa il blocco spegne tutti i display. Nel mio progetto viene settato sempre alto in modo tale che i display funzionino sempre.
- `AnodoHex`: è l'uscita da collegare agli anodi in comune dei display. Poiché vi sono 4 display il numero di anodi è 4.
- `CatodoHex`: è l'uscita da collegare ai catodi di ogni singolo display. Come già detto, tutti i catodi sono in comune tra i 4 display per lo stesso segmento e quindi bastano 7 segnali, uno per segmento, per controllare tutti i display.

Codice di implementazione

Per poter visualizzare sui 4 display a 7 segmenti un numero è necessario, a causa della configurazione hardware a catodi e anodi comuni, abilitare in modo alternato i display ad una frequenza sufficientemente elevata così che l'occhio umano non si accorga dell'aggiornamento. Tuttavia si deve porre attenzione a non esagerare in quanto una frequenza troppo elevata non lascerebbe il tempo di accendere i led risultando in una scritta opaca. All'interno della sezione variabili generiche si trovano la frequenza dell'oscillatore interno e quella di visualizzazione che devono essere impostate dal progetto principale che utilizza questo modulo. Per poter ottenere un aggiornamento della frequenza desiderata, è stata creata una FSM tramite la quale si accendono i display in modo alternato. A passare da uno stato all'altro provvede un contatore interno usato da un processo per decretare quando ci si trova in uno stato piuttosto che nell'altro. Inoltre per passare dalla forma binaria a quella necessaria per accendere correttamente i led del display a 7 segmenti, è stato utilizzato il blocco "*bin2seven*" che codifica le cifre in un codice adatto ai display. Nell'*appendice 3-iv* è possibile osservare il codice del codificatore "*bin2seven*".

Capitolo 4

Simulazioni [3]

Terminata la fase di progettazione e di scrittura del codice, sono passato alle simulazioni. Le simulazioni, per limitare i tempi da simulare, sono state fatte impostando la frequenza di aggiornamento così che ogni 4 periodi del clock esterno da misurare, uno venga campionato.

Le frequenze usate per la simulazione sono 3 diverse e pari a: 1MHz, 125KHz, 16KHz.

Vi sono riportate 2 tipologie di simulazione. La prima è quella "***Behavioral***" – *"Comportamentale"*che corrisponde al comportamento che dovrebbero assumere le uscite e gli ingressi della scheda analizzando solamente il codice che costituisce il progetto. Nella simulazione "*Behavioral*" pertanto viene simulato solo il codice e quindi non la vera e propria struttura hardware implementata; per questo, tra l'altro, non sono considerati i ritardi o le limitazioni dovute al FPGA quali i delays dovuti alla propagazione dei segnali in ingresso o uscita. La seconda tipologia di simulazione è quella "***Post place & route***" – *"Dopo il piazzamento e il cablaggio"*, nella quale viene fatta la simulazione del reale circuito che si sta implementando e utilizza un modello ricavato dal risultato della fase di *"place and route"* – *"piazzamento e cablaggio*" (quando cioè viene stabilito dove i blocchi e i componenti debbano essere posizionati) in questo modo, tenendo conto delle performance della scheda inserendo eventuali *"delays"* – *"ritardi"* si ha una stima molto vicina alla realtà del comportamento del sistema realizzato.

Nell'*appendice 4-i* è stato riportato il codice del *"testbench"* – *"banco di prova"* usato per la generazione dei clock nella simulazione del progetto.

Ogni simulazione riporta 10ms di tempo, il minimo necessario per poter visualizzare come variano tutte le 4 cifre. Poiché la frequenza di *"refresh"* – *"aggiornamento"* è impostata su 100Hz infatti, ogni cifra viene aggiornata dopo un tempo di 10ms che è il periodo di *"refresh"*.

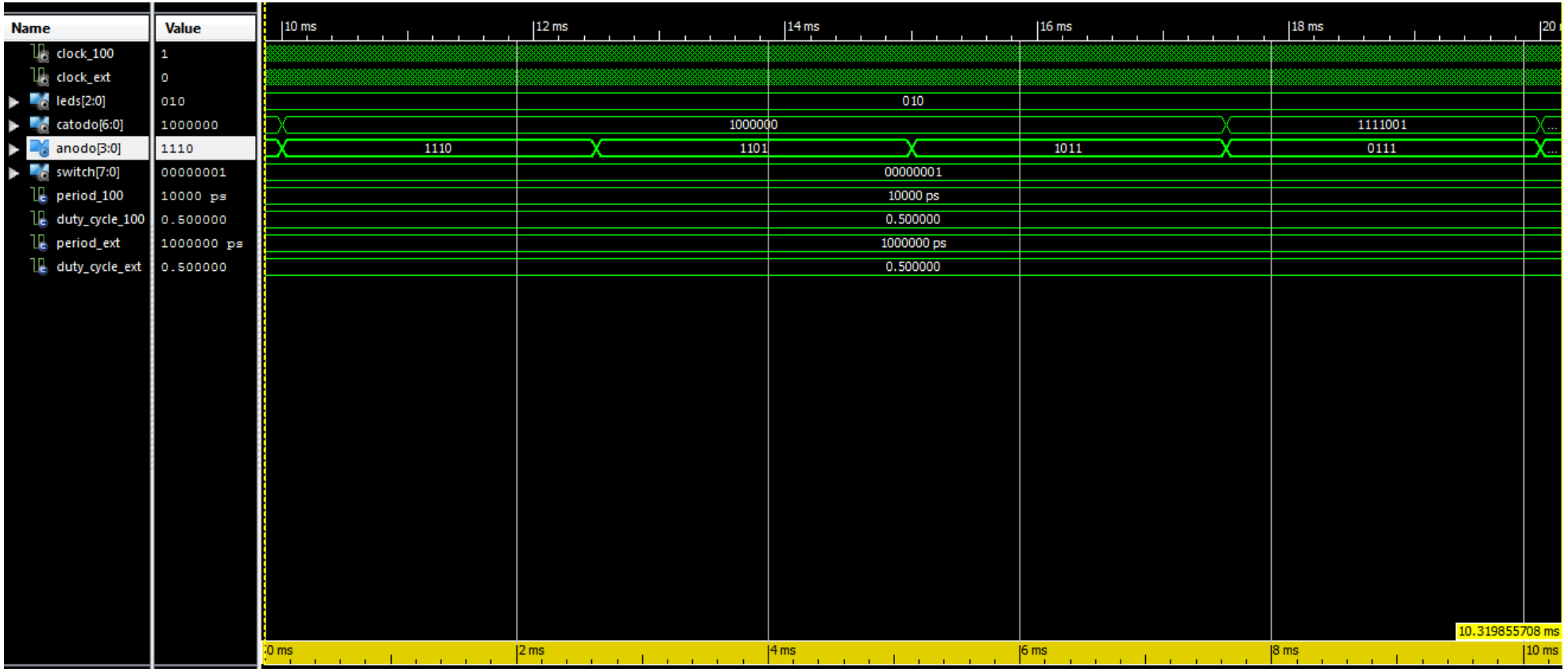
Le uscite anodo riportano quale cifra al momento viene visualizzata mentre le uscite catodo riportano i segmenti che vengono accesi con un segnale *"active low"*. Per tradurre le uscite è necessario fare riferimento al segmento di codice "*bin2seven*" riportato nel'appendice.

Simulazione Behavioral

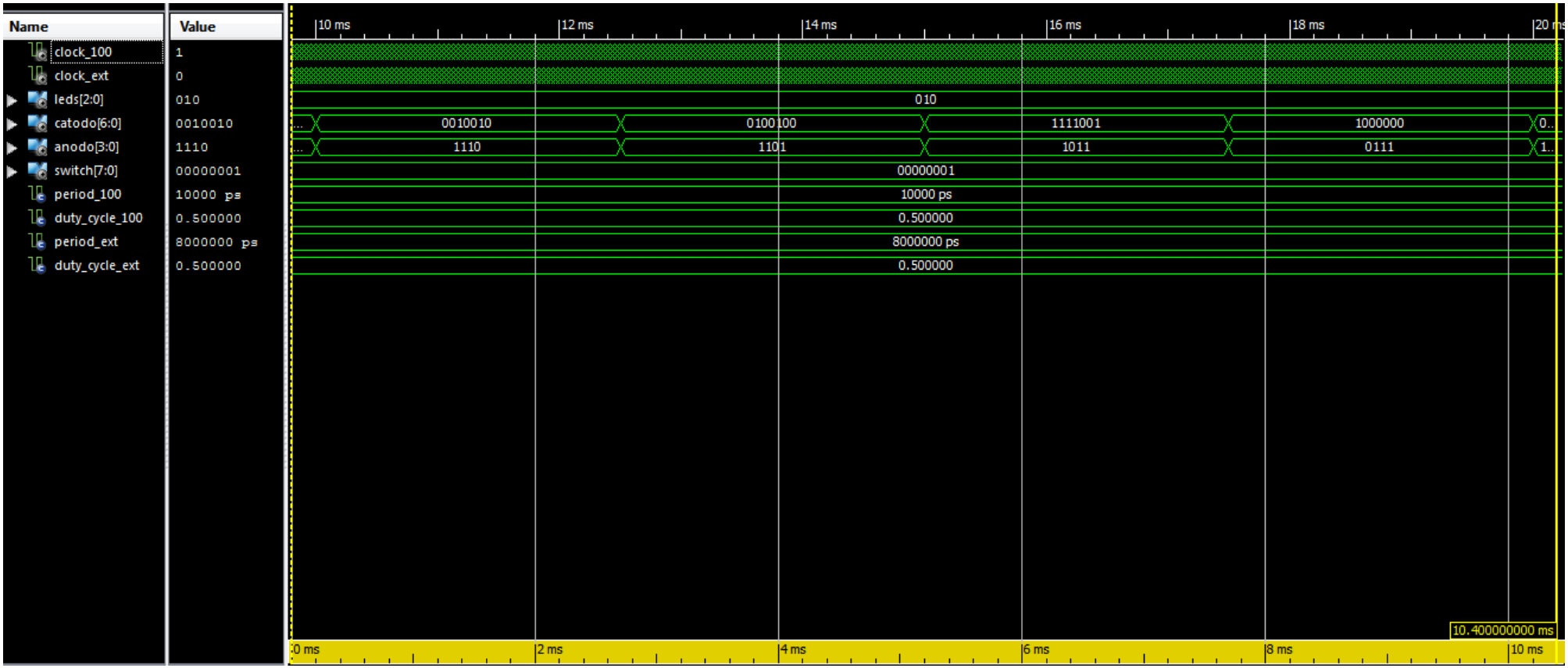
Nelle immagini sottostanti sono state riportate le simulazioni *"Behavioral"* – *"Comportamentali"* alle diverse frequenze.

È possibile notare che il comportamento delle simulazioni *"Behavioral"* è quello che ci si aspettava, sia le cifre che l'unità di misura costituita dai led risulta esatta, e quindi il codice usato è corretto dal punto di vista logico.

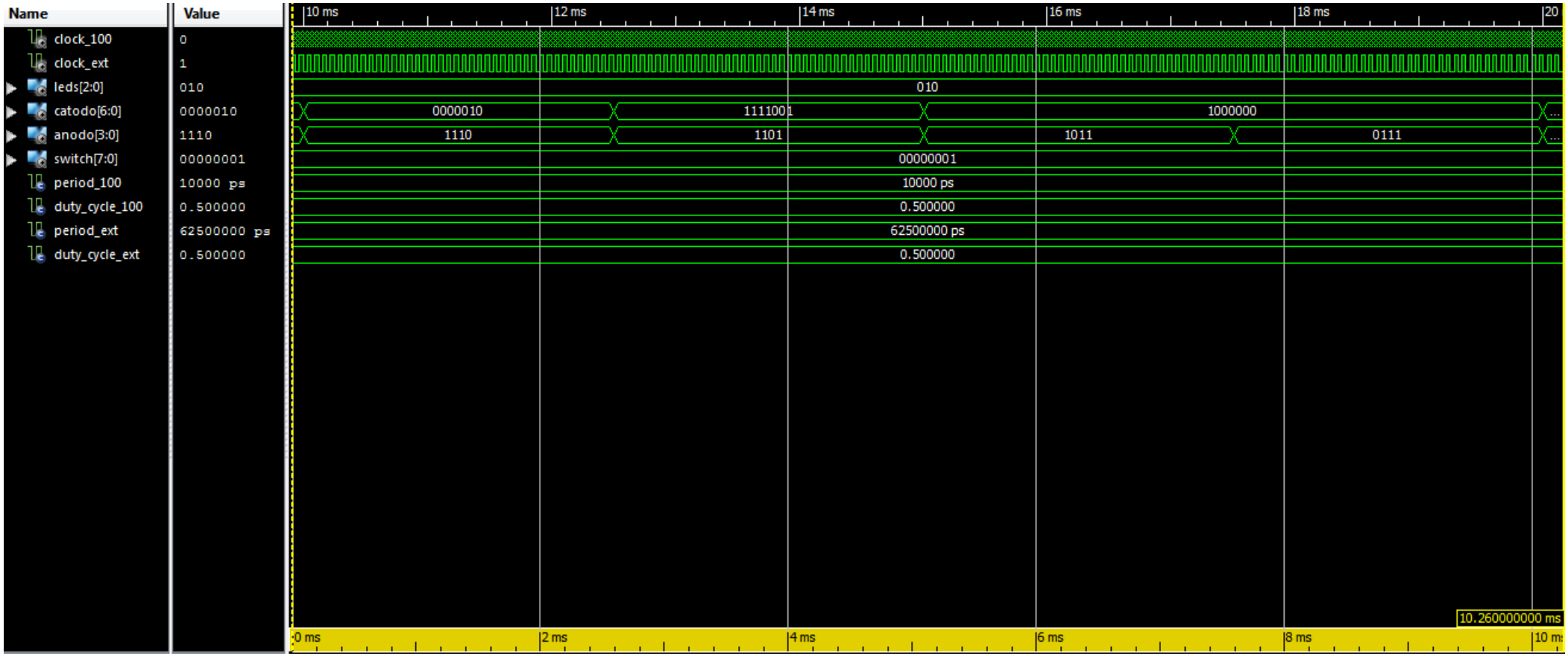
Simulazione Behavioral a 1 MHz



Simulazione Behavioral a 125 KHz



Simulazione Behavioral a 16 KHz



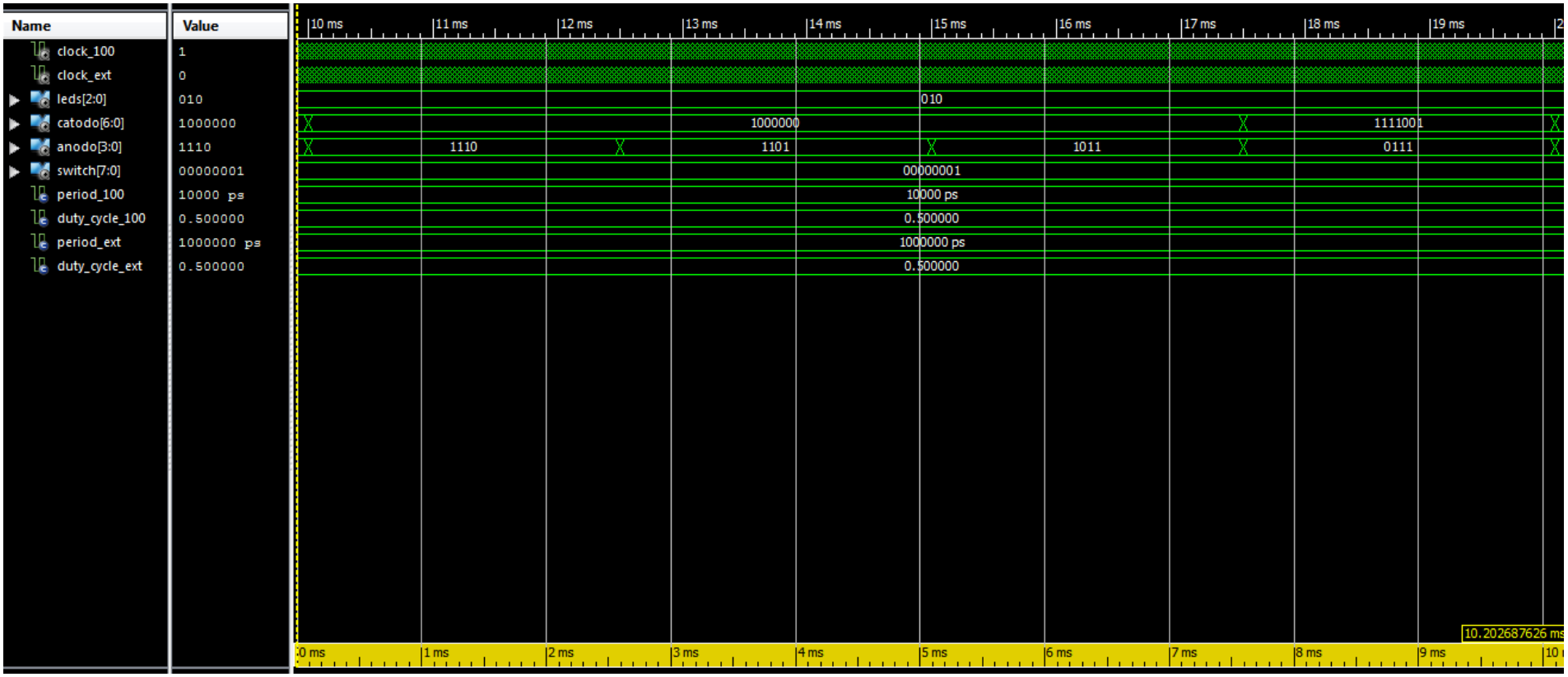
Simulazione Post Route&Place

Nelle immagini sottostanti sono riportate le simulazioni *"Post route & place"* - *"Dopo il piazzamento e il cablaggio".*

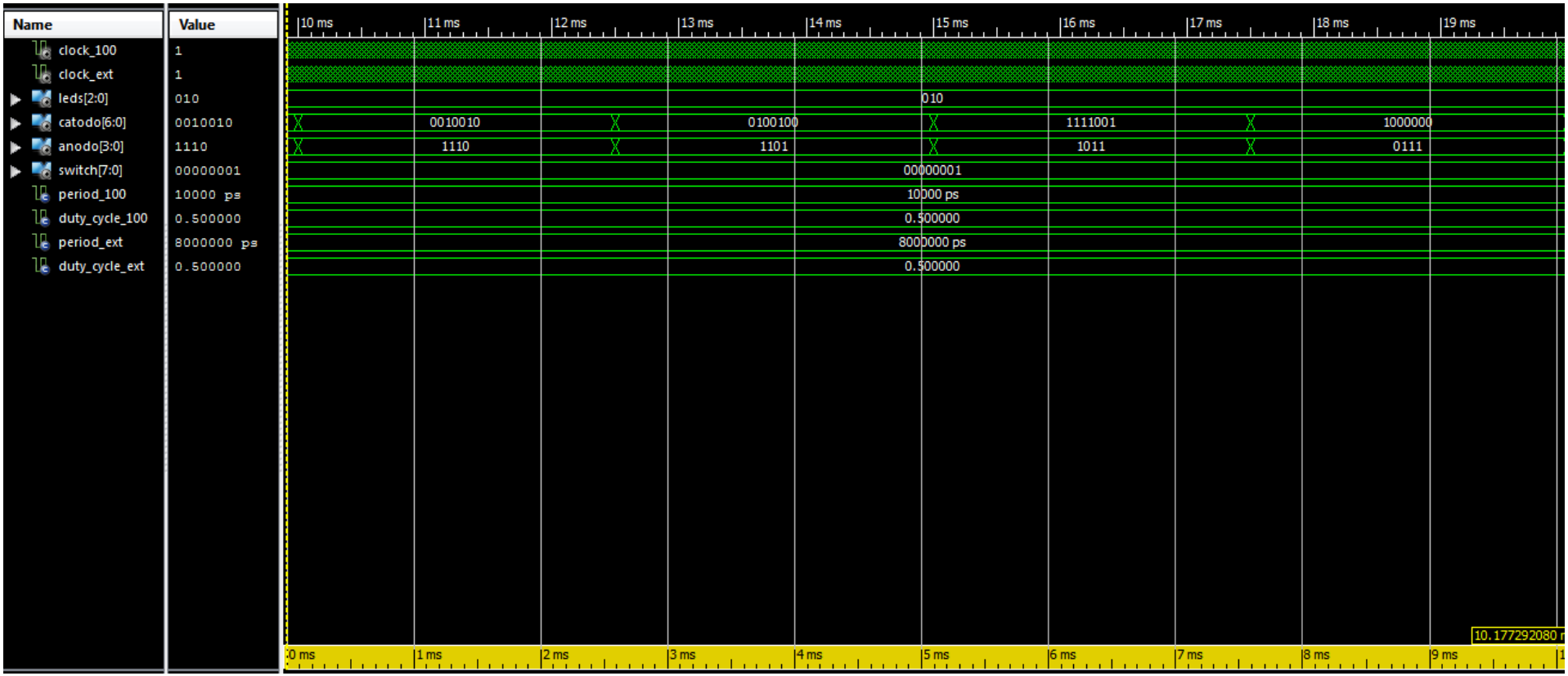
Nonostante ad una prima impressione sembrino risultare uguali alle simulazioni *"Behavioral"*, se guardate con occhio attento riportano in realtà delle piccole distorsioni presenti in particolar modo quando viene cambiata la cifra da visualizzare sul display a 7 segmenti. Le immagini "*dettaglio a 1 MHz*" e *"dettaglio a 16 KHz*" rappresentano un particolare del cambio cifra e cambio del display abilitato dove è possibile notare la presenza di disturbi che vanno a compromettere il corretto funzionamento. Tali disturbi sono presenti per ogni frequenza.

Nonostante questo l'andamento della simulazione rappresenta comunque un comportamento molto simile a quello ideale e le distorsioni hanno sempre una durata (dell'ordine del centinaio di *ps*) infinitesima rispetto al periodo di "*refresh*" del display o dei clock.

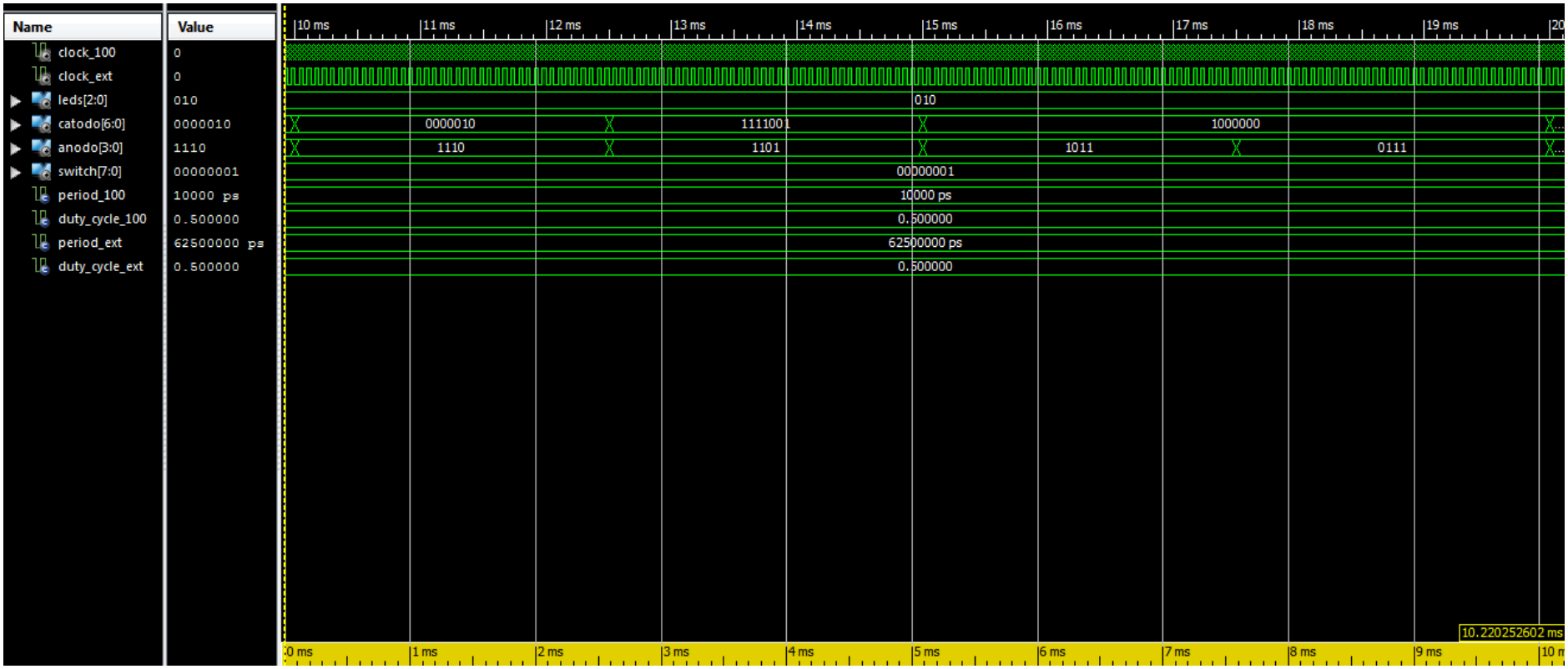
Simulazione Post route&place 1MHz



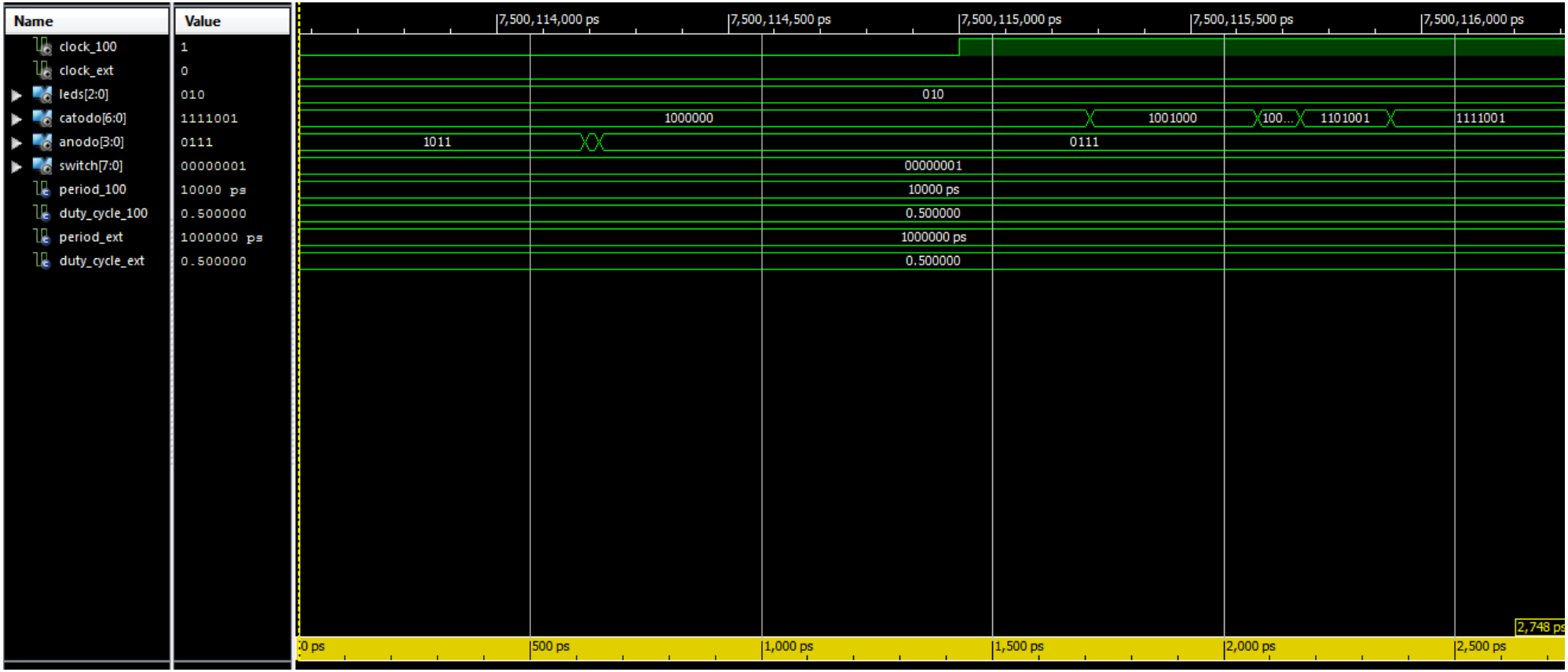
Simulazione Post route&place 125KHz



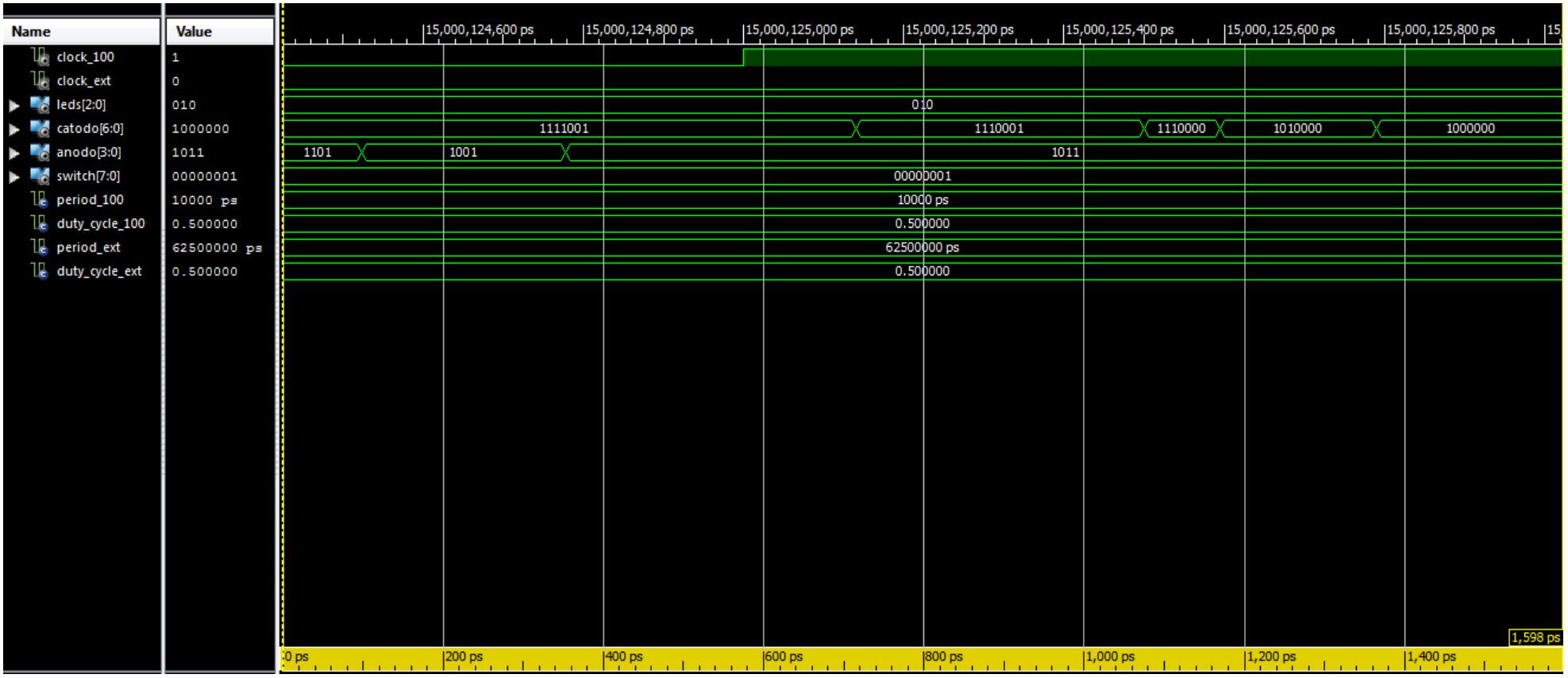
Simulazione Post route&place 16KHz



Particolare della simulazione Post route&place 1MHz



Particolare della simulazione Post route&place 16KHz



Test sulla board

I test sulla scheda *"Diligent"* sono stati effettuati principalmente tramite l'uso di un circuito generante un segnale digitale a frequenza di 125KHz. È stato possibile testare il progetto anche con un generatore d'onda digitale messo a disposizione dall'Università di Padova. Con tale generatore sono state testate diverse frequenze per verificare l'effettivo funzionamento del progetto nel range dato nelle specifiche.

Come ingressi/uscite della scheda sono stati usati tutti gli switch, 3 led degli 8 presenti sopra gli switch (quelli all'estrema destra), i 4 display a 7-segmenti, il clock interno a 100MHz e un pin di I/O presente sul modulo PMOD-D in alto a destra della scheda (vedi *figura 1-12* raffigurante la scheda *NEXYS-3*).

Sebbene la simulazione riporti un funzionamento corretto del progetto, nei test con la scheda si verificano delle variazioni nel calcolo della frequenza che causano l'apparizione di valori sbagliati nei display. Proprio per evitare di visualizzare un valore "sfarfallante" nei display sono stati usati gli switch per selezionare la frequenza con cui campionare il clock esterno, in tal modo è possibile mantenere costante per il tempo desiderato il valore evitando l'effetto di sfarfallio.

Poiché il valore corretto si verifica con una frequenza maggiore, il campionamento lo sceglie la maggior parte delle volte ed il valore è pressoché costante nei display. Può tuttavia capitare di poter visualizzare un risultato inappropriato.

Conclusioni

Il progetto ha come scopo il calcolo della frequenza di un'onda digitale. Tale frequenza viene visualizzata sui display a 7 segmenti presenti sulla scheda facendo uso anche dei led per l'ordine di grandezza della misura riportata.

È possibile osservare nella *figura 4-2* come tale progetto occupi ben il 78% delle slices disponibili sul FPGA. Sebbene non vi siano a disposizione altre architetture con cui comparare il progetto, è possibile ipotizzare che una delle possibili cause delle errate misurazioni sia dovuto ad un'eccessiva necessità di slices da parte del progetto che comporta una disposizione dei vari blocchi e componenti logici non ottimale dal punto di vista delle performance temporali. Ciò quindi ricondurrebbe a delays e disturbi che compromettono il corretto funzionamento.

Un "*warning*" – "*avvertimento*" è presente in fase di implementazione ed è dovuto al fatto che l'ingresso usato per il clock esterno è un semplice ingresso di I/O che quindi non è collegato nel FPGA ai pins di clock di cui esso dispone. Ciò fa si che non vengano utilizzati i bus veloci messi a disposizione sul FPGA per trasportare segnali di clock. Pertanto l'errore nella misurazione potrebbe essere dovuto anche a questa mancanza nella scheda di un ingresso dedicato a clock esterni.

Come è possibile notare anche nella *figura 4-1*, tutte le costanti temporali settate in fase di sintesi sono state rispettate, tali costanti sono settate nel progetto in modo tale da corrispondere alla frequenza interna da 100MHz e al limite massimo stabilito nelle specifiche della frequenza calcolabile cioè 1MHz.

Le impostazioni usate per la sintesi ed implementazione del progetto sono quelle che favoriscono un bilanciamento tra consumo energetico, prestazioni temporali e dimensioni in termini di area occupata dal progetto sul FPGA.

Nella *figura 4-1* e *figura 4-2* è stata riportata una parte del riassunto fatto dopo l'implementazione da parte dal software "*ISE Design Suite*" in dotazione con la scheda.

Project summary			
Project Name	FrequencyAnalyzer	**Errors**	No Errors
Target Device	xc6slx16 3csg324	**Warnings**	1 Warning
Product Version	ISE 14.2	**Routing Results**	All Signals Completely Routed
Design Goal	Balanced	**Timing Constraints**	All Constraints Met

Figura 4-1

Device Utilization Summary			
Slice Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	112	18’224	1%
Number of Slice LUTs	5’456	9’112	59%
Number of Occupied Slices	1’796	2’278	78%
Nummber of MUXCYs used	3’544	4’556	77%
Number of IOBs	24	232	10%

Figura 4-2

Tramite il programma "*XPower Analyzer*" è stato possibile simulare e analizzare, in termini potenza, i consumi del FPGA mentre esegue il progetto. In *figura 4-3* è riportata la stima dei consumi che potrebbe avere il progetto. Il consumo totale stimato è di 43mW di cui 22mW dovuti ai consumi dinamici mentre 20 mW a quelli statici. Analizzando i consumi del FPGA nel dettaglio le componenti dominanti risultano essere la dispersione con 20 mW e la gestione di ingressi e uscite con 15 mW.

Device		**Power consumption On-Chip (mW)**	
Family	Spartan6	**Clocks**	5
Part	xc6slx16	**Logic**	1
Package	csg324	**Signals**	1
Temp Grade	C-Grade	**IOs**	15
		Leakage	20

Supply Power (mW)	Total	Dynamic	Quiescent
	43	22	20

Junction Temperature (C)	26.2
Ambient Temp (C)	25.0

Figura 4-3

In conclusione si può dire che sebbene non sia stato raggiunto un funzionamento ottimale del progetto, in quanto talvolta calcoli valori errati, lo scopo del misuratore di frequenza è stato ottenuto. Probabilmente è possibile raggiungere la completezza del progetto tramite l'utilizzo di schede più complete o altri accorgimenti (vedi "*sviluppi futuri*").

Sviluppi futuri

Diverse migliorie possono essere apportate al progetto in caso di uno sviluppo futuro. Esse possono essere riassunte in 2 tipologie: progettazione e hardware.

Progettazione

Poiché l'obbiettivo principale del progetto è il calcolo della frequenza di un ingresso digitale, la sezione riguardante la visualizzazione del risultato è stata implementata in modo molto nativo con codice di "alto livello comportamentale" senza l'utilizzo di complesse o dedicate architetture che invece ne alleggerirebbero la realizzazione sul FPGA. Pertanto una miglioria che è possibile apportare è lo sviluppo e la ricerca di un'architettura che implementi le divisioni, le moltiplicazioni e i calcoli effettuati per visualizzare il risultato sui 4 display a 7-segmenti e sui led in modo ottimale.

Uno sviluppo futuro potrebbe includere un ulteriore blocco od una modifica a quelli esistenti con lo scopo di aumentare il numero di periodi del clock esterno entro il quale effettuare il conteggio del clock interno. Questa modifica permetterebbe di aumentare le prestazioni in termini di qualità della misura poiché ridurrebbe l'errore dovuto al conteggio del clock interno entro quello esterno.

Un'ulteriore miglioria progettuale può essere fatta usando, anziché il generatore interno da 100MHz collegato al progetto,dei generatori di clock esterni per generare clock a frequenze più elevate (per esempio la massima raggiungibile dal progetto che si ottiene in fase di place&route) e quindi migliorare la precisione della misura permettendo di aumentare il range delle frequenza calcolabili.

Hardware

Gli possibili sviluppi futuri proposti dal punto di vista della progettazione sono tuttavia difficili da realizzare dal punto di vista dell'implementazione sulla scheda corrente in quanto il progetto allo stato attuale occupa (come è possibile vedere nella figura 4-2) una alta percentuale delle slices del FPGA. Inoltre a causa della mancanza di un ingresso dedicato al clock esterno si è dovuto usare uno dei pin del PMOD solitamente dedicati ad ingressi I/O che non supportano velocità elevate di switch (a questo è dovuta la presenza di warning in fase di implementazione).

Pertanto è consigliato nel caso di ulteriori sviluppi il passaggio ad una scheda con maggiori proprietà e prestazioni, possibilmente facente uso di un FPGA con un numero maggiore di slices e pertanto avente ulteriore posto per implementare i blocchi logici sopra citati per migliorare il progetto.

Una modifica hardware, che è possibile apportare senza modificare la scheda su cui è stato implementato il progetto, consiste nell'utilizzo di cavi coassiali per la trasmissione del segnale così da ridurre la presenza di rumore sul segnale. Una delle possibili cause di una visualizzazione talvolta errata del risultato può essere infatti dovuta alla presenza di rumore nel segnale che durante il breve periodo di salita/discesa del segnale digitale può influenzare il corretto funzionamento della scheda creando fronti di salita/discesa inesistenti.

Bibliografia

[1] Testo "The Design Warrior's Guide to FPGAs";

[2] Slides corso "Laboratorio di Elettronica" a cura del professore D. Vogrig;

[3] Tools del pacchetto "Xilinx ISE Design Suite 14.2";

[4] Codice del progetto;

[5] Sito web della Xilinx: www.xilinx.com;

***[6]* Sito web della Diligent: *www.diligentinc.com*;**

Appendice

Appendice 3-i, Counter Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity HEXCounter is
        generic (
              INITVALUE  : integer;
              HEXDIGITS  : integer);
        Port (
              clk_fast : in std_logic;
              enable_n : in  STD_LOGIC;
              counter  : out  STD_LOGIC_VECTOR (HEXDIGITS*4 - 1 downto 0));
end HEXCounter;

architecture Behavioral of HEXCounter is
signal temp_counter       : unsigned(counter'range);
signal resetted           : std_logic;

component dff_vector
       Generic(
                VECTORLENGTH  : integer);
       Port (
                clk         : in  STD_LOGIC;
                D           : in  STD_LOGIC_VECTOR (VECTORLENGTH - 1 downto 0);
                Q_dff_v     : out  STD_LOGIC_VECTOR (VECTORLENGTH - 1 downto 0));
end component;

begin --Behavioral

dff_vector_counter: dff_vector
       Generic map(
                VECTORLENGTH  => HEXDIGITS*4)
       Port map(
                clk         => enable_n,
                D           => std_logic_vector(temp_counter),
                Q_dff_v     => counter);


cou: process(clk_fast)
begin
if clk_fast'event and clk_fast = '1' then
       if ENABLE_N = '1' then
              temp_counter <= temp_counter;
              resetted <= '0';
       else
              if resetted = '0' then
                     temp_counter <= (0=> '1',others => '0');
                     resetted <= '1';
              else
                     temp_counter <= temp_counter + 1;
              end if;
       end if;
end if;
end process cou;

end Behavioral;
```

Appendice 3-ii, SamplerExtClk Code

```
...

type statemachine is (stateenable,statedisable,counting);
signal current_state,next_state  : statemachine;
signal old_clk_slow              : std_logic;
signal counter                   : unsigned (26 downto 0);
signal numbercicle  : unsigned (26 downto 0):= to_unsigned( 260000000,27);
signal exp                       : unsigned(1 downto 0);
signal number                    : unsigned(5 downto 0);

begin--Behavioral
exp <= unsigned(SW (7 downto 6));
number <= unsigned(SW (5 downto 0));

state_reg: process (clk_fast)
begin
if clk_fast'event and clk_fast = '1' then
       if clk_slow = not old_clk_slow then
              current_state <= next_state;
       else
              current_state <= current_state;
       end if;
old_clk_slow <= clk_slow;
end if;
end process state_reg;

next_state_logic: process(clk_slow,current_state)
variable temp_next_state : statemachine;
begin
if clk_slow'event and clk_slow = '1' then
       if counter <=  numbercicle + 1 then
              counter <= counter + 1;
       else
              counter <= (others => '0');
       end if;
       temp_next_state := statedisable;
       case current_state is
              when stateenable =>
                           if clk_slow = '1' then
                                  temp_next_state := counting;
                           end if;
              when counting    =>
                           if counter = numbercicle then
                                  temp_next_state := statedisable;
                           else
                                  temp_next_state := counting;
                           end if;
              when others =>
                           if clk_slow = '1' then
                                  temp_next_state := stateenable;
                           end if;
              end case;
next_state <= temp_next_state;
end if;
end process next_state_logic;

...

end Behavioral;
```

Appendice 3-iii, BCDForm Code

```
...

trs: process(InHexValue,range_123)
       constant mask_1 : integer := 10;
       constant mask_2 : integer := 100;
       constant mask_3 : integer := 1000;
       constant mask_6 : integer := 1000000;
variable temp_in                                  : integer;
variable temp_val0,temp_val1,temp_val2,temp_val3 : integer;
variable tempDig1,tempDig2,tempDig3               : integer;

begin
temp_in := to_integer(unsigned(InHexValue));
       case range_123 is
              when 2 => temp_val3 := temp_in/mask_6;
                                    OutSizenumber <= "100";
              when 1 => temp_val3 := temp_in/mask_3;
                                    OutSizenumber <= "010";
              when others => temp_val3 := temp_in;
                                    OutSizenumber <= "001";
       end case;
        tempDig3    := temp_val3/mask_3;
        temp_val2 := temp_val3 - tempDig3*mask_3;
        tempDig2    := temp_val2/mask_2;
        temp_val1 := temp_val2 - tempDig2*mask_2;
        tempDig1    := temp_val1/mask_1;
        temp_val0 := temp_val1 - tempDig1*mask_1;
        temp_bcd (4*4 - 1 downto 3*4) <=
std_logic_vector(to_unsigned(tempDig3,4));
        temp_bcd (3*4 - 1 downto 2*4) <=
std_logic_vector(to_unsigned(tempDig2,4));
        temp_bcd (2*4 - 1 downto 1*4) <=
std_logic_vector(to_unsigned(tempDig1,4));
        temp_bcd (1*4 - 1 downto 0)   <=
std_logic_vector(to_unsigned(temp_val0,4));
end process trs;

choose_ran: process(InHexValue)
variable temp_in : integer;
constant mask1 : integer := 10000000;
constant mask2 : integer := 10000;
variable rest1,rest2      : integer;
begin --choose_ran
       temp_in := to_integer(unsigned(InHexValue));
       rest1 := temp_in/mask1;
       rest2 := temp_in/mask2;
       if rest1 = 0 and rest2 = 0 then
              range_123 <= 0;
       elsif rest1 = 0 then
              range_123 <= 1;
       else
              range_123 <= 2;
       end if;

end process choose_ran;

OutBCDValue (15 downto 0) <= temp_bcd(15 downto 0);

end architecture rtl;
```

Appendice 3-iv, Bin2seven Code

```
library ieee;
use ieee.std_logic_1164.all;

entity bin2seven is
  port(b : in  std_logic_vector(3 downto 0);
       s : out std_logic_vector(6 downto 0));
end entity bin2seven;

-- Encoding of s is:
--
--    - 0 -
--    5   1
--    - 6 -
--    4   2
--    - 3 -

architecture rtl of bin2seven is
begin
  with b select
       s <=
             "1000000" when "0000", -- 0
             "1111001" when "0001", -- 1
             "0100100" when "0010", -- 2
             "0110000" when "0011", -- 3
             "0011001" when "0100", -- 4
             "0010010" when "0101", -- 5
             "0000010" when "0110", -- 6
             "1111000" when "0111", -- 7
             "0000000" when "1000", -- 8
             "0011000" when "1001", -- 9
             "1111111" when others; -- Could also use don't cares here.
end architecture rtl;
```

Appendice 4-i, TestBench Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY freqAnalyzer_tb IS
END freqAnalyzer_tb;
ARCHITECTURE testbench OF freqAnalyzer_tb IS

   COMPONENT frequencyAnalyzer
      port (
            clk_100       :  in std_logic; --internal clk at 100 MHz
            clk_ext       :  in std_logic; --external clk
            SW            :  in std_logic_vector(7 downto 0);
            LED           :  out std_logic_vector(2 downto 0);
            CATODOHEX     :  out std_logic_vector(6 downto 0);
            ANODOHEX            : out std_logic_vector(4 - 1 downto 0) );
    END COMPONENT;

    SIGNAL CLOCK_100      : std_logic := '0';
       signal clock_ext   : std_logic := '0';
    signal leds           : std_logic_vector(2 downto 0);
       signal catodo      : std_logic_vector(6 downto 0);
       signal anodo       : std_logic_vector(4 - 1 downto 0);
       signal switch      : std_logic_vector (7 downto 0);

    constant PERIOD_100          : time := 10 ns;
    constant DUTY_CYCLE_100      : real := 0.5;
    constant OFFSET_100          : time := 100 ns;
    constant PERIOD_ext          : time := 100000 us;
    constant DUTY_CYCLE_ext      : real := 0.5;
    constant OFFSET_ext          : time := 100 ns;

      BEGIN
    UUT : FrequencyAnalyzer
           port map(
                   clk_100   => clock_100,
                   clk_ext   => clock_ext,
                   SW        => switch,
                   LED       => leds,
                   CATODOHEX => catodo,
                   ANODOHEX  => anodo);

   clock_100_gen: PROCESS
   BEGIN
            WAIT for OFFSET_100;
            CLOCK_100_LOOP : LOOP
                   CLOCK_100 <= '0';
                   WAIT FOR (PERIOD_100 - (PERIOD_100 * DUTY_CYCLE_100));
                   CLOCK_100 <= '1';
                   WAIT FOR (PERIOD_100 * DUTY_CYCLE_100);
            END LOOP CLOCK_100_LOOP;
   END PROCESS;

   clock_ext_gen: PROCESS
   BEGIN
            WAIT for OFFSET_ext;
            CLOCK_ext_LOOP : LOOP
                   CLOCK_ext <= '0';
                   WAIT FOR (PERIOD_ext - (PERIOD_ext * DUTY_CYCLE_ext));
                   CLOCK_ext <= '1';
                   WAIT FOR (PERIOD_ext * DUTY_CYCLE_ext);
            END LOOP CLOCK_ext_LOOP;
   END PROCESS;
            switch <= "00000001";
END testbench;
```