UNIVERSITÀ DI PADOVA

DIPARTIMENTO DI INGENERIA DELL'INFORMAZIONE
CORSO DI LAUREA IN INGEGNERIA ELETTRONICA

Interfaccia per VGA e Mouse USB su board FPGA NEXYS3

(VGA and USB Mouse Controllers on NEXYS3 FPGA board)

Laureando:
Loris Luise
Matricola: 1037252

Relatore:
Daniele Vogrig

Anno Accademico 2012/2013

Sommario

Questo documento descrive la realizzazione di un driver Mouse USB secondo il protocollo PS/2(IMPS/2) con relativa interfaccia video su piattaforma FPGA della famiglia Xilinx Spartan 6.

La scelta del FPGA garantisce un forte vantaggio nella realizzazione del prototipo per la sua facilità di riprogrammazione e per le potenzialità che ormai hanno raggiunto.

Il sistema realizzato si compone essenzialmente di due parti scritte in VHDL per l'implementazione sull'FPGA, la prima descrive un'interfaccia verso la periferica Mouse USB, invece nella seconda sezione si realizza un driver in grado di pilotare un monitor con risoluzione 640 per 480 pixel.

La scheda utilizzata (Nexys3) monta un FPGA di marca Xilinx e la PCB è stata realizzata dalla Digilent. Il software in uso è provveduto dalla Xilinx ed in particolare è stato usato il pacchetto "ISE Design Suite V14.5".

Indice

Capitolo 1

Introduzione

1.1 Scopo della tesi

Lo scopo di questa tesi è di realizzare un driver per la gestione di una periferica connessa alla board FPGA (Nexys 3) tramite porta USB e di un driver per la relativa visualizzazione attraverso l'interfaccia VGA. Le periferiche prese in esame sono il mouse e uno schermo con ingresso VGA vedi figura 0.1.
Il progetto è implementato attraverso il linguaggio VHDL, specifico per la descrizione di architetture e la progettazione di hardware digitale, ed il suo utilizzo è ampio in tutta Europa essendo uno standard nel settore.
La periferica connessa alla scheda, mouse o tastiera, prevede l'invio delle informazioni utili per una corretta acquisizione attraverso protocollo PS/2 invece, il monitor addetto alla visualizzazione prevede solo l'acquisizione delle informazioni.

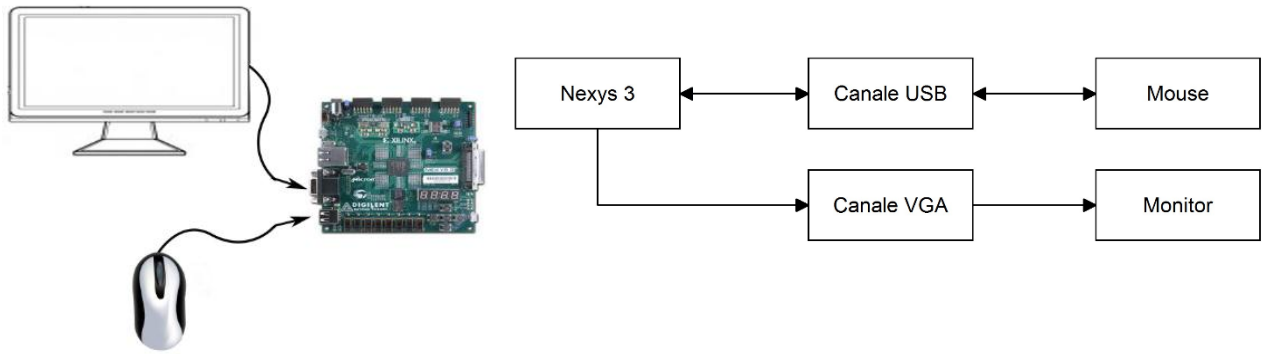


Figura 0.1 Schema generale del progetto

1.2 Struttura della tesi

Nei primi due capitoli si forniscono le nozioni base per comprendere i contenuti di questa tesi in particolare, nel secondo capitolo si prendono in considerazione i dispositivi utilizzati, i protocolli per la comunicazione e il tool di sviluppo.
Il terzo capitolo è mirato alla descrizione del progetto, il quale è stato diviso in moduli che poi sono stati istanziati all'interno del progetto finale per una maggiore riusabilità e leggibilità del software. Inoltre sono stati studiati i passi per la programmazione della scheda focalizzando l'attenzione su aspetti quali le tecnologie usate, il linguaggio usato, i componenti progettati per il pilotaggio di led, pulsanti ect.
Il quarto capitolo mostra i risultati ottenuti e i test effettuati concludendo la tesi esponendone i possibili sviluppi futuri.

Capitolo 2

Hardware, software e interfacce di comunicazione

2.1 I dispositivi programmabili

I circuiti logici programmabili costituiscono un nutrito gruppo di componenti elettronici il cui comportamento viene definito tramite programmazione da parte del progettista stesso, anche successivamente al momento in cui il dispositivo viene montato nel circuito, al contrario dei normali circuiti integrati che vengono progettati e realizzati per uno specifico uso.
I primi componenti programmabili fecero la loro comparsa intorno agli anni ’70, basati sulla tecnologia ad antifusibile al silicio: un componente che normalmente si comporta come isolante ma che in seguito ad opportune sollecitazioni in tensione diventa conduttore, permettendo quindi di realizzare un insieme di funzioni combinatorie (tramite collegamento di elementi logici) relativamente semplici.
Pregio di tale tecnologia è la non volatilità, per la quale sono ancora utilizzati nella realizzazione di PROM e di applicazioni speciali (aerospaziali, militari e biomedicali); d’altro lato gli alti costi, la difficoltà e la lentezza di programmazione oltre alla non reversibilità sono caratteristiche che ne decretarono la decadenza.

Con il diffondersi della tecnologia CMOS, verso gli anni ’80, fu possibile integrare un maggior numero di componenti per unità di area permettendo quindi di realizzare circuiti integrati sempre più complessi; si iniziò a collegare più blocchi PLD tramite una matrice di interconnessione dando alla luce i CPLD (*Complex PLD*) che ebbero successo grazie alla possibilità di riprogrammare i dispositivi impiegando memorie SRAM: è così possibile associare ad ogni cella di memoria un transistor MOS, programmando quindi le connessioni come aperte o chiuse in funzione del valore memorizzato.

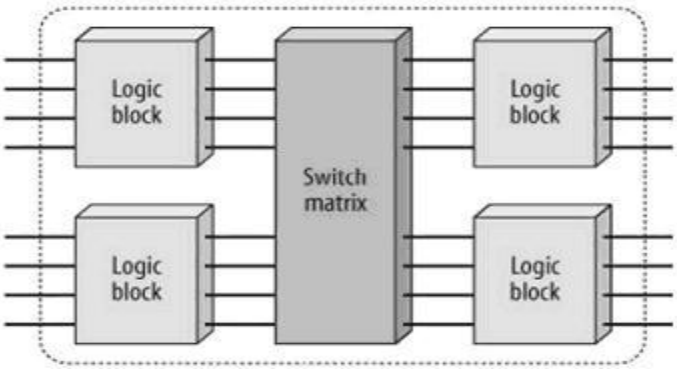


Figura 1.1 Architettura CPLD

Venne sviluppata anche la tecnologia PLA (*Programmable Logic Array*), basata su matrici di connessioni per realizzare funzioni AND e OR, ottenendo funzioni logiche a *n* ingressi definendo quali ingressi e quale risultato intermedio dovesse essere collegato ad una specifica porta logica integrata nel componente.

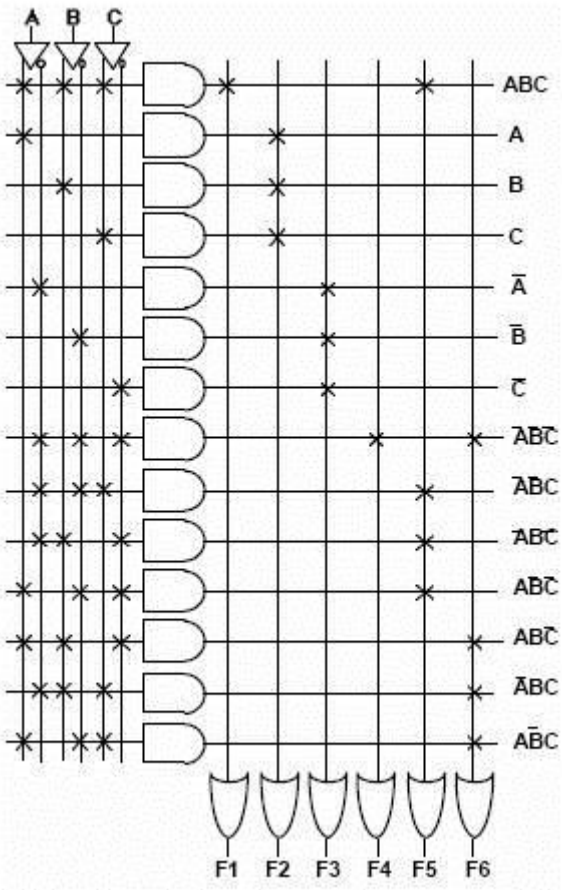


Figura 1.2 PLA

2.2 FPGA (Field Programmable Gate Array)

Un FPGA è un circuito semi-custom (ha prestazioni ottimizzate rispetto alle specifiche di sistema) e array-based (contiene una matrice di celle prefabbricate).
È un insieme di celle ed interconnessioni prefabbricate configurabili (e riconfigurabili successivamente). La funzione logica svolta da ciascuna cella e i collegamenti fra i terminali delle celle sono determinati commutando interruttori programmabili.
Gli elementi fondamentali dell'architettura di un FPGA sono visibili in figura:

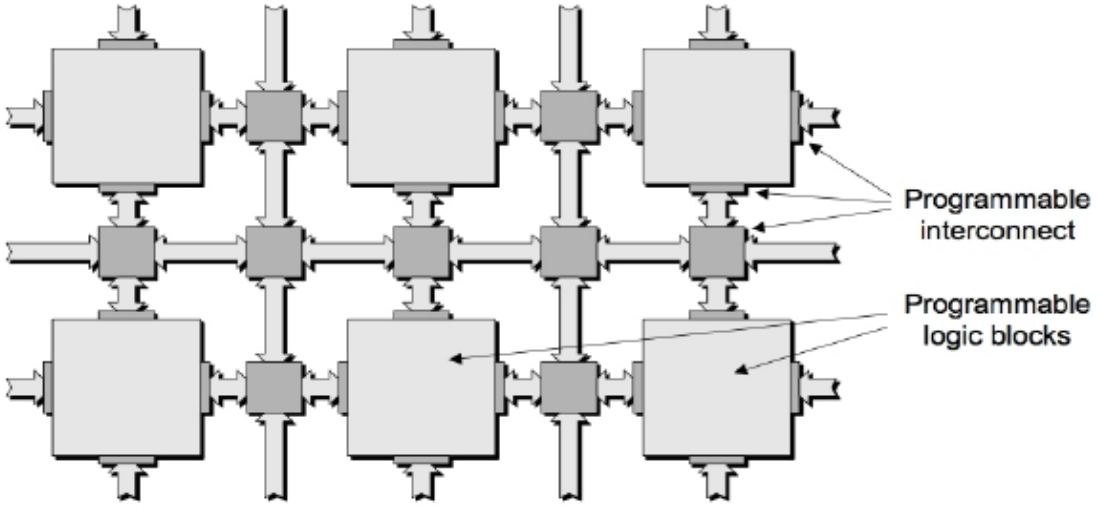


Figura 1.3 Architettura FPGA

I singoli blocchi logici CLB (*Configurable logic block*) sono moduli di logica combinatoria con uno o più registri la cui funzione è programmabile. Essi sono inseriti tra le matrici di commutazione e collegati alle loro linee. Le matrici permettono di determinare a piacere le connessioni fra i terminali, in questo modo è possibile realizzare qualsiasi circuito.
Un CLB è composto da due slices (o più) ognuna delle quali ha due celle logiche LC (*Logic cell*) le quali contengono principalmente una LUT (*LookUp Table*), un registro (Flip-Flop o Latch) e un multiplexer.

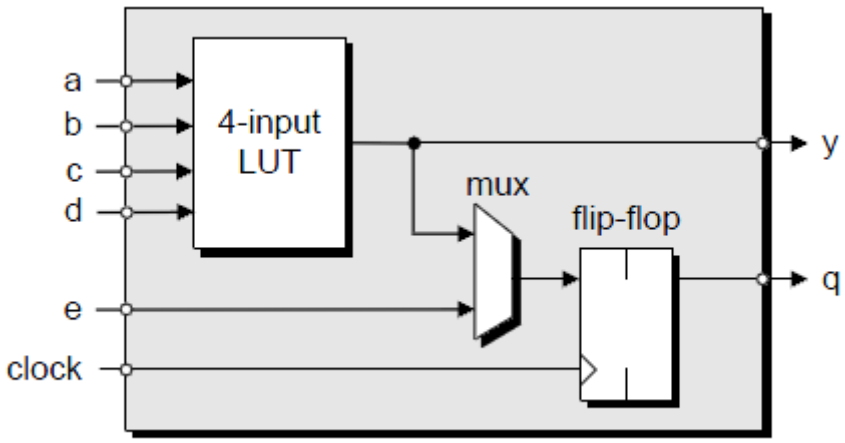


Figura 1.4 Logic cell

Un FPGA contiene questi blocchi in gran numero, i quali possono svolgere diverse funzioni.
Nel blocco logico rappresentato in figura 1.4, ogni LUT può realizzare una qualsiasi funzione logica a quattro ingressi e una uscita.

I vantaggi fondamentali degli FPGA sono:

- Si compera il componente finito e lo si programma sul campo;
- La progettazione è assistita da strumenti semi-automatici;
- Terminato il progetto, la programmazione del componente richiede poche secondi;
- Il componente può essere riprogrammato.

Gli svantaggi dell'utilizzo della tecnologia FPGA sono:

- Utilizzo incompleto di celle ed interconnessioni;
- Prestazioni ridotte rispetto ai potenziali della tecnologia;
- Costi non competitivi per produzioni in grandi numeri.

2.2.1 Nexys 3

L'FPGA utilizzato in questo progetto appartiene alla famiglia Spartan 6 della Xilinx. Come tutti gli FPGA della Xilinx presentano una struttura base comprendente: Slices raggruppate in CLB, IOB (*Input-Output Block*), interconnesioni riprogrammabili, risorse di memoria, moltiplicatori, buffer di clock globali e logica boundary scan.

La scheda chiamata "Nexys 3" è prodotta dalla Digilent e monta un FPGA Xilinx Spartan 6-XC6LX16. Le caratteristiche dell'FPGA in questione sono:

- 2278 Slices ognuna delle quali contiene 4 LUT a 6 ingressi e 8 Flip-Flop;
- 576 Kbits di Ram veloce;
- 2 celle di clock (quattro DCM (*Digital Clock Manager*) e due PLL (*Phase Loked Loop*));
- 32 DSP (*Digital Signal Processing*) Slices;
- < 500 MHz velocità di clock;

Nella figura 1.5 è riportato lo schema a blocchi di tutte le proprietà della scheda in questione con i diversi dispositivi di I/O forniti come ingresso o uscite al FPGA.

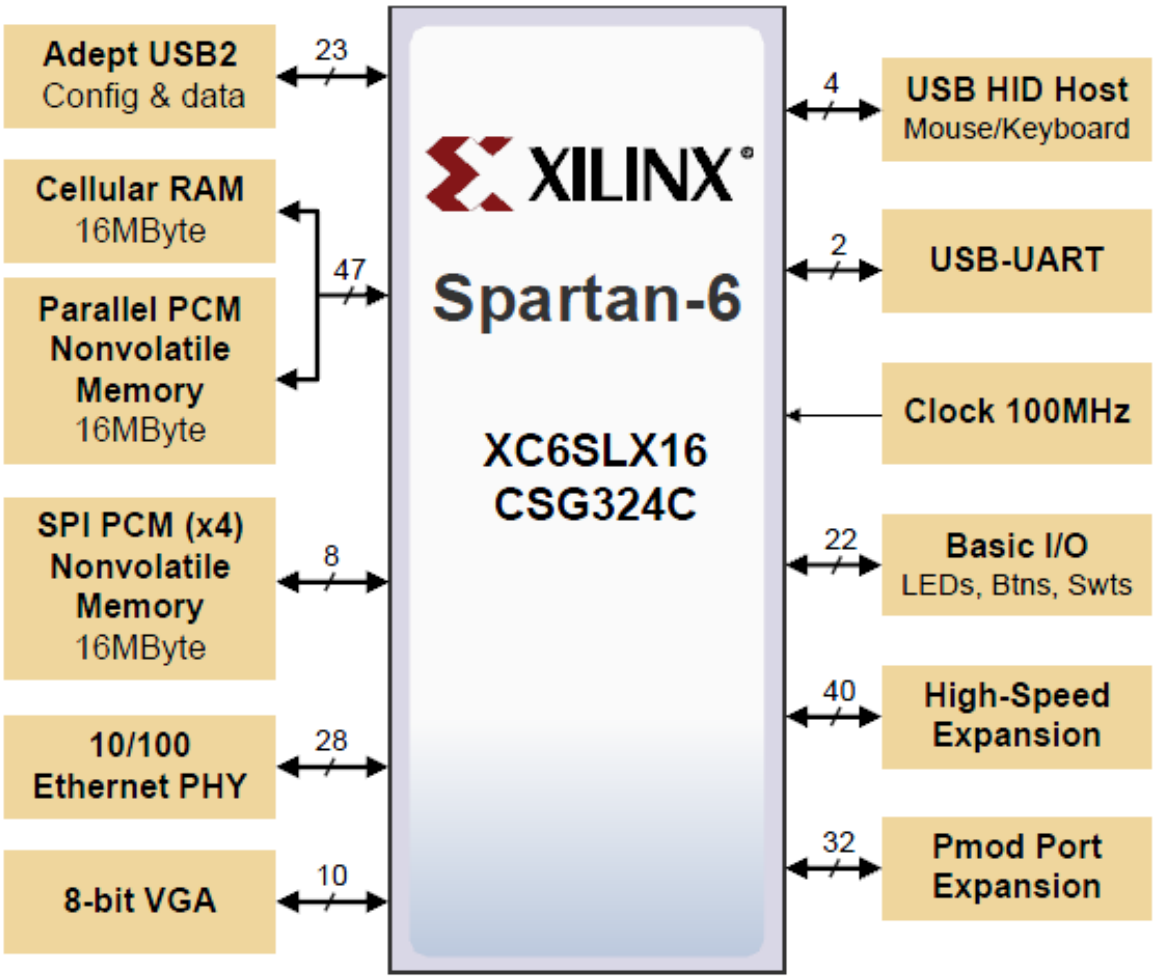


Figura 1.5 Schema a blocchi

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

- Generatori di clock a 100 MHz tramite oscillatore CMOS;
- Porta USB-HID Host per il collegamento di mouse o tastiera USB;

- Porta VGA a 8 bit per il collegamento del monitor a 256 colori.

Nella figura 1.6 è riportata un'immagine della board "Nexys 3" usata nel progetto:

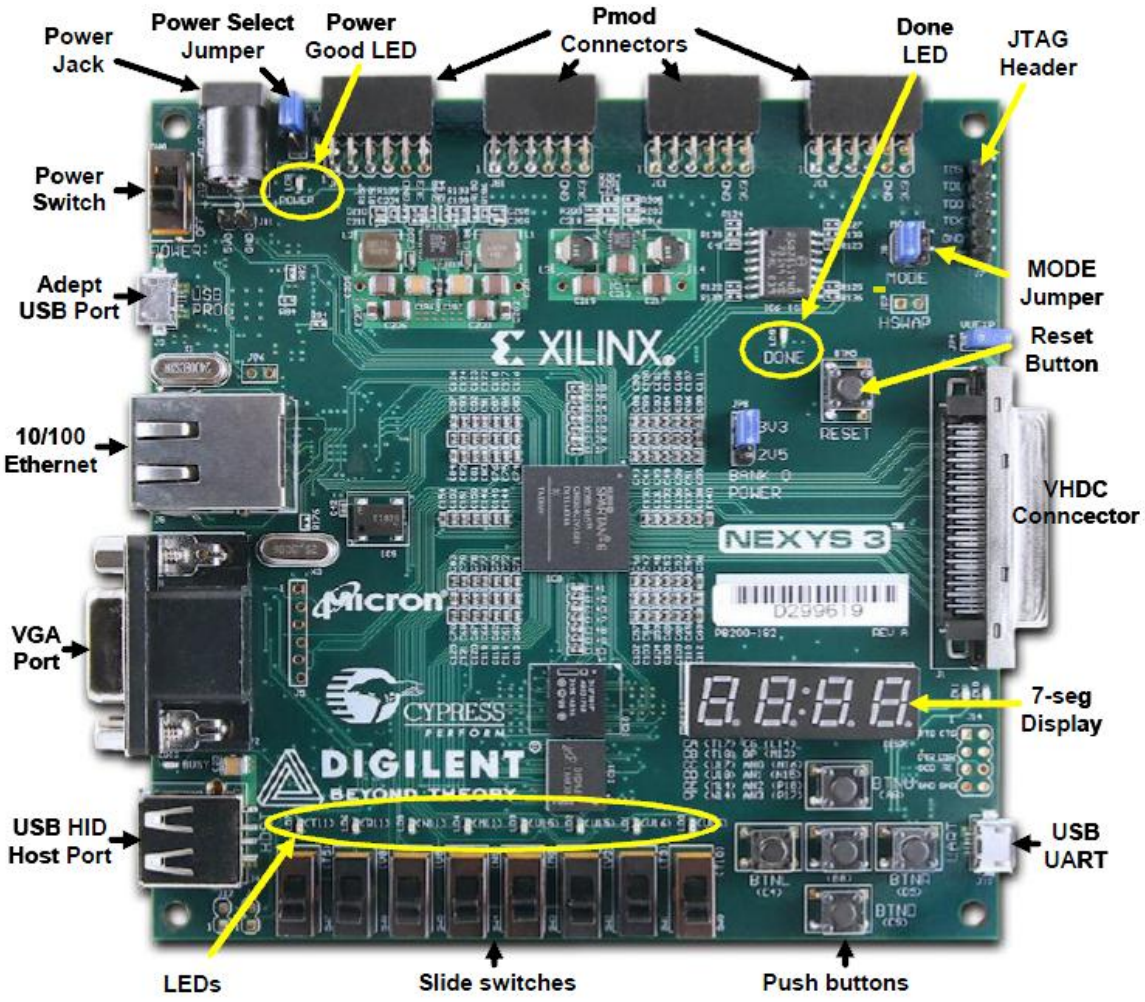


Figura 1.6 Vista Board

Nella figura 1.7 sono riportate le connessioni I/O di base, è stato usato:
un solo ingresso il pulsante (B8) per resettare il mouse e 3 uscite a Led, LD0, LD1 e LD2 per i pulsanti destro centrale (rotellina) e sinistro rispettivamente.

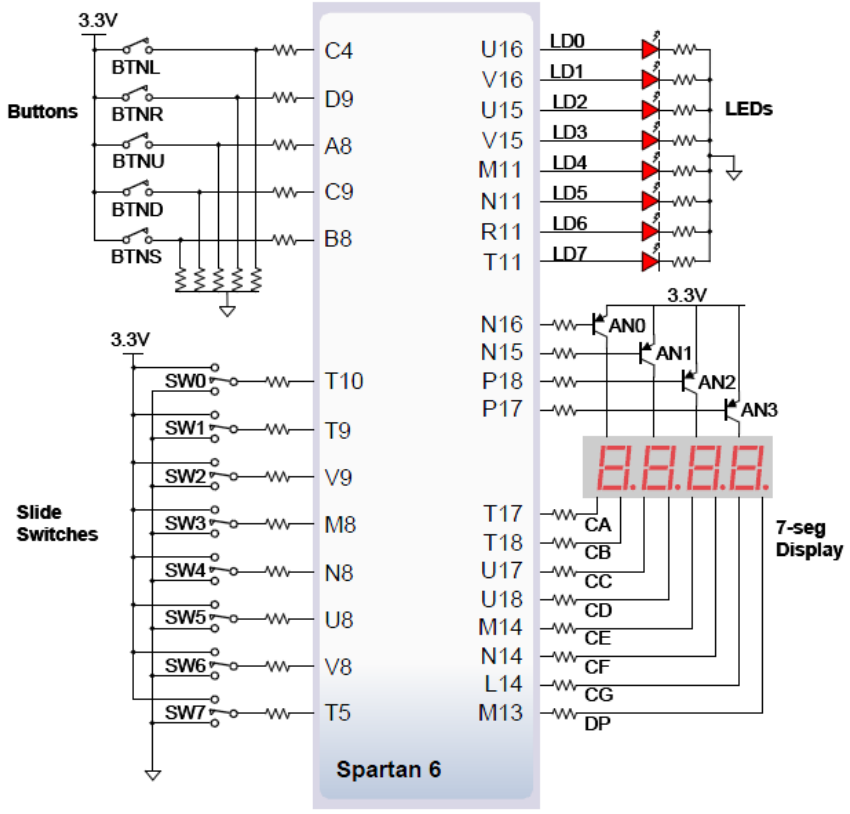


Figura 1.7 Pulsanti-Deviatori-Led

Nella figura 1.8 è riportato lo schema di connessione per la porta VGA la quale permette di visualizzare sullo schermo immagini con risoluzione 640x480 ad una frequenza di refresh di 60 Hz a 256 colori. Le uscite dell'FPGA sono digitali e assumono solo due valori 0 Volt e 3.3 Volt. Grazie alle resistenze e ai 75 Ω di terminazione del monitor, otteniamo valori intermedi contenuti nello standard VGA: 0 ÷ 0.7 Volt. I due segnali HSYNC e VSYNC utilizzano lo standard TTL.

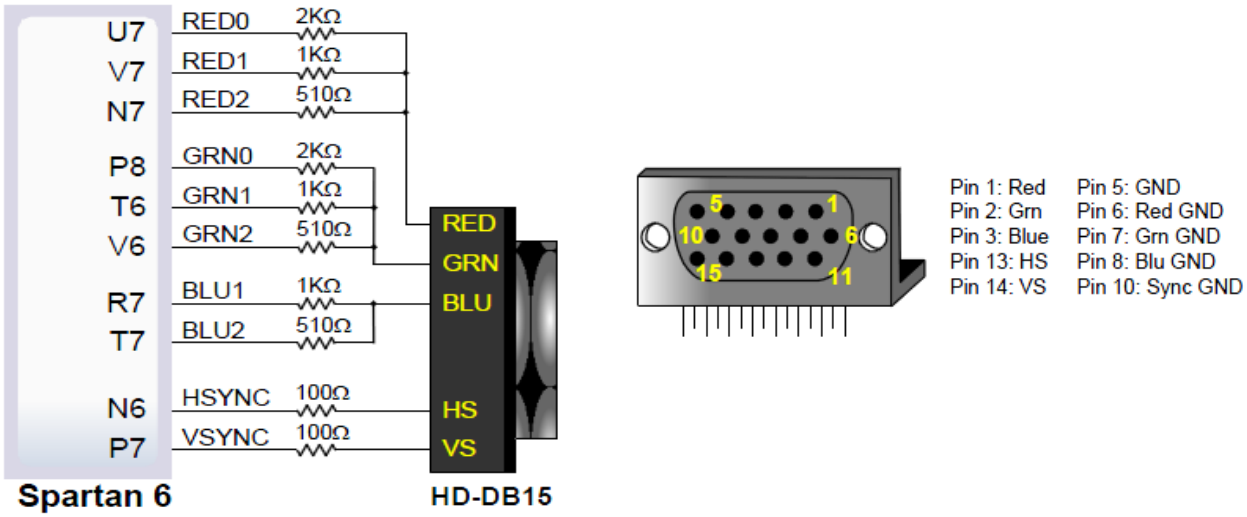


Figura 1.8 Porta VGA

Nella figura 1.9 è riportato lo schema della porta USB HID Host la quale oltre la possibilità di programmare la scheda permette di collegare al connettore J4 un Mouse (usato nel progetto) o una tastiera. Il controllo di tali periferiche e una precisa gestione del protocollo PS/2 avviene attraverso un microcontrollore PIC24FJ192 della Microchip opportunamente programmato dalla Digilent Inc. per tale scopo.

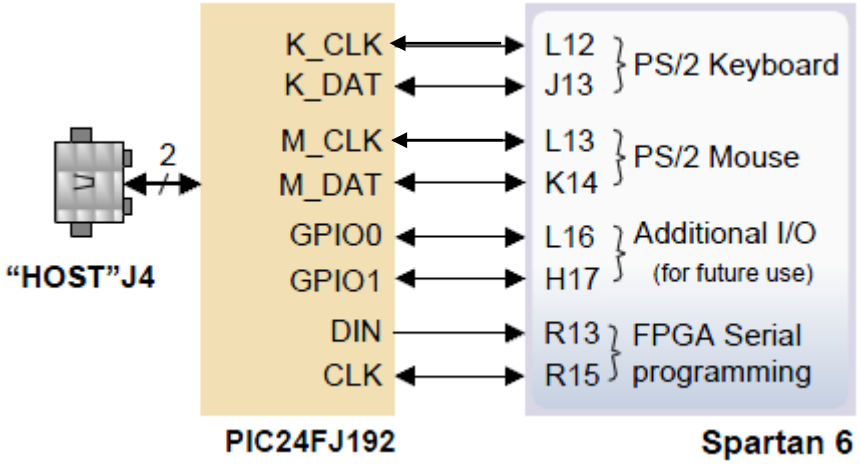


Figura 1.9 Porta USB HID Host

2.3 Linguaggio VHDL

I linguaggi descrittivi HDL (*Hardware Description Language*) nascono negli anni '80 con lo scopo di documentare progetti hardware complessi. L'esigenza era essenzialmente quella di utilizzare una metodologia standard per descrivere il comportamento dei sistemi elettronici.
Questi linguaggi sono molto simili ai linguaggi software, ma contemplano costrutti atti alla definizione di architetture e di funzionalità tipiche dell'hardware, come la sincronizzazione ed il parallelismo dei processi.
Se inizialmente questi linguaggi sono strutturati al solo scopo di documentazione ben presto essi vengono utilizzati anche per la simulazione al fine di verificare il corretto comportamento del sistema.
L'aumento degli anni della complessità dei sistemi, permessa anche dal progredire incessante della tecnologia, ha reso sempre più stringente l'esigenza di automatizzare al massimo le procedure di sintesi. Tecniche ed algoritmi sempre più sofisticati hanno permesso di sviluppare CAD che avessero capacità di tradurre descrizioni ad alto livello di sistemi elettronici in architetture che facessero riferimento direttamente a blocchi logici di libreria.
Naturalmente il tipo di descrizione di sistemi, al fine della loro successiva sintesi, non può più prescindere dall'implementazione e la sintesi del linguaggio deve rispettare alcuni vincoli precisi che ne rendano possibile l'implementazione, primo tra questi la necessità che il sistema sia sincrono. La tipologia di descrizione adatta ad essere sintetizzata è la descrizione RTL (*Registrer Transfer Logic*).

I linguaggi HDL attualmente più utilizzati sono il VHDL (VHSIC, *Very High Speed Integrated Cirtuits HDL*) diffuso soprattutto in Europa e il Verilog.

I campi di utilizzo dei linguaggi descrittivi HDL possono essere classificati in:

- Descrizione di sistemi al fine di effettuarne la sintesi;
- Descrizione di sistemi al fine di descriverne il comportamento;
- Descrizione strutturale;
- Descrizione del testbench;

La descrizione comportamentale di blocchi logici è effettuata prima di prevedere la realizzazione RTL di un blocco da sintetizzare, oppure dopo la sintesi per descrivere il comportamento di un modulo già sintetizzato.
La descrizione strutturale del sistema consiste in un listato che contiene istanze dei moduli e le loro interconnessioni. Ciò costituisce la *netlist* del sistema che viene utilizzata soprattutto nell'interscambio di dati tra i diversi pacchetti software.
La descrizione del *testbench*, è la descrizione comportamentale degli stimoli esterni del sistema. Questo approccio permette di descrivere i segnali in ingresso e di controllare quelli in uscita per verificare se corrispondono a quelli attesi semplificando notevolmente la fase di test.

2.4 Protocollo PS/2

Il protocollo PS/2 è un protocollo seriale, sincrono e bidirezionale sviluppato dall'IBM e ampliamente usato al giorno d'oggi per comunicare tra pc mouse e tastiera.
Esistono tre diversi connettori, USB, 5-PIN DIN(AT/XT) e 6-PIN DIN(PS2), quest'ultimi due connettori (dal punto di vista elettronico) sono perfettamente compatibili (il protocollo è sempre il PS/2, l'unica differenza sta nella disposizione dei pin).

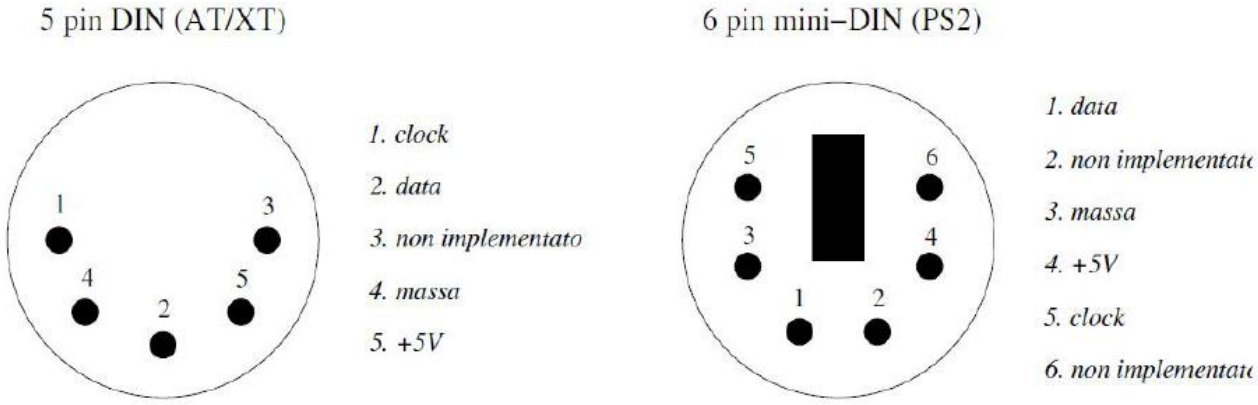


Figura 2.0 Connettori Mouse/Tastiera

Come si può notare dalla figura precedente, tutta la comunicazione si basa su un BUS formato da due linee bidirezionali: la linea di clock e la linea di dati.
Il BUS è in uno stato di attesa quando le due linee (clock e dati) sono in uno stato logico alto. Questo è l'unico stato nel quale è permesso alla periferica (Mouse o tastiera) trasmettere dati. L'host ha sempre il controllo finale sul BUS e può in ogni momento inibire la comunicazione mettendo a massa la linea di clock; alla periferica invece (solo a lei) spetta il compito di generare il clock.
I dati spediti dalla periferica all'host sono letti sul fronte di discesa del clock, viceversa i dati trasmessi dall'host alla periferica vengono letti sul fronte di salita.
La frequenza del clock deve stare all'interno del range 10,0 ÷ 16,7 KHz. Questo significa che il clock deve rimanere alto per 30 ÷ 50 µs.
Tali parametri sono assolutamente cruciali e agire su di essi è estremamente delicato.

Se l'host vuole trasmettere i dati deve innanzitutto inibire la comunicazione mettendo a massa il clock per almeno 100µs, quindi mettere a massa anche la linea dati e rilasciare il clock, questa sequenza fa capire alla periferica che sono in arrivo dei dati e quest'ultima inizia a generare il clock (vedi figura 2.1).

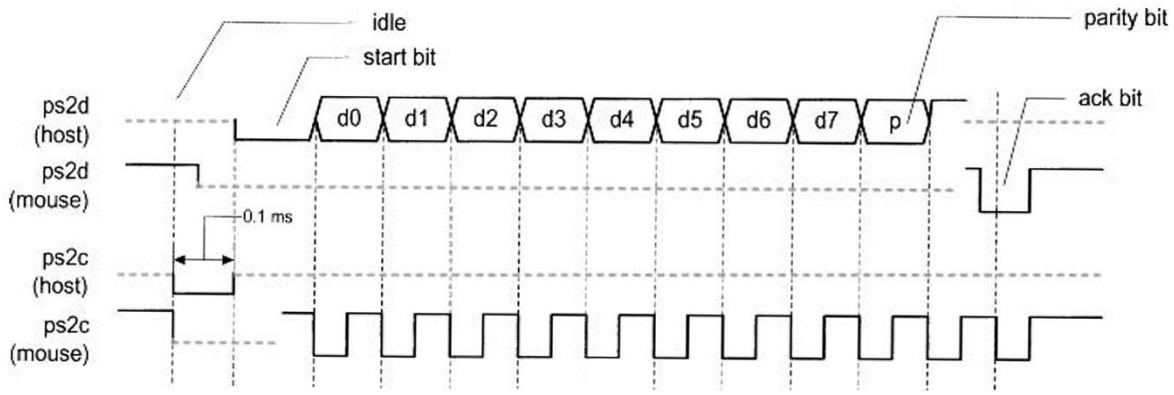


Figura 2.1 Comunicazione host-periferica

Tutti i dati vengono trasmessi in modo seriale un byte alla volta e ogni byte viene inserito in un frame composto da undici bit:

1 bit di start. Questo bit è sempre 0;
8 bit di dati (1 byte), ordinati dal numero meno significativo;
1 bit di parità;
1 bit di stop. Questo bit è sempre 1.

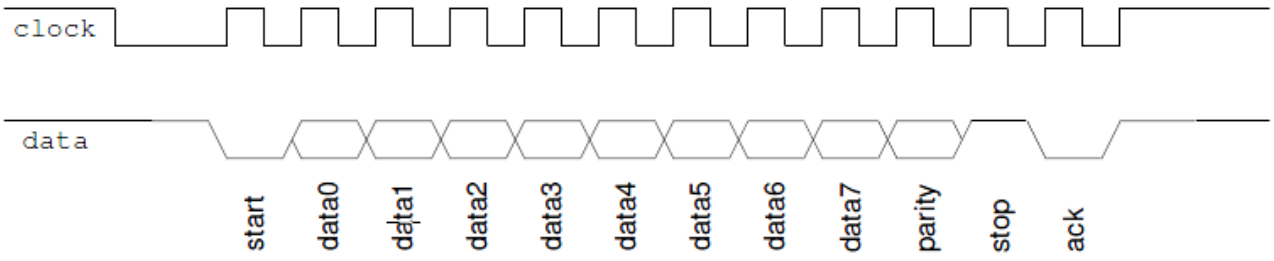


Figura 2.2 Frame e temporizzazioni

Alla fine di questo frame l'host si aspetta un *acknowledge* bit da parte della periferica (nel caso il dato sia stato correttamente interpretato), quindi rilascia la linea dati. Se l'host non rilascia la linea dati dopo l'undicesimo ciclo di clock la periferica capisce che c'è stato un errore nella comunicazione e continua a generare il clock finché la linea dati non viene rilasciata.
Quando la periferica vuole spedire un dato, come prima cosa controlla la linea di clock per assicurarsi che sia in uno stato logico alto. Se non lo è significa che l'host sta inibendo la comunicazione e la periferica deve bufferizzare i dati da spedire finché la linea di clock non viene rilasciata per almeno 50µs, quindi può spedire i suoi dati.
Per trasmettere i dati la periferica usa lo stesso frame di 11 bit visto in precedenza.

2.4.1 Protocollo PS/2 Mouse

Lo standard PS/2 prevede l'impiego di due contatori che tengono traccia dei movimenti X e Y. Questi registri contengono un numero a 9 bit in complemento a 2 per rappresentare "quantità" e direzione dello spostamento, oltre ad un altro bit per ogni asse, detto di overflow.
Essi vengono resettati ad ogni invio, a meno che non venga interrotta prematuramente la comunicazione.
Questi contatori vengono incrementati dal mouse ogni volta che rileva un movimento. Il parametro che determina di quanto debbano essere incrementati si chiama risoluzione. La risoluzione di default è di 4 conteggi ogni millimetro, ma può essere variata dall'host attraverso l'apposito comando.
Il valore dei due contatori X e Y viene trasmesso assieme allo stato dei pulsanti (centrale, destro, sinistro) e ai flag di overflow, in un pacchetto a 3 byte come raffigurato nella seguente tabella:

	7	6	5	4	3	2	1	0
Byte 1	Y overflow	X overflow	Y sign	X sign	Sempre 1	Centrale	Destro	Sinistro
Byte 2	Movimento asse X							
Byte 3	Movimento asse Y							

Il secondo e il terzo byte contengono i valori dei due contatori di movimento, che rappresentano quanto si è spostato il mouse dall'ultimo invio di dati. Infatti, questi contatori vengono aggiornati ogni volta che il mouse, leggendo i suoi sensori, rileva un movimento e vengono resettati ogni volta che il mouse invia con successo un pacchetto di dati.
Il valore contenuto in questi due byte deve essere considerato come un valore a 9 bit in complemento a 2 dove il bit più significativo è il relativo bit di segno che si trova nel byte 1 (del pacchetto di risposta) al bit 4 e 5(X sign e Y sign rispettivamente).
Ogni contatore può contare da -256 a +255, settando a 1 il relativo bit di overflow nel caso si dovesse superare questo valore.

Il modo in cui il mouse invia le sue risposte dipende strettamente alla modalità operativa in cui si trova.
Nello standard PS/2 queste modalità sono 4:

- **Modo Reset:** Il mouse entra in questa modalità all'accensione o dopo aver ricevuto un comando di reset dall'host (FF_{16}) e imposta tutti i parametri di default (sample rate = 100, risoluzione = 4 conteggi/mm, scaling 1:1 e disabilita il data reporting). A questo punto il mouse oltre al byte di *acknowledge* (FA_{16}) invia un byte per confermare il successo dell'operazione o un eventuale errore (AA_{16}, FC_{16} rispettivamente). Infine viene inviato il codice 0x00 corrispondente al ID di un mouse PS/2 standard. Dopo aver inviato l'ID di dispositivo, il mouse entra in modo Stream disabilitando il data reporting, vale a dire che il mouse non invierà nessun pacchetto di movimento finché non riceverà il relativo comando di enable ($F4_{16}$).
- **Modo Stream:** Questa modalità è la modalità di default impostata dopo un reset. In questa modalità il mouse invierà un pacchetto di dati ogni volta che riceverà un movimento sui sensori o sui pulsanti. Può essere modificata anche la frequenza con la quale avvengono questi reporting (sample rate) da un minimo di 10 ad un massimo di 200 campioni al secondo.
- **Modo Remote:** In questa modalità invece il mouse invierà un pacchetto di dati solo dopo che gli viene richiesto dall'host tramite comando specifico. Flag e contatori vengono aggiornati ogni volta che viene rilevato un movimento o un cambiamento di stato e vengono azzerati solo dopo essere stati trasmessi all'host.
- **Modo Wrap:** Questa modalità viene utilizzata generalmente solo in fase di debug e di test. Qui il mouse rispedisce indietro all'host ogni byte che riceve (*echo*) senza processarli, così da poter capire se il dispositivo e il bus funzionano correttamente.

Di seguito sono riportati i comandi (usati nel progetto stesso) riconosciuti da un mouse PS/2 standard:

- FF_{16} Reset – Il mouse risponde son un *acknowledge* (FA_{16}) e entra in modalità Reset;
- $F4_{16}$ Enable data reporting – Il mouse risponde con un *acknowledge* (FA_{16}) e riabilita il reporting. Questo comando ha effetto solo in modalità Stream;
- $F3_{16}$ Set sample rate – Il mouse risponde con FA_{16} e si mette in attesa di un altro byte che verrà impostato come nuovo sample rate. Questo valore può essere 10, 20, 40, 60, 80(50_{16}), 100(64_{16}) o 200($C8_{16}$) campionamenti al secondo ed infine il mouse risponde con un altro *acknowlrdge*;
- $F2_{16}$ Get device ID – Il mouse risponde con FA_{16} seguito dal device ID (00_{16} nel caso di mouse PS/2 standard).

Altri comandi servono per impostare il resolution, scaling ed entrare nelle altre modalità (Wrap e Remote).

2.4.2 Intellimouse

Intellimouse, ideata dalla Microsoft è diventata in assoluto la più utilizzata estensione al protocollo PS/2, tanto che ormai viene considerato un vero protocollo, chiamato il più delle volte IMPS/2.
Queste "aggiunte" (rotella di scorrimento e altri pulsanti) richiedono però un diverso impacchettamento delle risposte del mouse che diventa composto da 4 byte anziché 3.
Al momento dell'avvio qualsiasi mouse si comporta in modo standard, con le consuete risposte a 3 byte.
Per attivare le estensioni abbiamo bisogno di inviargli specifiche sequenze di comandi.
Per attivare la rotella l'host dovrà inviare in sequenza i seguenti comandi:

- Imposta sample rate ($F3_{16}$);
- 200 campionamenti/secondo ($C8_{16}$);
- Imposta sample rate ($F3_{16}$);
- 100 campionamenti/secondo (64_{16});
- Imposta sample rate ($F3_{16}$);
- 80 campionamenti/secondo (50_{16})

Completata questa sequenza, andremo ad interrogare il mouse chiedendo il suo ID ($F2_{16}$); se risponde 00_{16} continuerà a comportarsi da mouse PS/2 standard, mentre se risponde con 03_{16} ci ritroveremo con un mouse con rotellina funzionante. Da questo punto le sue risposte avranno uno schema a 4 byte come nella seguente tabella:

	7	6	5	4	3	2	1	0
Byte 1	Y overflow	X overflow	Y sign	X sign	Sempre 1	Centrale	Destro	Sinistro
Byte 2	Movimento asse X							
Byte 3	Movimento asse Y							
Byte 4	Bit di estensione del segno				Zs	Z2	Z1	Z0

Come si può notare i primi 3 byte seguono la struttura standard vista del protocollo PS/2, mentre l'ultimo byte è invece la rappresentazione numerica dell'asse Z, indica cioè di quanti scatti è stata mossa la rotella dal momento dell'ultima lettura. È un valore a 4 bit in complemento a 2 dove il bit più significativo (Zs) rappresenta il segno, quindi i possibili valori sono compresi nell'intervallo -8, +7.
I 4 bit più significativi sono delle estensioni del bit di segno e hanno lo stesso valore di Zs (per qualche ragione di compatibilità).

Per attivare anche il quarto e quinto pulsante dovremmo inviare una sequenza di comandi leggermente diversa dalla precedente:

- Imposta sample rate ($F3_{16}$);
- 200 campionamenti/secondo ($C8_{16}$);
- Imposta sample rate ($F3_{16}$);
- 200 campionamenti/secondo ($C8_{16}$);
- Imposta sample rate ($F3_{16}$);

- 80 campionamenti/secondo (50_{16});

Andando ora a inviare la richiesta ID ($F2_{16}$) questa volta dovremmo aspettarci come risposta 04_{16}.
Lo schema dei dati inviati dal mouse, stavolta prende questa forma:

	7	6	5	4	3	2	1	0
Byte 1	Y overflow	X overflow	Y sign	X sign	Sempre 1	Centrale	Destro	Sinistro
Byte 2	Movimento asse X							
Byte 3	Movimento asse Y							
Byte 4	Sempre 0	Sempre 0	Pulsante 4	Pulsante 5	Zs	Z2	Z1	Z0

Sostanzialmente uguale a prima, si differenzia solo nei 4 bit più significativi del quarto byte, dove i bit 7 e 8 sono inutilizzati e mantenuti a 0 e i bit 4 e 5 rappresentano lo stato dei relativi pulsanti (1 = premuto, 0 = rilasciato).

2.5 Interfaccia VGA

Verso la fine degli anni ’80 la IBM lanciò la VGA (*Video Graphics Array*), inizialmente i chip VGA vennero costruiti sulle motherboard ma visto il successo e la crescente domanda, la IBM pensò bene di creare delle schede grafiche dedicate in cui alloggiava i circuiti integrati VGA. Questo standard poteva fornire una risoluzione di 640x480 a 16 colori o di 320x400 a 256 colori; ancor oggi, lo standard VGA viene usato ed è il minimo richiesto per un moderno PC.

2.5.1 VGA e Nexys 3

Il controller VGA (integrato nel monitor) genera due rampe di comando (verticale e orizzontale) partendo dai segnali HSync e VSync che sono forniti alla porta dal dispositivo che la comanda (vedi figura 2.2), assieme ai tre segnali RGB analogici che di volta in volta danno le informazioni di colore del singolo pixel.
Si definiscono tutti i pixel uno ad uno partendo dall’angolo in alto a sinistra (0,0) fino a quello in basso a destra (479, 639), intendendo lo schermo come una matrice di pixel. Quando si è “fuori schermo” i segnali RGB (*Red-Green-Blue*) devono essere a 0. I dati dei pixel si susseguono alla frequenza di clock di 25 MHz.

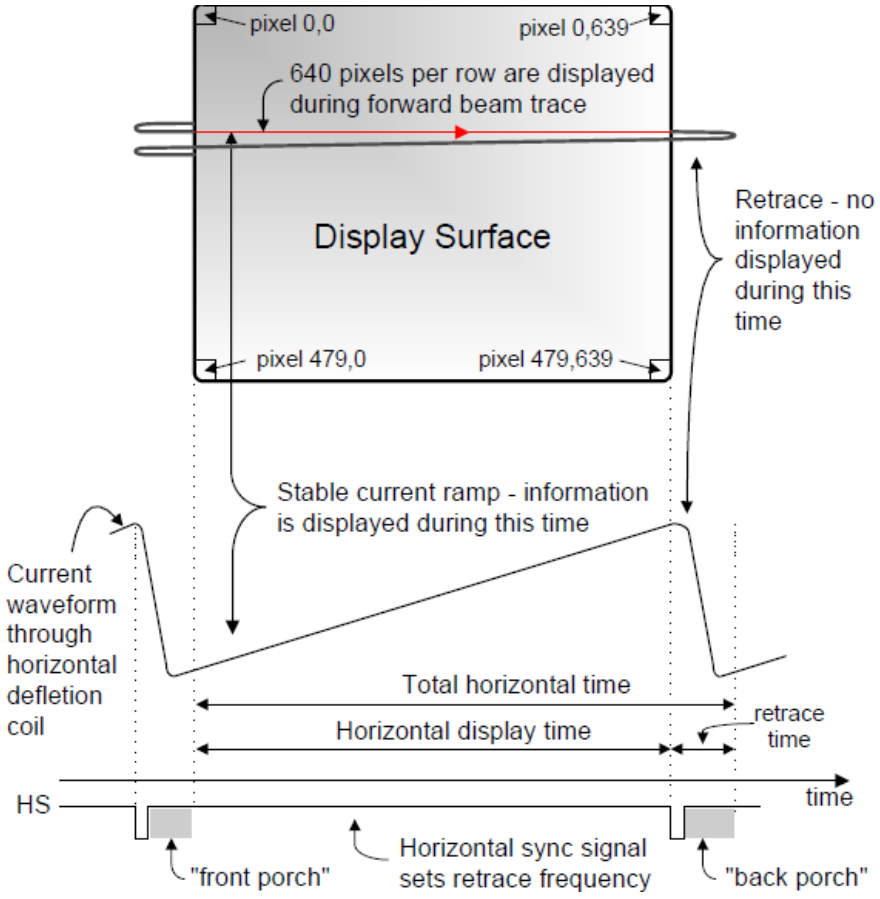


Figura 2.2 Segnali VGA

Come mostrato nella figura 2.3 oltre ai segnali RGB sono presenti i due segnali di sincronismo HSync e VSync i quali devono essere in uno stato logico alto nell'area in cui vengono disegnati i pixel (*active region*), e poi andare allo stato logico basso (e rimanervi per un tempo detto "*sync time*") nella regione in cui i pixel non vengono disegnati (*blanking region*) dopo un tempo detto "*front porch*"; i segnali di sincronismo tornano allo stato logico alto entro un tempo chiamato "back porch" prima di uscire dalla blanking region (vedi figura 2.4). La durata dei tempi "*sync time*", "*front porch*" e "*back porch*" è calcolata a partire dalla risoluzione dello schermo che nel progetto è 640x480.

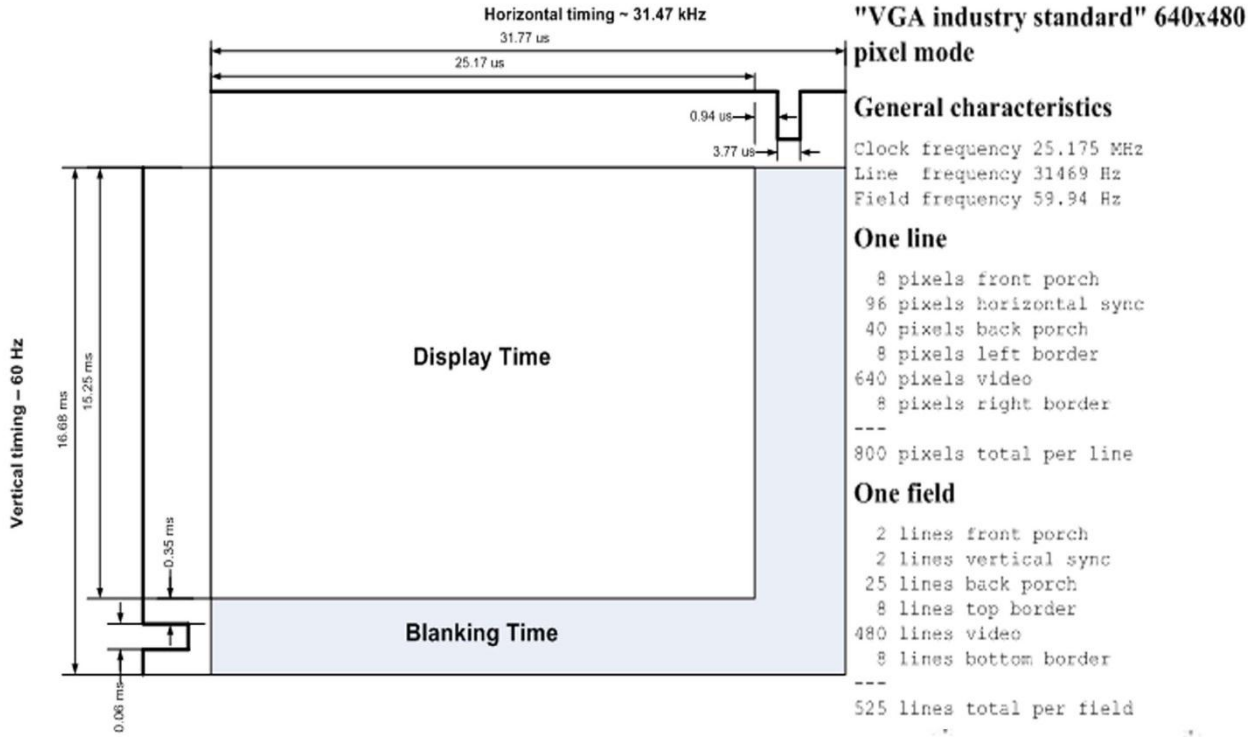


Figura 2.3 Temporizzazione VGA 640x480

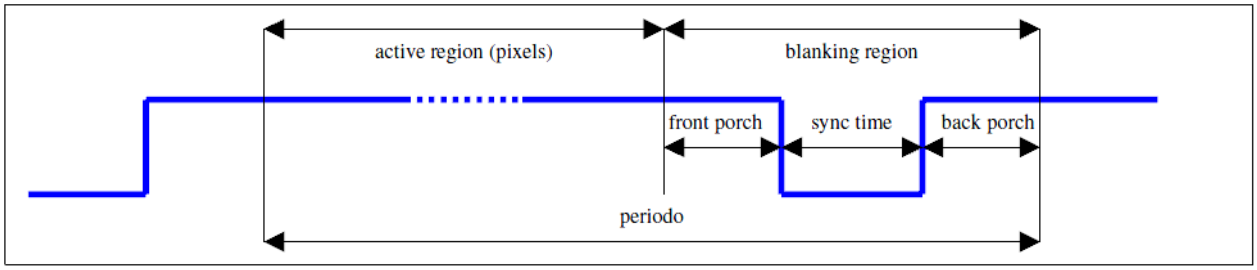


Figura 2.4 Sincronismo VGA

2.6 Tool di sviluppo

I file di configurazione dell'FPGA sono generati da tool come il software usato durante la progettazione ed implementazione fornito da Xilinx "***Ise Design Suites***" (in questo caso abbiamo usato la versione 14.5). Questo software integra tutte le risorse necessarie per progettare, modificare e generare configurazioni di FPGA utilizzando le più diverse metodologie di progetto (linguaggio VHDL e Verilog, schemi circuitali, metodi di tipo grafico o basati su blocchi).

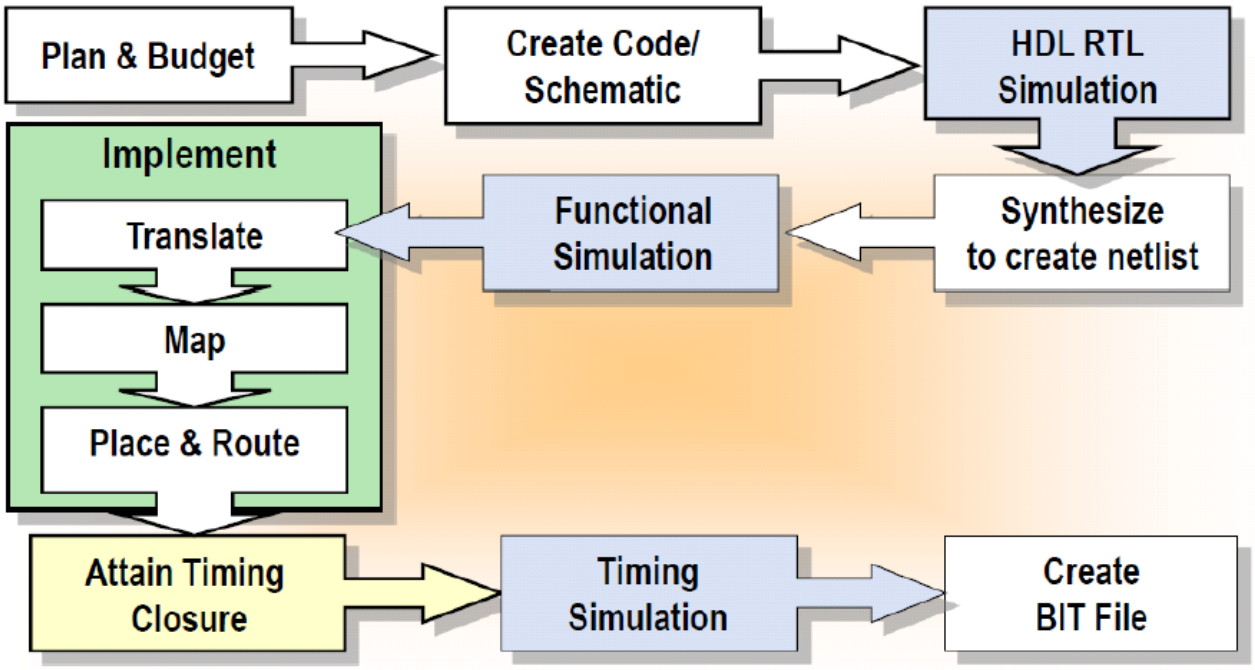


Figura 2.5 Sequenza temporale progettazione su FPGA

Nella figura 2.5 viene rappresentata la sequenza temporale e le varie fasi a cui va sottoposto il progetto prima di una effettiva implementazione sulla scheda.
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 realizzare. 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 quindi all'implementazione. Questa 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 tutti i ritardi che introduce

l'hardware della scheda. Se tutte le specifiche sono rispettate si può quindi generare il file (.bit) 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 direttamente sull'FPGA o nella PCM della scheda e che servirà a programmare tutte le strutture contenute nel FPGA.

Capitolo 3

Il Progetto

Questo è il capitolo focale dell'intero scritto, è il capitolo in cui si analizzerà punto per punto il codice scritto per la programmazione della scheda FPGA.
L'obbiettivo dei precedenti capitoli è appunto quello di fornire gli strumenti necessari alla comprensione del lavoro svolto, trattato appunto nel capitolo 3.
In questo capitolo verrà presentata una visione generale a blocchi del progetto e di seguito verranno spiegati blocco per blocco le funzionalità dei componenti e come sono stati costruiti, seguendo una semantica bottom-up.

3.1 Sezione Mouse

Nella figura 3.1 è rappresentato il progetto dell'interfacciamento del mouse sotto forma di black box.

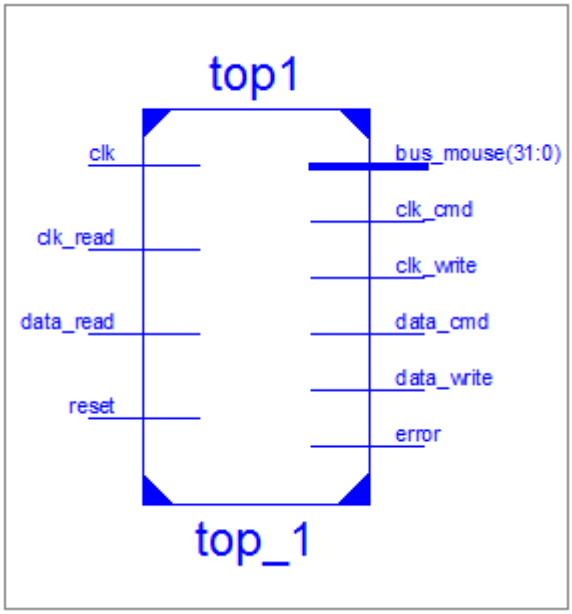


Figura 3.1 Black Box modulo Mouse(`top1`)

Analisi input/output:

- `clk`, `clk_read`: rappresentano rispettivamente il clock interno e il clock generato dal Mouse. Il clock interno della board è a 100MHz;
- `data_read`: rappresenta la linea dati (in lettura) in comunicazione con il Mouse;

- `reset`: rappresenta il pulsante B8 della scheda (tasto centrale) per avviare la procedura di reset del mouse in caso di errore;
- `bus_mouse(31:0)`: questo bus rappresenta i dati ricevuti dal mouse da spedire all'interfaccia atta alla visualizzazione video;
- `clk_cmd, data_cmd`: rappresentano il comando per il buffer tri-state sulla linea dati e sulla linea clock diretta verso il mouse;
- `clk_write, data_write`: rappresentano la linea (clock e dati) in scrittura verso il Mouse, vengono abilitate se e solo se i rispettivi comandi (`clk_cmd` e `data_cmd`) sono ad un valore logico alto;
- `error`: rappresenta un bit di errore del mouse, questo bit viene visualizzato in un apposito led (LD7).

Codice di implementazione:

Il codice che realizza questo blocco è fatto perlopiù da inizializzazioni e dichiarazioni dei sottoblocchi di cui è composto. Al suo interno sono contenute solo le definizioni dei blocchi da collegare e i segnali per collegarli.

Andando leggermente più nel dettaglio, vediamo ora come si presenta all'interno il progetto. Nella figura 3.2 è riportato lo schema interno del modulo `top1`, è possibile visualizzare le interconnessioni tra i vari moduli realizzati.

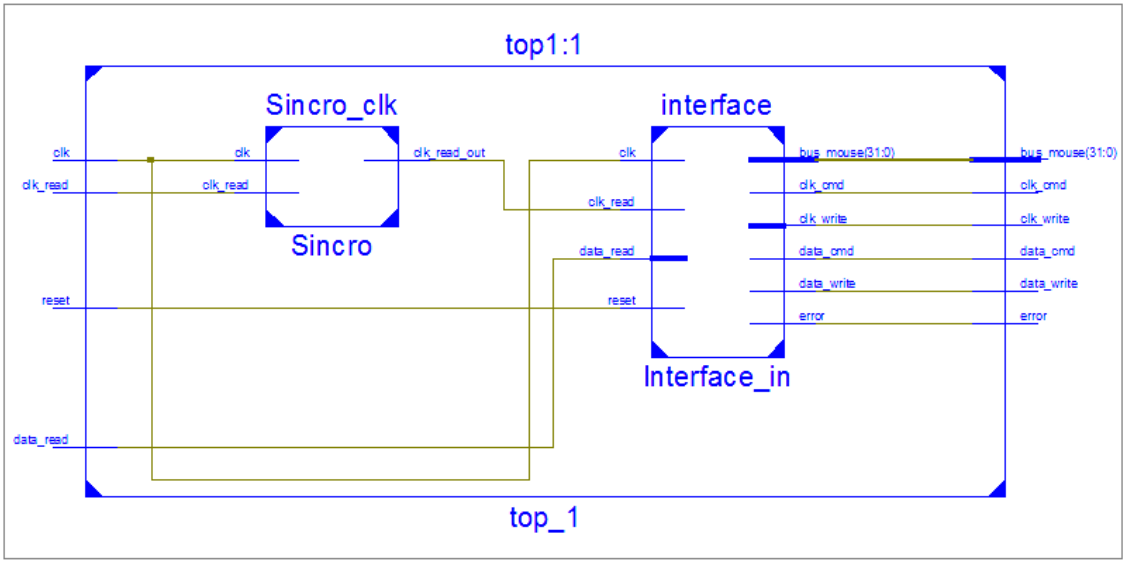


Figura 3.2 Schema interno del modulo `top1`

Andiamo ora ad analizzare nel dettaglio i singoli blocchi.

3.1.1 Sincronizzazione

Questo modulo serve per una corretta sincronia fra board e mouse, per inviare e ricevere dati. Questa *entity* ha lo scopo di rilevare i fronti di discesa del clock seriale PS2(`clk_read`) e di portare a livello logico alto la linea di uscita `clock_read_out` per un singolo ciclo di clock.

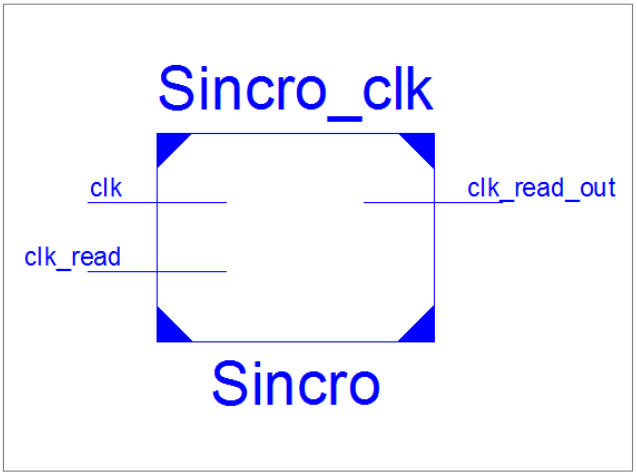


Figura 3.3 Modulo di sincronismo

Analisi input/output:

- `clk`: si tratta del clock per rilevare i fronti di discesa del clock del mouse, come si può notare dalla figura 3.2, questo segnale di ingresso è collegato al clock interno della scheda a 100 MHz;
- `clk_read`: questo segnale è il clock del mouse, generalmente a 10 ÷ 16,7 KHz;
- `clk_read_out`: questo segnale serve al blocco successivo(interface) per "capire" quando c'è stato un fronte negativo del clock del mouse; per leggere o scrivere sulla linea dati.

Codice di implementazione:

Il codice del modulo `sincro` è riportato in appendice A.
L'implementazione consiste in una serie di due flip-flop di tipo D comandati dal clock della scheda e la loro uscita viene portata all'ingresso di una AND; l'uscita del primo flip-flop viene riportata direttamente mentre l'uscita del secondo viene negata.
Nella figura 3.4 è riportato lo schema di connessione fra i vari blocchi logici e la figura 3.5 è riporta una simulazione del sincronismo.

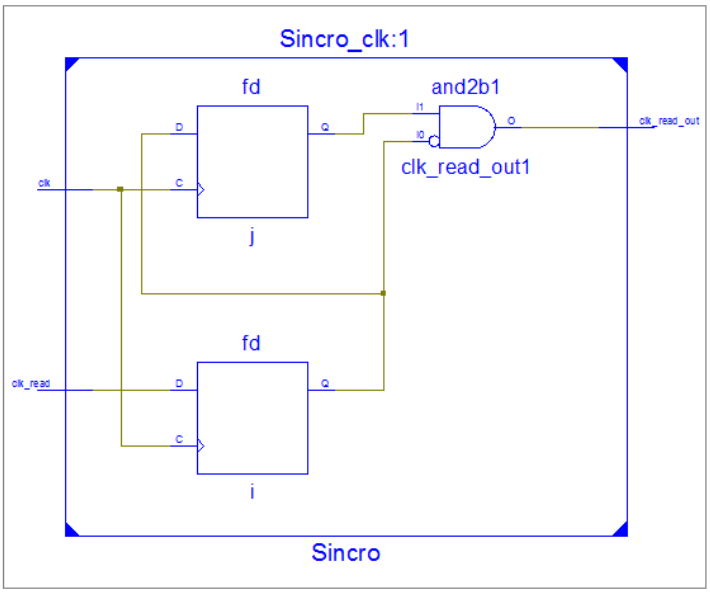


Figura 3.4 Schematico sincronismo

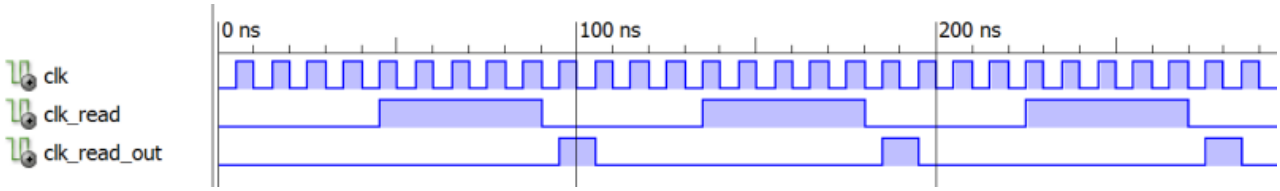
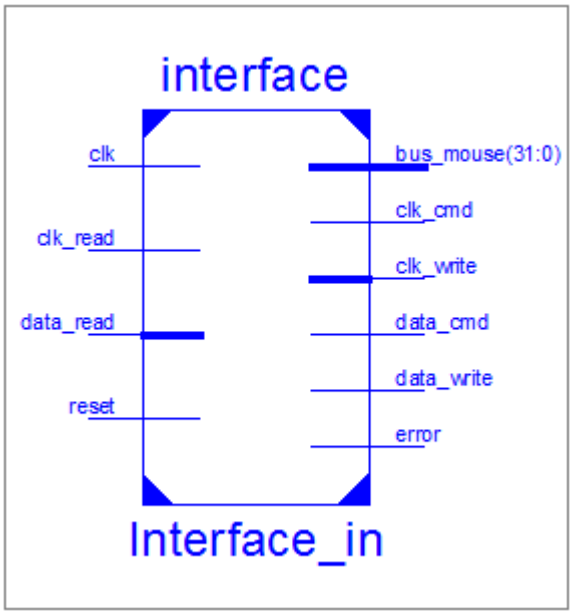


Figura 3.5 Simulazione sincronismo

3.1.2 Interface

Il blocco `interface` (interfaccia) è il *core* (nucleo) del progetto in quanto permette di "dialogare" con il mouse.



Questo blocco è realizzato tramite una FSM (*Finite State Machine – Macchina a stati Finiti*), gli stati sono: `idle`, `host_dc` (*host to device communication*), `device_hc` (*device to host communication*), `sending` e `control data init`.
Ogni stato si occupa di una precisa e ben determinata azione, in particolare `host_dc` è atto alla "*request to send*" – "richiesta di invio" di un dato ovvero alla inibizione della comunicazione, `sending` all'invio del dato, `device_hc` alla ricezione del dato (o dei dati) inviati dal mouse, `control data init` al controllo del dato ricevuto e `idle` all'attesa dell'inizio della ricezione di nuovi dati.
Nella figura 3.6 è rappresentato il diagramma a stati del modulo `interface`.

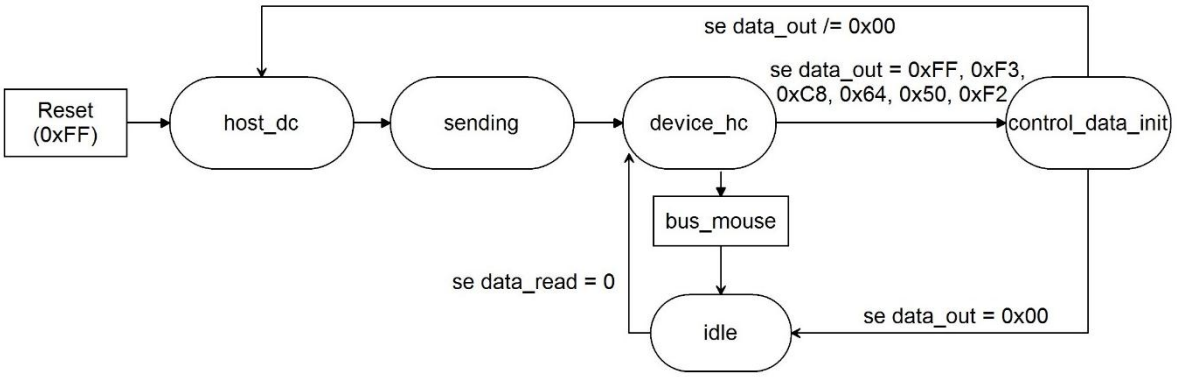


Figura 3.6 Diagramma a stati (`interface`)

Analisi input/output:

- `clk`: all'interno del modulo viene usato solo al fine di permettere alla macchina di stato di aggiornare lo stato corrente con il successivo, come si può notare dalla figura 3.2, questo segnale di ingresso è collegato al clock interno della scheda a 100 MHz;
- `clk_read, data_read, clk_write, data_write`: linea di clock e di dati del mouse rispettivamente in lettura e scrittura;
- `clk_cmd, data_cmd`: comando per il buffer tri-state per la linea clock e dati;

- `bus_mouse(31:0)`: è un bus in uscita composto da 32 bit i quali rappresentano i 4 byte di informazione del mouse (asse z, asse y, asse x e pulsanti); questa informazione serve per avere una rappresentazione video del mouse;
- `error`: è un segnale in uscita "*active high*", rappresenta un bit di errore dovuto al mouse;
- `reset`: rappresenta il pulsante B8 della scheda (tasto centrale) per avviare la procedura di reset del mouse in caso di errore.

Codice di implementazione:

Il codice di implementazione è riportato in appendice B.
Alla prima accensione oppure dopo un errore del mouse bisogna resettare (premendo il pulsante B8 della board) la periferica connessa alla scheda quindi caricare FF_{16} su `data_out` e azzerare tutti i contatori e tutti i comandi verso il mouse. Fatto ciò come si vede nel diagramma a stati (figura 3.6) la macchina a stati finiti entra nello stato `host_dc`. Di seguito è riportata una breve spiegazione con annesso diagramma di flusso degli stati.

Stato host_dc

Come si può notare nello schema a blocchi (Figura 3.7) lo stato `host_dc` è composto perlopiù da un contatore `count`, finché il conteggio non arriva a 16000 ovvero 160µs viene tenuta a livello logico basso la linea di clock. Dopo tale soglia viene rilasciata la linea di clock ponendola in alta impedenza, viene spedito il primo dato ponendo a '0' la linea dati (il primo bit è sempre 0 e l'ultimo sempre 1) e vengono azzerati i vari contatori.
Al termine si entra nello stato `sending`.

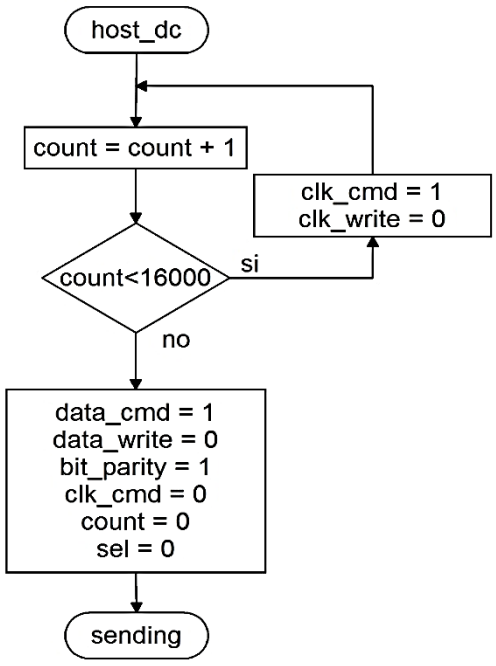


Figura 3.7 Schema a blocchi stato `host_dc`

Stato sending

Questo stato è un "invio generale" di un dato dall'host(FPGA) verso il mouse calcolando automaticamente il bit di parità, si può inviare qualunque comando infatti nel progetto verrà spedito FF_{16}, $F3_{16}$, $C8_{16}$,64_{16}, 50_{16}, $F2_{16}$ e $F4_{16}$ sempre nello stesso stato(`sending`).
Lo stato viene percorso solo quando il mouse incomincia a generare il clock, in particolare quando il modulo di sincronizzazione rileva un fronte di discesa e genera il `clk_read_out` (collegato sul segnale d'ingresso `clk_read` del modulo interface). Il rilevamento di questi fronti fa partire un contatore di tipo *integer* (`sel`), finché il contatore non conta 8 fronti di clock fa scandire e spedire al mouse tramite il segnale `data_write` uno ad uno i bit di un segnale interno (`data_out`) dal meno significativo al più significativo e calcola il bit di parità ponendo uno XOR fra il bit di parità e il bit del `data_out`. Queste due istruzioni sono:

```
data_write <= data_out(sel);
bit_parity <= bit_parity xor data_out(sel);
```

Al rilevamento dell'ottavo e nono fronte di clock viene spedito rispettivamente il bit di parità e il bit di stop (1). Insieme al bit di stop vengono azzerati vari contatori e viene preparato il mouse alla ricezione ponendo sia la linea di dati che la linea di clock in alta impedenza. Al termine si entra nello stato `device_hc`.
Nella figura 3.8 è riportato lo schema a blocchi dello stato `sending`.

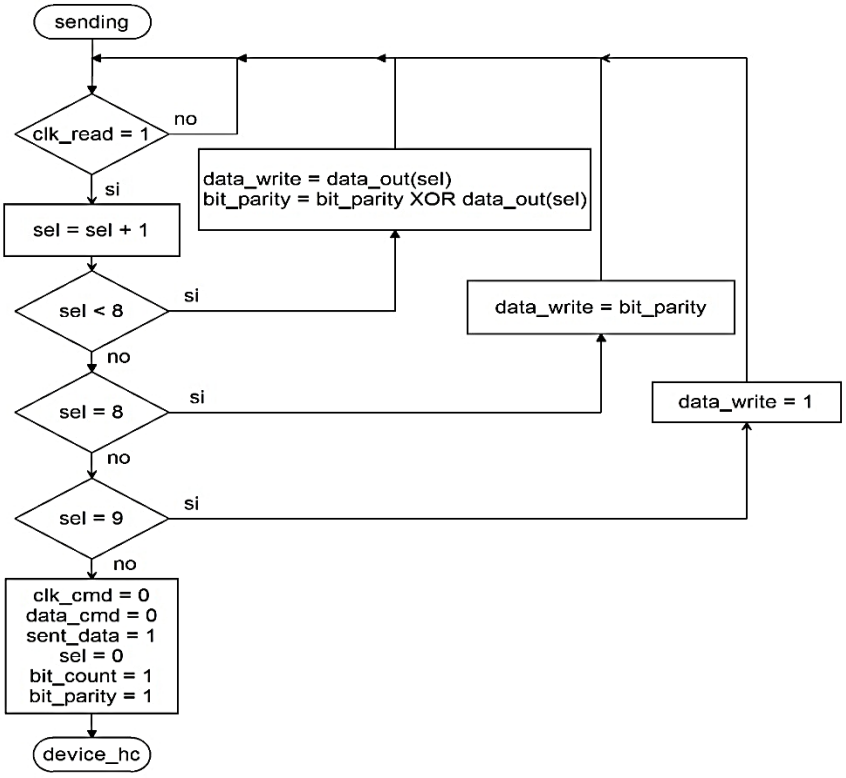


Figura 3.8 Schema a blocchi stato `sending`

Stato device_hc

Questo stato è un "ricezione generale" di uno o più dati da parte del mouse verso l'host. In questo stato viene controllato il bit di parità e il bit di stop, in caso di errore viene opportunamente segnalato attraverso il segnale `error`.
Come nello stato precedente, `device_hc` viene percorso solo quando avviene un fronte di salita sul segnale `clk_read e`, ad ogni fronte viene aumentato di 1 il contatore `bit_count`. Questo contatore conta bit per bit i dati spediti dal mouse che possono essere 11 bit se dopo aver inviato un dato (ad esempio $F4_{16}$) sto aspettando un *acknowledge* (FA_{16}), o 22 se il dato inviato è $F2_{16}$ il quale mi ritorna FA_{16} e ID mouse (standard o intellimouse) o 33 se il dato spedito è FF_{16} il quale mi ritorna un FA_{16}, AA_{16} e ID mouse (o nessun dato spedito ma il mouse di tipo standard riporta le informazioni dopo l'abilitazione dell'*enable data reporting*) oppure 44 se ho un intellimouse che sta riportando le informazioni dei vari contatori e pulsanti al suo interno.
Il bit di parità viene calcolato come nello stato precedente, l'unica differenza sta nel fatto che in questo stato devo leggere il bit per effettuare lo XOR mentre, i dati ricevuti vengono concatenati in un apposito bus composto da 8 bit (`data_reg1, data_reg2, data_reg3 e data_reg4`). Questi 8 bit per essere riempiti hanno bisogno di 8 fronti di clock come mostrato in tabella 3.9, nel progetto sono 9 ma il primo bit (bit di start) viene "eliminato" dall'ultimo bit che si concatena. Il codice per queste operazioni è:

```
    bit_parity <= data_read xor bit_parity;
data_reg1 <= data_read & data_reg1(7 downto 1);
```

clock	data_read	data_reg=xxxxxxxx
1° fronte	0	0xxxxxxx
2° fronte	0	00xxxxxx
3° fronte	0	000xxxxx
4° fronte	0	0000xxxx
5° fronte	1	10000xxx
6° fronte	0	010000xx
7° fronte	1	1010000x
8° fronte	1	11010000

Tabella 3.9 Concatenazione

Come si può notare nello schema a blocchi in figura 3.10 e 3.10.1 lo stato device_hc è composto principalmente da 4 "sottoblocchi", il primo va dal `bit_count` 0 a 11, il secondo da 12 a 22 il terzo da 23 a 33 e il quarto da 34 a 44. Questi "sottoblocchi" sono circa uguali quindi ora esamineremo solo il primo e il terzo.
Il primo "sottoblocco", dopo aver calcolato il bit di parità e aver concatenato i bit spediti dal mouse controlla se il bit di parità corrisponde con quello che il mouse

spedisce, in caso negativo lo segnala attraverso il segnale `error`. Le seguenti istruzioni sono quelle di:

- resettare il bit di parità per prepararlo per il prossimo sottoblocco quindi ponendolo a valore logico alto;
- controllare se `data_out` è uguale ad un certo valore, in caso affermativo si andrà allo stato `control_data_init` non prima di aver portato a 1 il `bit_count` e in caso negativo si continuerà nello stato attuale controllando il bit di stop.

Il terzo "sottoblocco" si differenzia solo nell'ultima parte la quale dopo aver interrogato un segnale interno (`int_mouse`) ovvero se il mouse è standard (0) o intellimouse (1). Lo stato procede con la concatenazione di tre bus da 8 bit con un byte completamente a zero e l'aggiornamento dello stato in `idle` se il mouse è standard oppure continua verso il quarto "sottoblocco" se la periferica in questione è un intellimouse.
Questa concatenazione avviene attraverso la seguente istruzione:
Nel caso di mouse standard:

```
bus_mouse <= "00000000" & data_reg3(7 downto 0) &
              data_reg2(7 downto 0) & data_reg1(7 downto 0);
```

Nel caso di intellimouse:

```
bus_mouse <= "00000000" & data_reg3(7 downto 0)
          & data_reg2(7 downto 0) & data_reg1(7 downto 0);
```

A questo punto il `bus_mouse` è così composto:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0	X_7	X_6	X_5	X_4	X_3	X_2	X_1	X_0	y_o	x_o	y_s	x_s	1	c	d	s

Nel caso di intellimouse:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Z_s	Z_s	Z_s	Z_s	Z_s	Z_2	Z_1	Z_0	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0	X_7	X_6	X_5	X_4	X_3	X_2	X_1	X_0	y_o	x_o	y_s	x_s	1	c	d	s

Figura 3.9.1 Composizione `bus_mouse`

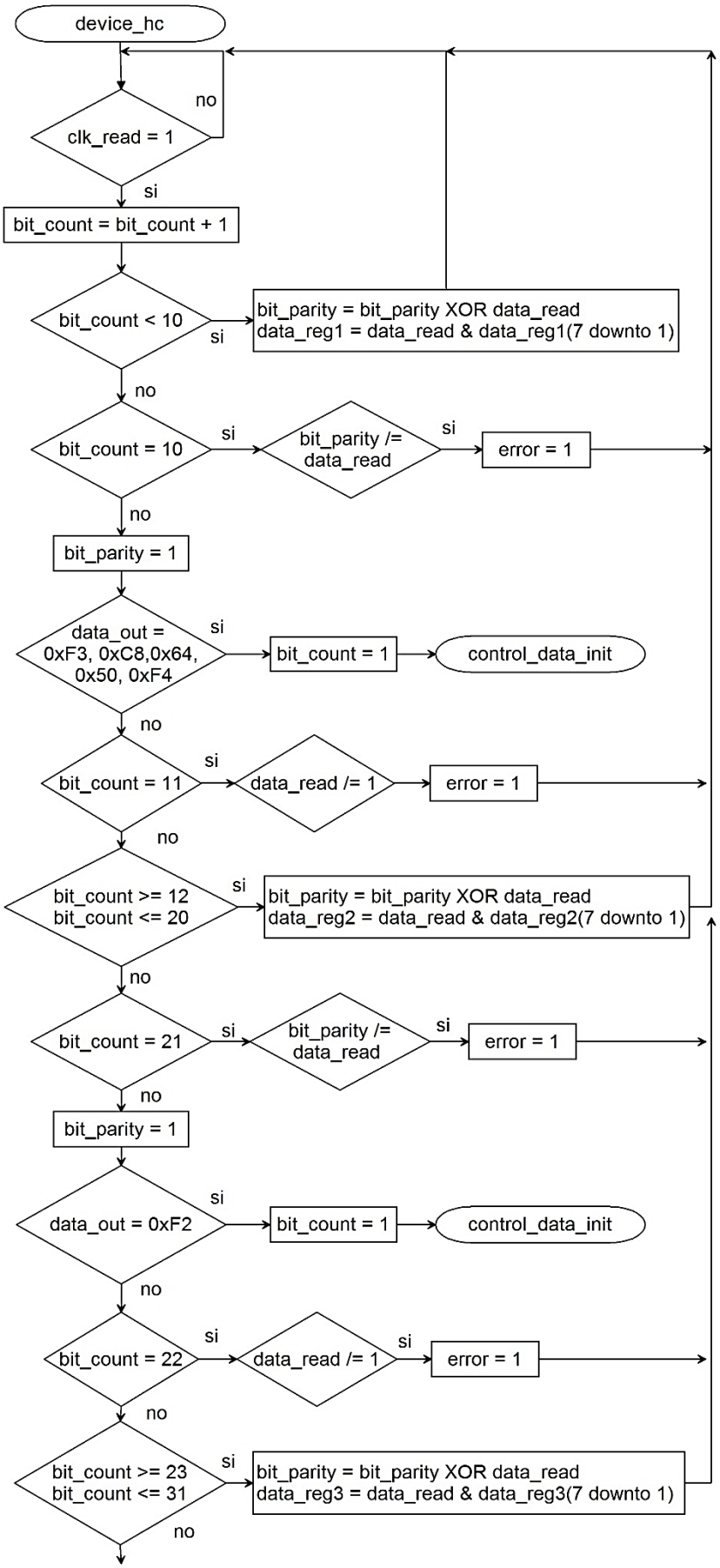


Figura 3.10 Schema a blocchi `device_hc` *(parte1)*

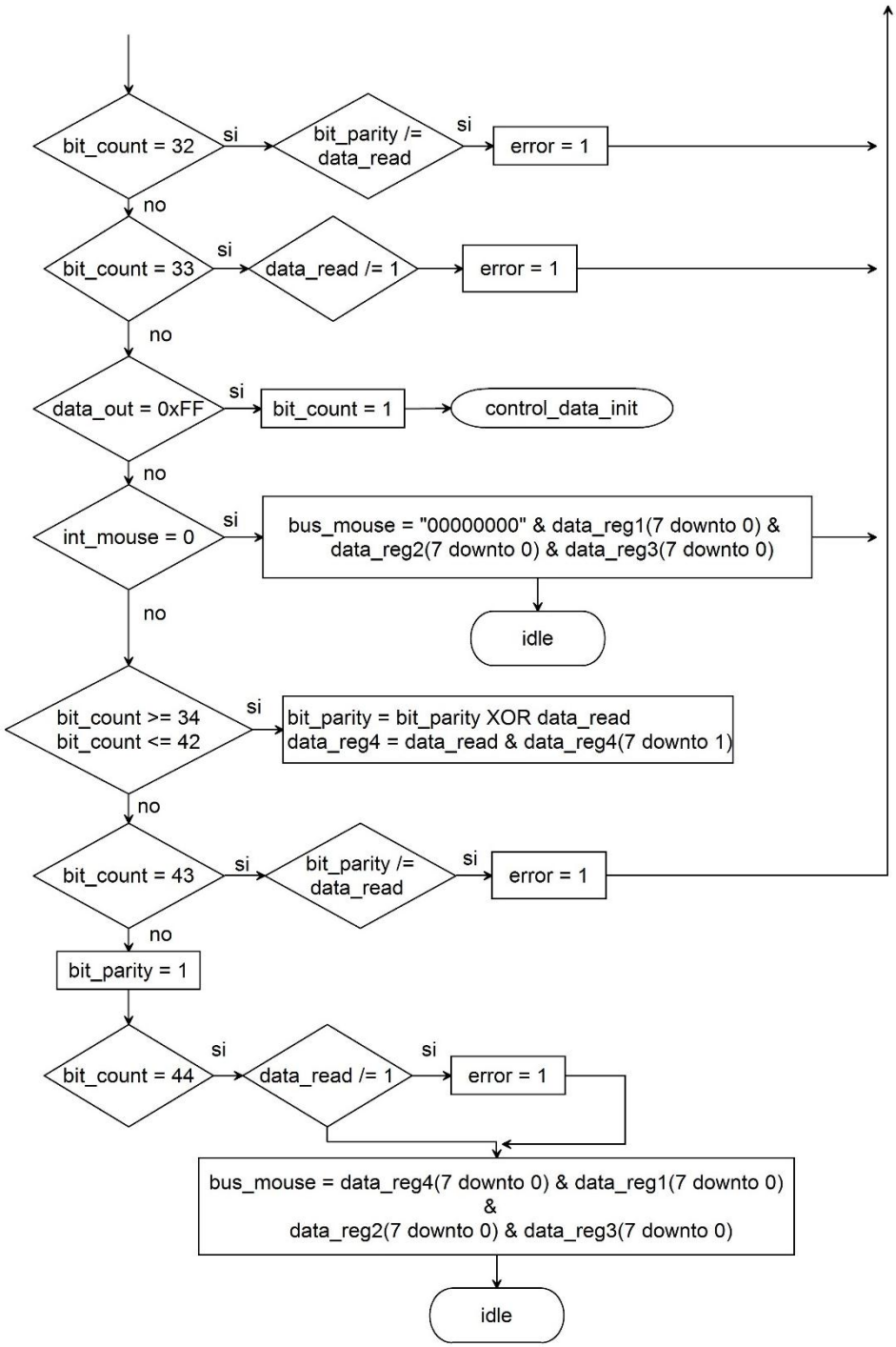


Figura 3.10.1 Schema a blocchi `device_hc` *(parte2)*

Stato idle

In questo stato si ha una prima fase di inizializzazione ovvero azzerare il contatore `bit_count`, portare al valore di default (1) il bit di parità e aver messo in alta impedenza le linee di clock e dati verso il mouse ed una seconda fase di attesa.
Questa attesa è un controllo sulla linea dati in lettura, se arriva il primo bit (bit di start a 0) allora avviene l'aggiornamento di stato in *`device_hc`*.
Nella figura sottostante è riportato lo schema a blocchi.

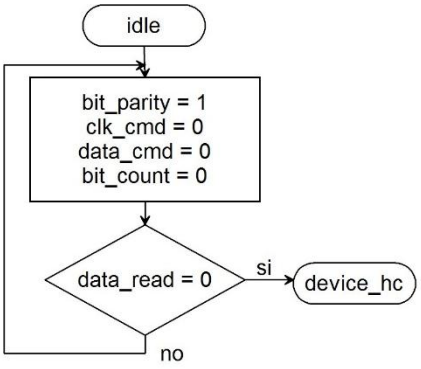


Figura 3.11 Schema a blocchi stato `idle`

Stato control_data_init

Questo stato effettua un controllo sui byte di risposta del mouse, ad ogni dato inviato bisogna controllare la risposta.
Il primo dato inviato è reset (FF_{16}), dopo il controllo dei 3 byte di risposta l'FPGA incomincia con la sequenza di dati da inviare al mouse attraverso lo stato `host_dc` e `sending` per attivare la rotellina, in sequenza:
$F3_{16}$, $C8_{16}$, $F3_{16}$, 50_{16}, $F3_{16}$ e 50_{16}.
Tutti i dati inviati dovranno rispondere con un *acknowledge* altrimenti viene portato a livello logico alto il segnale `error`.
Gli ultimi passaggi sono controllare l'id del mouse per sapere che tipo di mouse si ha collegato ($F2_{16}$) il quale risponde con 2 byte e infine si attiva l'*enable data reporting* ($F4_{16}$); si attende la risposta, si setta a zero il `data_out` così da far capire che non c'è stato nessun invio di dati e si aggiorna lo stato in `idle`.
Lo schema a blocchi è visionabile in figura 3.12 e 3.12.1.

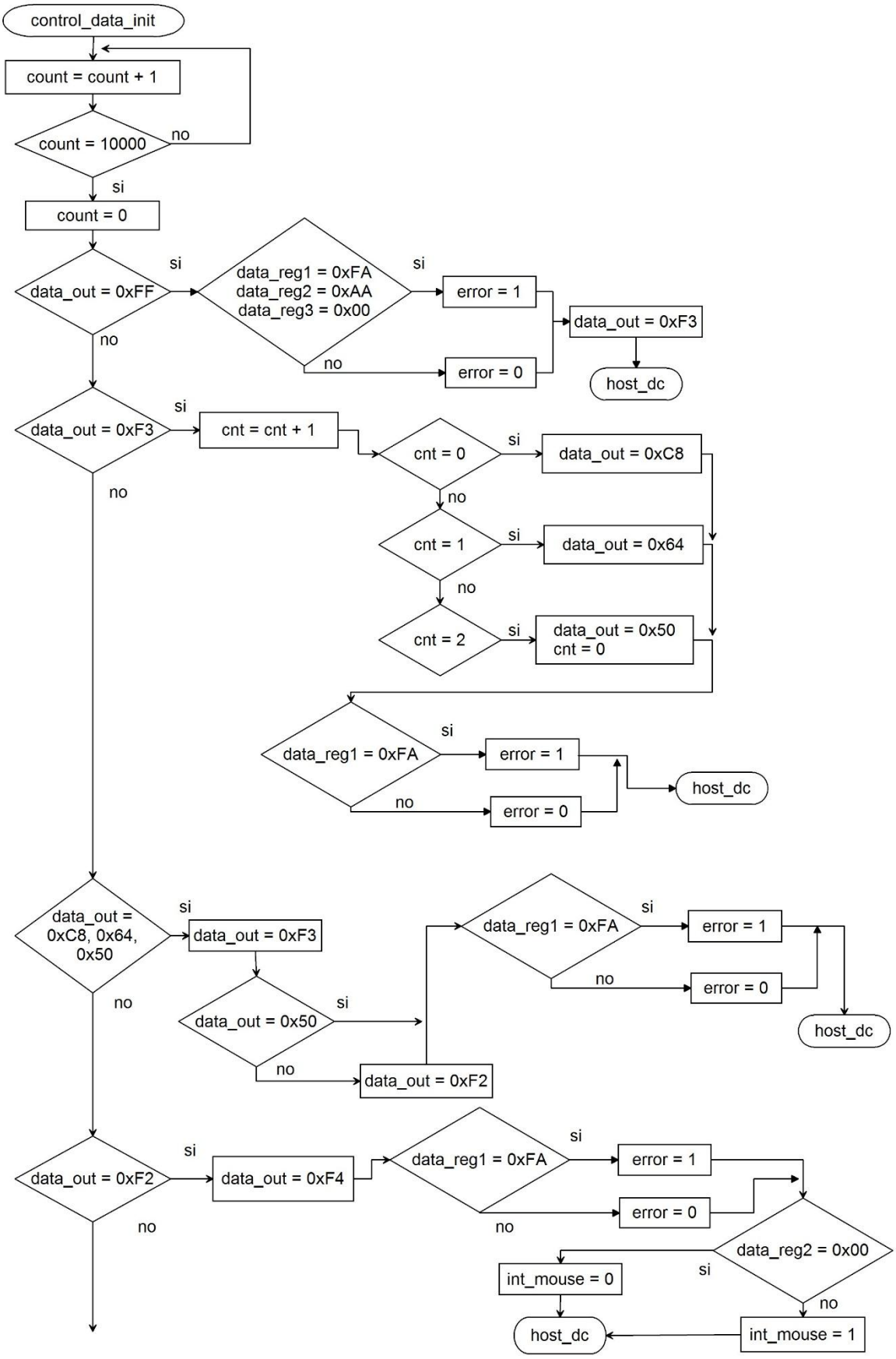


Figura 3.12 Schema a blocchi `control_data_init` *(parte1)*

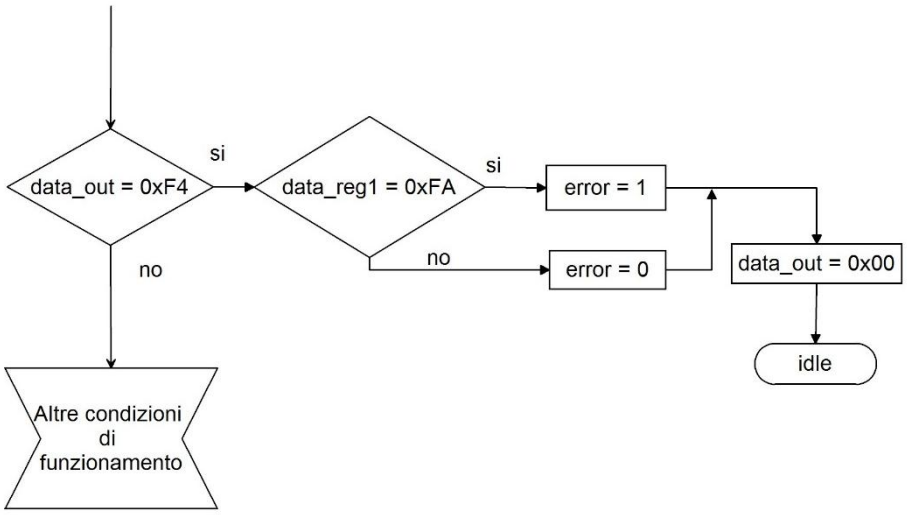


Figura 3.12.1 Schema a blocchi `control_data_init` *(parte1)*

3.2 Sezione Video

Come nella sezione precedente, nella figura 3.13 è riportato il progetto dell'interfacciamento video sotto forma di "*black box" – scatola nera.*

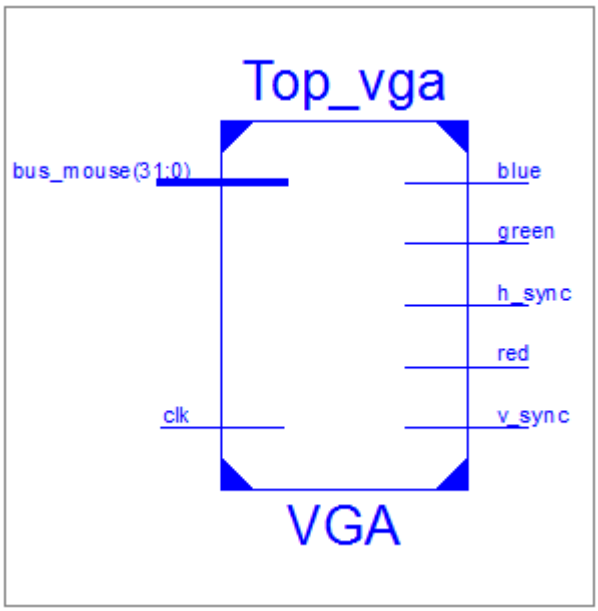


Figura 3.13 Black box modulo video (`top_vga`*)*

Analisi input/output:

- `clk:` rappresenta il clock interno della scheda a 100 MHz;

- `bus_mouse(31:0)`: questo bus riporta le informazioni acquisite dal mouse su contatori di spostamento e pulsanti;
- `blue,green,red`: questi tre segnali in uscita rappresentano l'informazione del colore dei pixel diretta sulla porta VGA, rispettivamente al pin 3 , 2, 1 della figura 1.8;
- `h_sync, v_sync`: rappresentano i due segnali di sincronismo, rispettivamente orizzontale e verticale.

Codice di implementazione:

Anche in questo caso come nel precedente (sezione 3.1) il codice che realizza questo blocco è fatto perlopiù da inizializzazioni e dichiarazioni dei sottoblocchi di cui è composto.
Andando leggermente più nel dettaglio, vediamo ora come si presenta al suo interno il progetto video. Nella figura 3.14 è riportato lo schema interno del modulo `top_vga`, è possibile visualizzare le connessioni fra i vari moduli realizzati.

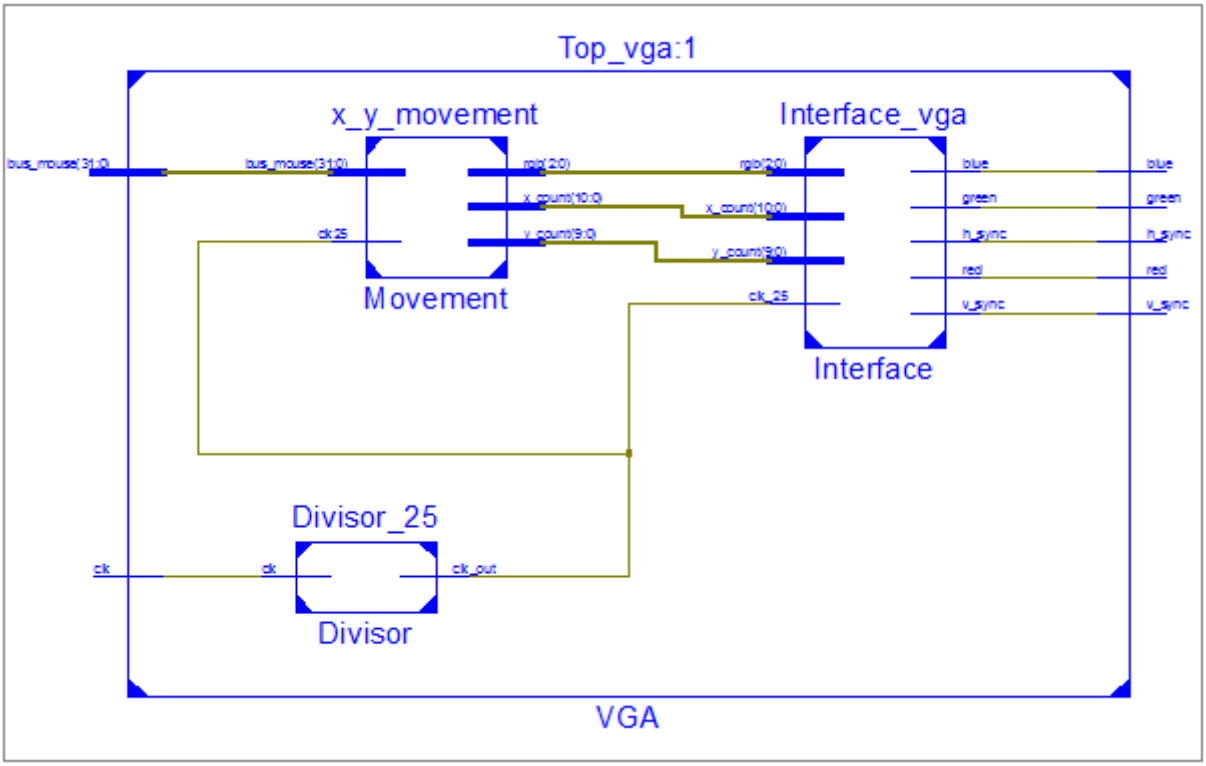
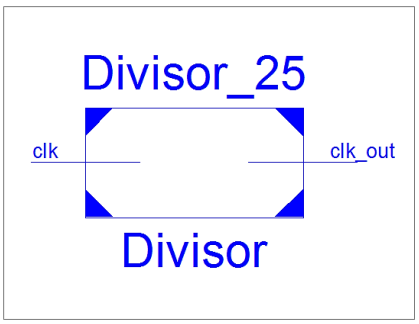


Figura 3.14 Schema interno modulo `top_vga`

Andiamo ora ad analizzare nel dettaglio i singoli blocchi.

3.2.1 Divisor

Questo sottoblocco presente all'interno di `top_vga` costituisce un divisore di frequenza, da 100 MHz interni della scheda a 25 MHz.
L'inserimento di questo blocco si è reso necessario in quanto i dati dei pixel da spedire al monitor si susseguono alla frequenza di 25 MHz (paragrafo 2.5.1).



Analisi input/output:

- `clk`: rappresenta il clock interno della scheda a 100 MHz;
- `clk_out`: rappresenta il clock a 25 MHz in uscita.

Codice di implementazione:

Il codice utilizzato per il blocco è molto semplice ed è costituito principalmente da un contatore `count` (0 ÷ 4). Se il contatore è compreso fra 0 e 1 viene tenuta a livello logico alto la linea di `clock_out` e viceversa se il contatore è compreso fra 2 e 3.
Nella figura 3.15 è rappresentato il relativo funzionamento. Si può notare che il segnale `clk` ha un periodo di 10 ns mentre il segnale `clk_out` di 40 ns i quali corrispondo relativamente a 100 MHz e 25 MHz di frequenza.

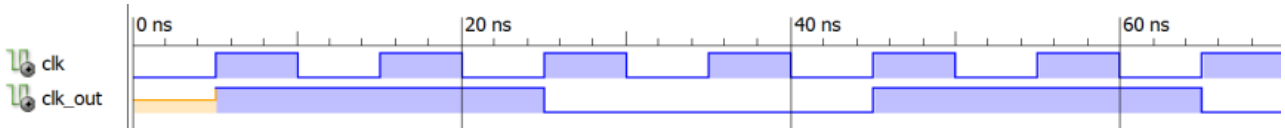
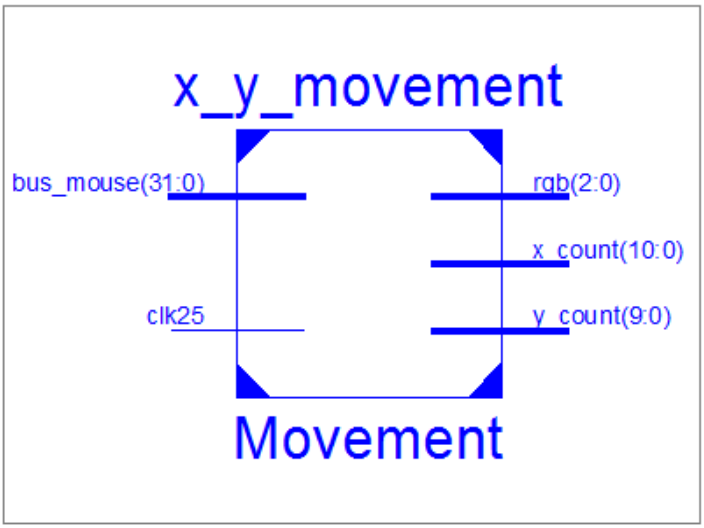


Figura 3.15 Simulazione `Divisor_25`

3.2.2 x_y_movement

Il blocco `x_y_movement` è un blocco molto importante, il quale si occupa del relativo spostamento del cursore e della rotellina del mouse. In questo blocco si potrebbe prevedere anche l'uso anche dei pulsanti della periferica in questione.



Analisi input/output:

- `bus_mouse(31:0)`: questo segnale d'ingresso riporta l'informazione inviata dal mouse relativa ai suoi contatori;
- `clk25`: segnale di clock proveniente dal blocco "*divisor_25*" il quale ha una frequenza di 25 MHz;
- `rgb(2:0)`: segnale d'uscita, "*rgb*"–"*red green blue*" rappresenta il colore di sfondo dello schermo, il quale si cambia attraverso la rotazione oraria o antioraria della rotellina;
- `x_count(10:0), y_count(9:0)`: rappresentano le coordinate in pixel sullo schermo del puntatore del mouse rispettivamente dell'asse x e asse y.

Codice di implementazione:

Il codice riportato in questo blocco descrive il movimento del cursore del mouse sul monitor, il quale si avvale di due contatori `x_position` e `y_position`.
Il primo contatore è riferito all'asse x e il secondo all'asse y, sono inizializzati rispettivamente al valore 320 e 240 pixel e possono assumere valori che vanno da 0:640 e 0:480 pixel. Questa inizializzazione è fatta per avere all'accensione il cursore del mouse in una posizione centrale dello schermo.
Il contatore riferito all'asse x (`x_position`) viene incrementato/decrementato in rifermento al bit 4 (bit di segno asse x) del `bus_mouse`, il quale subisce una variazione in base al valore riportato dal `bus_mouse` nel secondo byte (da bit 15 a bit 8 vedi tabella 3.9.1). Se questa variazione porta il contatore ad un valore maggiore di

640 o minore di 0 “fuori schermo” viene impostato il valore di “fondo schermo” che è 640 nel primo caso e 0 nel secondo.
La stessa situazione avviene per il contatore dell’asse y (`y_position`) anche se con condizioni differenti ovvero, viene incrementato/decrementato in base al bit 5 (bit di segno asse y) del `bus_mouse`, il quale subisce una variazione in base al valore riportato nel terzo byte (da bit 23 a bit 16) del `bus_mouse`. In questo caso i valori di “fondo schermo” sono 0 e 480.
Quando il bit 4 o 5 del `bus_mouse` sono ad un valore logico alto vuol dire che il bisogna convertire il byte da complemento a 2 in binario o decimale, questo è stato fatto con la seguente istruzione (nel caso dell’asse x):

```
not(bus_mouse(15 downto 8)) + '1'
```

la quale nega i bit da 15 a 8 e ci somma 1 al valore negato.
La stessa situazione avviene per la rotellina ovvero il contatore `z_position` il quale si avvale del bit 27 come bit di segno e dei bit 26:24 del `bus_mouse` come incremento o decremento.

Nella seguente tabella sono riportati i colori ottenibili con 3 bit(*rgb*):

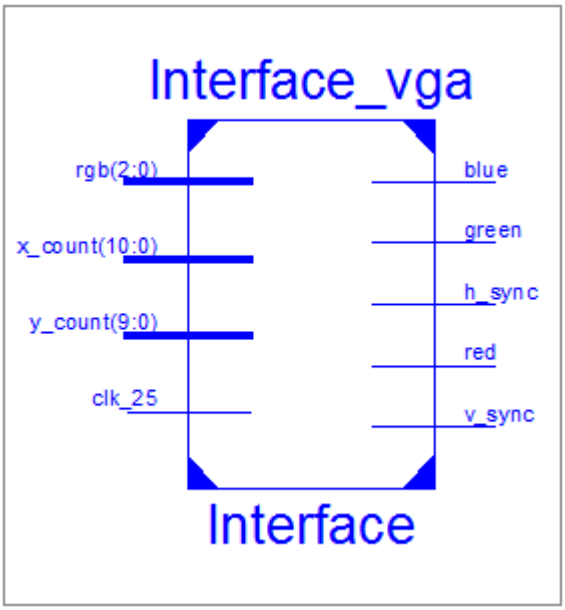
ROSSO(R)	VERDE(G)	BLU(B)	COLORE
0	0	0	Nero
0	0	1	Blu
0	1	0	Verde
0	1	1	Ciano
1	0	0	Rosso
1	0	1	Magenta
1	1	0	Giallo
1	1	1	Bianco

Tabella 3.16 Colori ottenibili con 3 bit

Nell’appendice C è riportato il codice del modulo `x_y_movement`.

3.2.3 Interface_vga

Il blocco `interface_vga` si occupa del corretto invio dei dati per i pixel dello schermo. Questo modulo scansiona tutti i pixel del monitor definendoli uno ad uno ed invia il relativo segnale di sincronismo.



Analisi input/output:

- `rgb(2:0)`: segnale d'ingresso il quale proviene dal blocco `x_y_movement` (vedi figura 3.14) il quale definisce il colore di sfondo dello schermo;
- `x_count(10:0), y_count(9:0)`: rappresentano la posizione del cursore sullo schermo;
- `clk25`: segnale di clock proveniente dal blocco `divisor_25` alla frequenza di 25 Mhz il quale scandisce l'invio dei dati per i pixel dello schermo;
- `blue, green, red`: segnali d'uscita diretti alla porta VGA i quali definiscono il colore dello schermo attraverso la loro combinazione (tabella 3.16);
- `h_sync, v_sync`: rappresentano i due segnali di sincronismo per lo schermo, rispettivamente orizzontale e verticale.

Codice di implementazione:

Per poter visualizzare sul monitor il cursore del mouse è necessario scandire uno ad uno i pixel dello schermo attraverso due contatori `v_count` e `h_count` rispettivamente dei pixel verticali e orizzontali, partendo dal pixel di coordinate(0,0) in alto a sinistra, proseguendo sulla riga fino al pixel (0,639) definendogli il colore *rgb* con le seguenti istruzioni:

```
if (v_count >= 0 and v_count <= 479) then
 red_signal <= rgb(0);
 blue_signal <= rgb(1);
 green_signal <= rgb(2);
end if;
```

Queste istruzioni operano attraverso l'uso di `v_count` ovvero, il colore di sfondo dello schermo viene definito per tutti i pixel (anche fuori schermo) ma solo alla fine viene abilitato attraverso una AND:

```
red <= red_signal and video_en;
blue <= blue_signal and video_en;
green <= green_signal and video_en;
```

Questa AND avviene tra uno dei tre segnali rgb e il `video_en` che a sua volta viene generato da un'altra AND:

```
video_en <= horizontal_en and vertical_en;
```

Questo segnale (`video_en`) viene portato a livello logico alto solo in *"active region"* ovvero solamente dal pixel di coordinate (0,0) al pixel (639,479). Quindi, il colore di sfondo viene definito solo in questa regione.

Oltre alla definizione di questi tre segnali, fino al pixel 639 viene tenuto ad un valore logico alto il segnale `horizontal_en`. Continuando sulla riga si arriva al sincronismo orizzontale che si ha tra il pixel 659 e 755 con le seguenti istruzioni:

```
if(h_count >= 659 and h_count <= 755) then
 h_sync <= '0';
else
 h_sync <= '1';
end if;
```

A questo punto il contatore orizzontale non viene azzerato finché non raggiunge il pixel di valore 799, quando ciò accade, viene aumentato il contatore verticale di un pixel cosicché da passare alla riga successiva, scandendola come la precedente.
Fino la riga 479 viene tenuto a livello logico alto il valore di `vertical_en`. Continuando sulle righe successive si ha la stessa situazione del contatore orizzontale, viene posto il sincronismo verticale a livello logico basso dal pixel 493 al pixel 494 con le istruzioni:

```
if(v_count >= 493 and v_count <= 494) then
 v_sync <= '0';
else
 v_sync <= '1';
end if;
```

Si arriva così alla fine dello schermo ponendo il reset al contatore verticale con le condizioni:

```
if(v_count >= 524 and h_count <= 699)then
 v_count <= (others => '0');
elsif(h_count = 699) then
 v_count <= v_count +1;
end if;
```

Il cursore del mouse viene definito attraverso le istruzioni:

```
if( v_count - y_count >= -10 and v_count - y_count <= 10 and -- pointer
     h_count - x_count >= -10 and h_count - x_count <= 10) then
      if((v_count - y_count >= 0 and v_count - y_count <= 0) or
           (h_count - x_count >= 0 and h_count - x_count <= 0))then
                red_signal <= '1';
                green_signal <='0';
                blue_signal<='0';
      end if;
     end if;
```

Le quali fanno uso dei segnali d'uscita del modulo `x_y_movement` (`x_count(10:0)`, `y_count(9:0)`) i quali identificano la posizione del cursore sullo schermo.
Per la definizione del cursore si è dovuto usare la libreria:

```
use IEEE.STD_LOGIC_SIGNED.ALL;
```

Poiché questa definizione fa uso dei numeri negativi.
In questo caso il cursore viene fatto attraverso l'incrocio di una riga orizzontale e una verticale di larghezza ciascuna di 2 pixel e di lunghezza 10 pixel di colore rosso.
Di seguito è possibile visionare lo schema a blocchi del modulo `interface_vga`.
L'intero codice è visionabile in appendice D.

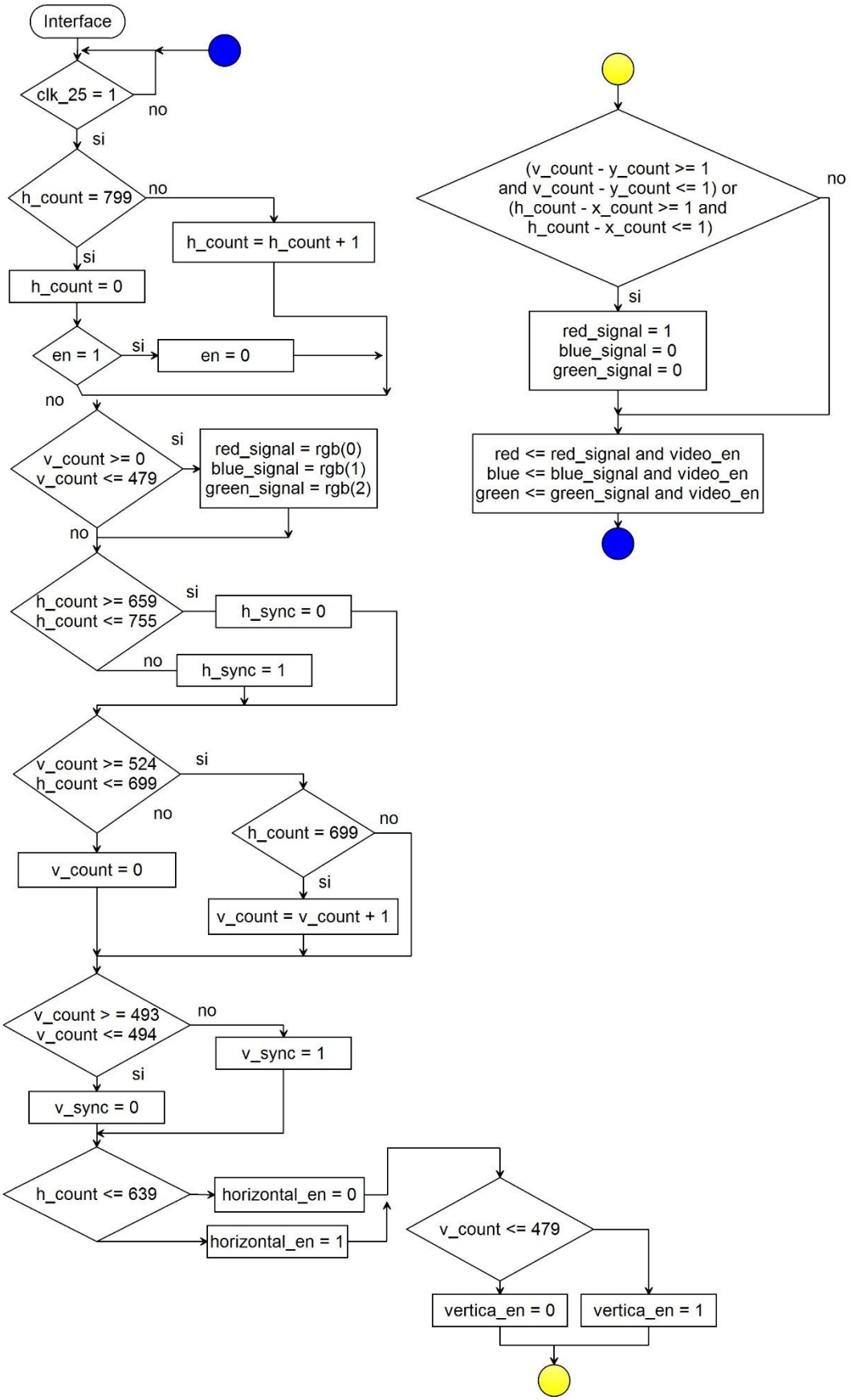
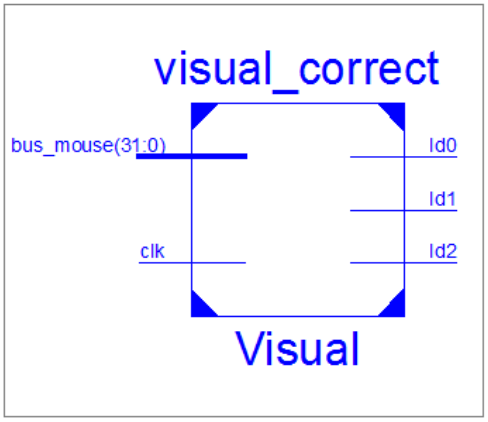


Figura 3.16.1 Schema a blocchi `interface_vga`.

3.3 Sezione visualizzazione



Questa sezione è composta da un solo blocco, `visual` il quale è predisposto a fornire ai led LD0 , LD1 e LD2 i segnali provenienti rispettivamente dal pulsante destro, centrale(rotellina) e sinistro del mouse.

Analisi input/output:

- `clk`: segnale in ingresso, rappresenta il clock interno della scheda (100 MHz);
- `bus_mouse(31:0)`: questo segnale d'ingresso riporta l'informazione inviata dal mouse relativa ai suoi contatori e ai suoi pulsanti;
- `ld0, ld1, ld2`: segnali in uscita "*active high*" atti alla corretta visualizzazione del pulsante destro centrale e sinistro del mouse.

Codice di implementazione:

Il codice del blocco `visual` non è altro che la definizione di tre flip-flop di tipo D sensibile ai fronti di salita del clock. Nella figura 3.17 è riportato lo schema interno del blocco *`visual`*.

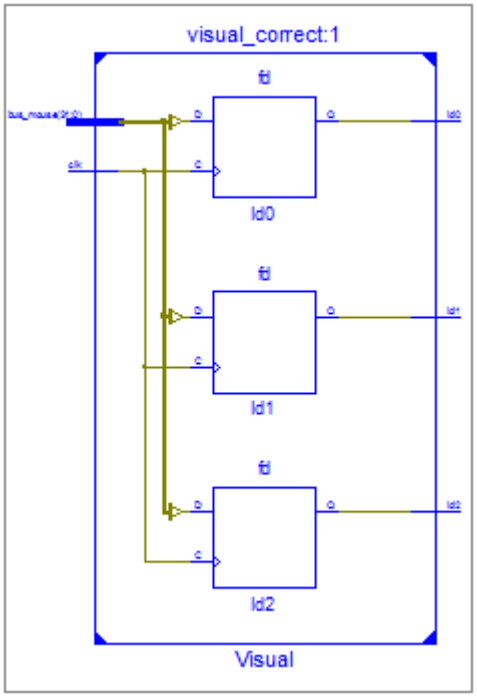


Figura 3.17 Schema interno `visual`

Di seguito in figura 3.18 è riportato un esempio di funzionamento del flip-flop D.
La figura 3.18 mostra il funzionamento di un flip-flop D attivo sul fronte di salita del clock.
Il valore di Q è inizialmente indeterminato. In corrispondenza del primo fronte di salita del clock risulta D=0, per cui l'uscita Q si porta a livello logico basso. Le successive variazioni di D, sia per clk=0 che per clk=1, sono ignorate dal flip-flop che è sensibile soltanto alle transizioni $0 \rightarrow 1$ del clock. Ad esempio, in corrispondenza del secondo fronte di salita del clock risulta D=1 e l'uscita Q si porta a livello logico alto.

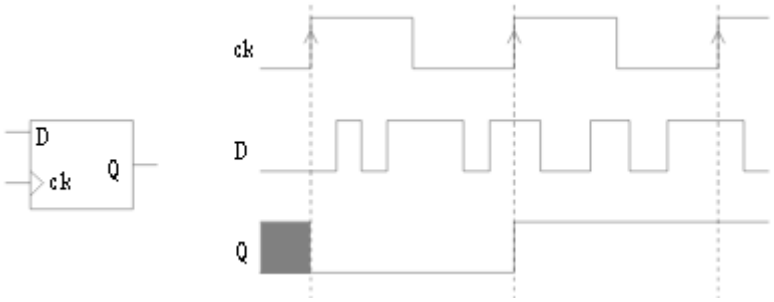


Figura 3.18 Flip-flop attivo sul fronte di salita del clock

3.4 Visione dell'insieme

Nella figura 3.17 è rappresentato il progetto dell'interfacciamento delle periferiche sotto forma di scatola nera.

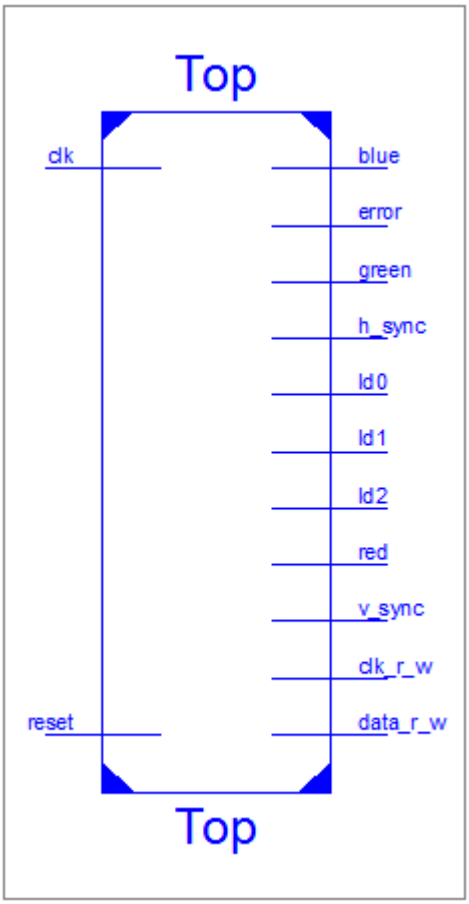


Figura 3.17 Blocco `Top`

Analisi input/output:

- `clk`: segnale in ingresso, rappresenta il clock interno della scheda (100 MHz);
- `reset`: segnale in ingresso, rappresenta il pulsante B8 della scheda(tasto centrale) per avviare la procedura di reset del mouse in caso di errore;
- `blue, green, red`: questi tre segnali in uscita rappresentano l'informazione diretta sulla porta vga;
- `h_sync, v_sync`: segnali in uscita i quali rappresentano i due segnali di sincronismo per lo schermo, rispettivamente orizzontale e verticale;
- `ld0, ld1, ld2`: segnali in uscita "*active high*" collegati rispettivamente al bit 1, 2 e 0 del `bus_mouse` per visualizzare il corretto funzionamento del tasto destro, centrale e sinistro del mouse;

- `error`: segnale in uscita "*active high*" collegato al led LD7, rappresenta un bit di errore dovuto al mouse;
- `clk_r_w, data_r_w`: rappresentano rispettivamente la linea di clock e di dati verso il mouse, linee bidirezionali gestibili tramite buffer tri-state.

Codice di implementazione:

Il codice che realizza questo blocco è fatto perlopiù da inizializzazioni e dichiarazioni dei sottoblocchi di cui è composto. Al suo interno sono contenute le definizioni dei blocchi da collegare, i segnali per collegarli e la dichiarazione di due buffer tri-state per la linea dati e clock verso il mouse.
Le istruzioni per istanziare i due buffer sono:

```
data_r_w <= data_write when data_cmd = '1' else 'Z';
data_read <= data_r_w;
clk_r_w <= clk_write when clk_cmd = '1' else 'Z';
clk_read <= clk_r_w;
```

Questo dispositivo per la linea dati (ugualmente anche per la linea di clock) fa si che quando si vuole scrivere sulla linea (`data_r_w`) bisogna portare a livello logico alto il segnale di controllo (`data_cmd`), mentre quando si vuole solo leggere il dato inviato dal mouse lo si porti a '0'.
Una rappresentazione grafica è riportata in figura 3.18.

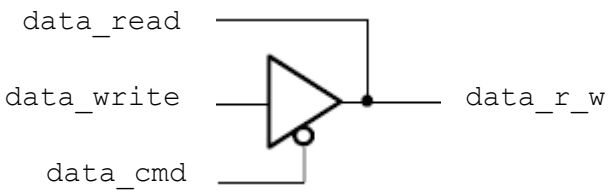


Figura 3.18 Buffer tri-state

Andando leggermente più nel dettaglio, vediamo ora come si presenta all'interno il progetto.
Nella figura 3.19 è riportato lo schema interno del modulo `Top`, è possibile visualizzare le interconnessioni tra i vari moduli realizzati nel progetto e la realizzazione dei buffer tri-state.

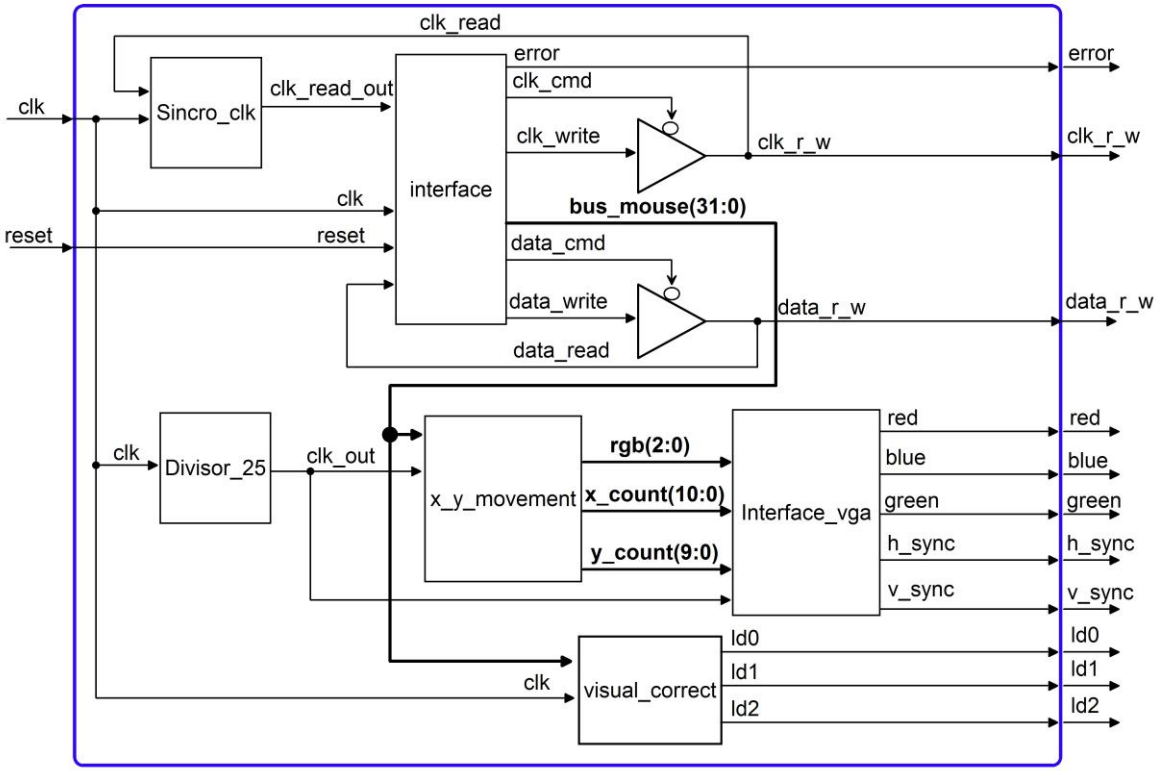


Figura 3.19 Schema interno modulo `Top`

Capitolo 4

Conclusioni

4.1 Simulazioni

Terminata la fase di progettazione e di scrittura del codice, sono passato alle simulazioni.
Le simulazioni (*testbench*) tramite il software *Isim* della Xilinx sono uno strumento molto utile per verificare se il codice scritto compie effettivamente le operazioni volute.
I *testbench* sono dei file che simulano il comportamento dell'oggetto in questione attraverso il cambiamento di certi segnali a cura del programmatore.
In precedenza si è riportato il grafico del relativo blocco di sincronizzazione (figura 3.5) e del blocco di divisore di frequenza (figura 3.15).
Nella figura 4.1 è riportato uno spezzone del *testbench* del blocco `interface` in particolare è riportato l'invio da parte del host verso il mouse del dato di reset (FF_{16}) incapsulato in un pacchetto da 11 bit tutti a 1 tranne il bit di start, in particolare il comando per il buffer tri-state (`data_cmd`) è posto a '1' ovvero in scrittura e il dato tramite il segnale `data_write` viene inviato al mouse (vedi stringa e numeri neri nella figura sottostante). Questo invio del dato era stato scritto in precedenza nel blocco `interface` (stato `sending`) invece nel *testbench* viene riportata la simulazione di risposta del mouse, in questo caso, per questione di spazi è stata riportata la risposta con il byte di *acknowledge*(FA_{16}) senza il seguito dei rimanenti byte (AA_{16} e 00_{16}).
La risposta simulata avviene quando il segnale di comando del buffer tri-state (`data_cmd`) viene portato ad un valore logico basso il quale segnala l'intenzione di voler leggere sulla linea `data_read`, infatti questa transizione è seguita dalla lettura sulla linea dati. Sulla linea dati (`data_read`) tramite il *testbench* è stato spedito il byte FA_{16} sempre nell'apposito pacchetto di 11 bit, lo si può notare dalla stringa e dai numeri rossi nella figura 4.1.
Oltre a questa simulazione si può simulare l'intero progetto attraverso il blocco `Top`, per verificare il corretto funzionamento sia della parte mouse che della parte video.

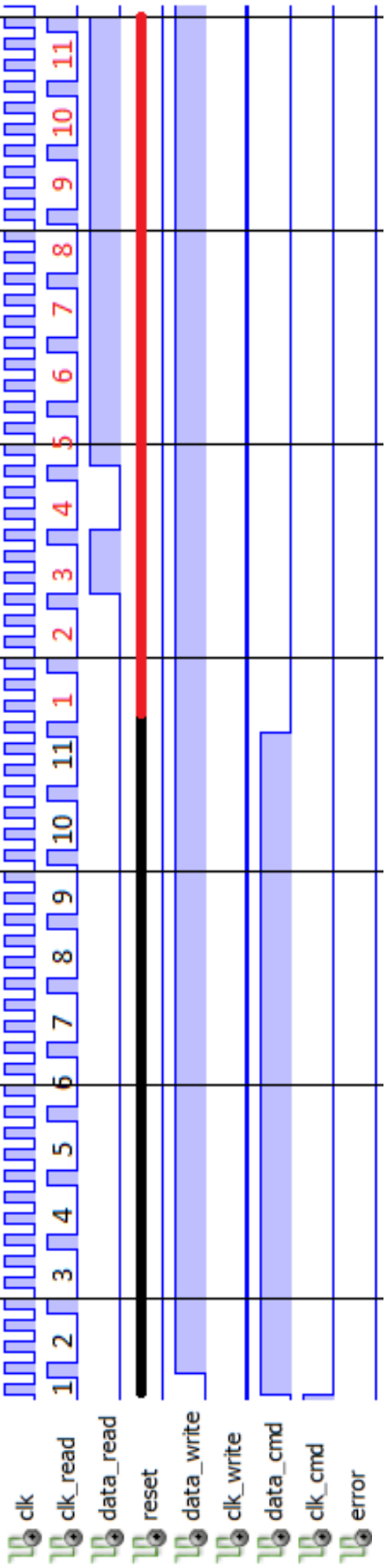


Figura 4.1 Testbench blocco `interface`

La simulazione nella figura 4.1 è di tipo *"Behavioral"–"Comportamentale"* che corrisponde al comportamento che dovrebbero assumere le uscite e gli ingressi della scheda analizzando solamente il codice che costituisce il progetto stesso. Nella simulazione in questione 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 *"Post place & route"–"Piazzamento e cablaggio"* quando 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 più vicina alla realtà del comportamento del sistema realizzato.

4.2 Risultati ottenuti

In fase di sintetizzazione sono presenti 2 *"warning"–"avvertimento"* e sono dovuti al fatto che il `bus_mouse(31:3)` nel blocco *visual* non è usato infatti si è usato solo il bit 0, il bit 1 e il bit 2 ovvero tasto destro, sinistro e centrale rispettivamente, il secondo *"warning"* è dovuto ad un segnale `initial` il quale viene inizializzato ad un valore e poi all'accensione del dispositivo viene portato ad un valore logico alto e da quel valore non viene più modificato. Questo segnale serve per avviare il ciclo di reset e di impostazione del mouse in automatico senza che, ad ogni accensione premere il pulsante B8 ovvero reset per avviare la comunicazione con il mouse.

Per quanto riguarda le risorse occupate, il progetto ne occupa una minima parte. Come si può notare in Tabella 4.2 è stata riportata una parte del riassunto fatto dopo l'implementazione da parte del software *"ISE Design Suite"* in dotazione con la scheda.

Riepilogo Utilizzo del dispositivo			
Utilizzazione Logica	**Usati**	**Disponibili**	**Uso %**
Numero di registri delle Slice	209	18,224	1%
Numero di Slices LUT	364	9,112	3%
Numero di Slices utilizzate	134	2,278	5%
Numero di MUXCY utilizzati	132	4,556	2%
Numero di IOB utilizzati	13	232	5%

Tabella 4.2 Riassunto dell'utilizzo dei dispositivi

Nella figura 4.3 è riportata una foto del relativo progetto finito comprendente interfaccia Mouse Usb e monitor VGA, inoltre nella figura 4.4 è riportata un'immagine dell'accensione dei led LD0, LD1 e LD2 attraverso la pressione del pulsante destro, centrale e sinistro contemporaneamente.

Figura 4.3 Monitor Mouse e NEXYS3

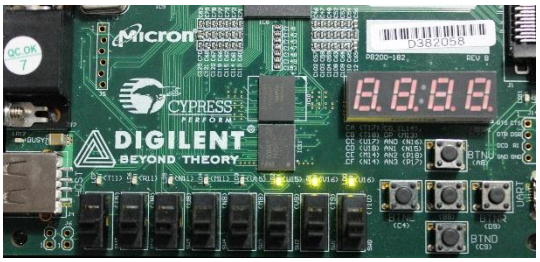


Figura 4.4 Accensione Led pulsanti Mouse

4.3 Conclusioni

Dopo aver scoperto i tanti pregi del VHDL e della programmazione concorrenziale, dopo aver scoperto le potenzialità dell'avere una scheda talmente flessibile che può essere programmata in ogni modo che si voglia, dopo aver imparato ad usare nuovi tool, nuovi ambienti, nuovi assemblatori etc., dopo aver avuto l'occasione di studiare approfonditamente il metodo con cui una periferica con porta USB, soprattutto riguardo al mouse, si interfaccia con un microprocessore e dopo aver approfondito lo studio sulla comunicazione con un monitor attraverso la porta VGA, l'impressione che lascia il lavoro svolto è sicuramente positiva.
Un problema riscontrato (e risolto grazie all'aiuto del docente) nel corso del progetto riguarda principalmente l'interfacciamento e la comunicazione con il mouse.

4.3.1 Sviluppi futuri

Un sviluppo di questo progetto potrebbe essere quello di riuscire ad implementare l'interfaccia mouse ed estenderla nella IMPS 2 totalmente, in questo progetto si è realizzato l'interfaccia per il funzionamento della rotellina e non dei pulsanti ausiliari in quanto non era in mio possesso un intellimouse con il quarto e quinto pulsante.
Un ulteriore sviluppo potrebbe essere quello di interfacciarsi con un pc attraverso la porta seriale (uart) e impostare i vari settaggi del mouse ovvero sample rate, resolution, scaling e far funzionare il mouse in altri modi come wrap e remote. Questo si potrebbe fare con un applicazione di LabView e attraverso il protocollo seriale andare a comunicare con FPGA e mouse.
Nella figura 4.5 è riportata l'interazione con FPGA tramite host PC based Front Panel.

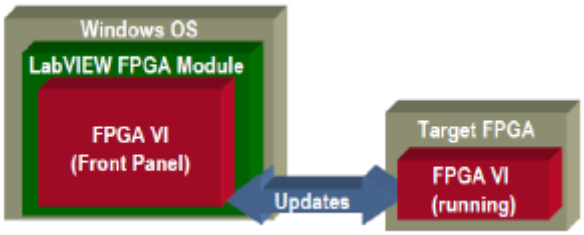


Figura 4.5 Interazione FPGA PC

Appendice

Appendice A,

sincro.vhd

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

entity Sincro_clk is
    Port ( clk_read : in  STD_LOGIC;
           clk : in  STD_LOGIC;
           clk_read_out : out  STD_LOGIC);

end Sincro_clk;

architecture Behavioral of Sincro_clk is
signal i : STD_LOGIC := '0';
signal j : STD_LOGIC := '0';
begin
 process(clk)
 begin
  if(clk'event and clk ='1') then
  i <= clk_read;
  j <= i;
  end if;

end process;
 clk_read_out <= j and (not i) ;

end Behavioral;
```

Appendice B,

interface.vhd

```
.....
if (clk'event and clk = '1') then
 if(reset = '1' or initial = '0') then              -- reset machine
  data_reg1 <= (OTHERS => '0');
  data_reg2 <= (OTHERS => '0');
  data_reg3 <= (OTHERS => '0');
  data_reg4 <= (OTHERS => '0');
  data_out <= (OTHERS => '1');                      -- send 0xFF to device
  count <= (OTHERS => '0');
  bus_mouse<= (OTHERS => '0');
  error <= '0';
  bit_count <= (OTHERS => '0');
  bit_parity <= '1';                                -- last bit is one
  data_write <= '0';
  data_cmd <= '0';
  clk_write <= '0';
  clk_cmd <= '0';
  state <= host_dc;
  sel <= 0;
  cnt <="00";
  int_mouse <='0';
  initial <= '1';
else
 case state is
  when host_dc =>                                   -- host to device communication
       count <= count + 1;
       if (count < 16000) then                      -- attend 160 microseconds
        clk_cmd <= '1';                             -- for write on the line clock
        clk_write <= '0';
        state <= host_dc;
       else
        data_cmd <= '1';                            -- for write on the line data
        data_write <= '0';                          -- first bit(bit start)
        bit_parity <= '1';                          -- last bit is one
        clk_cmd <= '0';
        count <= (OTHERS =>'0');
        sel <= 0;
        state <= sending ;
        end if;

when sending =>                                             -- send byte to device
 if (clk_read = '1' ) then       -- write a bit on the data line when the clock is low
  sel <= sel + 1;
   if (sel < 8) then
       data_write <= data_out(sel);
       bit_parity <= bit_parity xor data_out(sel);
   elsif(sel = 8) then
       data_write <= bit_parity;
   elsif(sel = 9) then
       data_write <= '1';
   else
       data_cmd <= '0';                             -- preparing read data and clock
       clk_cmd <= '0';
       sel <= 0;
       bit_count <= "000001";
       bit_parity <= '1';                           -- last bit is one
       state <= device_hc;
   end if;
end if;
```

```
     when device_hc =>
      if (clk_read = '1') then -- read a bit on the data line when the clock is low
       bit_count <= bit_count + '1';
        if(bit_count < 10) then                    -- control parity first byte;
            bit_parity <= data_read xor bit_parity;
            data_reg1 <= data_read & data_reg1(7 downto 1);

     elsif(bit_count = 10) then                    -- first parity bit
           if(bit_parity /= data_read) then
            error <='1';
           end if;
          bit_parity <= '1';                       -- last bit is one
           if(data_out = "11110011" or data_out = "11001000" or data_out = "11110100"
           or data_out = "01100100" or data_out = "01010000") then  -- F3,C8,64,50,F4
            state <= control_data_init;
            bit_count<="000001";
           end if;
           elsif(bit_count = 11) then              -- first stop bit
            if(data_read /= '1') then
             error <='1';
            end if;

  elsif(bit_count <= 20) then                      -- control parity second byte
      bit_parity <= data_read xor bit_parity;
      data_reg2 <= data_read & data_reg2(7 downto 1);
     elsif(bit_count = 21) then             -- second parity bit
      if(bit_parity /= data_read) then
       error <='1';
      end if;
      bit_parity <= '1';                           -- last bit is one
      if (data_out = "11110010") then              -- sent 0xF2?
      state <= control_data_init;
      bit_count <= "000001";
      end if;
     elsif(bit_count = 22) then                    -- second stop bit
      if(data_read /= '1') then
       error <='1';
      end if;

 elsif(bit_count <= 31) then                       -- control parity third byte ;
     bit_parity <= data_read xor bit_parity;
     data_reg3 <= data_read & data_reg3(7 downto 1);
     elsif(bit_count = 32) then                    -- parity bit
      if(bit_parity /= data_read) then
       error <='1';
      end if;
      bit_parity <= '1';                           -- last bit is one
     elsif(bit_count = 33) then                    -- third stop bit
      if(data_read /= '1') then
       error <='1';
      end if;
      if(data_out = "11111111") then               -- sent FF?
       bit_count <= "000001";
       state <= control_data_init;
      else
       if (int_mouse = '0') then
        bus_mouse <= "00000000" & data_reg3(7 downto 0)
                         & data_reg2(7 downto 0) & data_reg1(7 downto 0);
        state <= idle;
        end if;
       end if;
```

```
   elsif(bit_count <= 42) then                              -- control parity fourth byte;
       bit_parity <= data_read xor bit_parity;
       data_reg4 <= data_read & data_reg4(7 downto 1);
       elsif(bit_count = 43) then       -- parity bit
        if(bit_parity /= data_read) then
         error <='1';
        end if;
        bit_parity <='1';
        bus_mouse <= data_reg4(7 downto 0)& data_reg3(7 downto 0)
                    & data_reg2(7 downto 0)  & data_reg1(7 downto 0);
        elsif(bit_count = 44) then
         if(data_read /= '1') then
          error <='1';
         end if;
        state<=idle;
       end if;
      end if;


when control_data_init =>
       count <= count + 1;
        if (count = 10000) then                      -- attend 100 microseconds
         if (data_out = "11111111") then             -- sent 0xFF
              state <= host_dc;
              data_out<="11110011";                  -- 0xF3
              if (data_reg1 = "11111010" and data_reg2 = "10101010"
                     and data_reg3 = "00000000") then -- receive 0xFA,0xAA,0x00
               error <= '0';
              else
               error <= '1';
              end if;

        elsif (data_out = "11110011") then           -- sent 0xF3?
              state <= host_dc;
              cnt <= cnt + '1';
              if (cnt = "00") then
              data_out <= "11001000";                -- 0xC8
              elsif (cnt = "01") then
                data_out <= "01100100";              -- 0x64
              elsif (cnt = "10") then
                data_out <= "01010000";              -- 0x50
                cnt <= "00";
              end if;
              if (data_reg1 = "11111010") then       -- receive 0xFA?
                error <='0';
              else
                error <='1';
              end if;

        elsif ((data_out = "11001000") or (data_out = "01100100")
                     or(data_out = "01010000")) then  -- sent 0xC8,0x64,0x50?
               state <= host_dc;
               data_out <= "11110011";               -- 0xF3
               if (data_out = "01010000") then       -- sent 0x50?
                data_out <= "11110010";              -- 0xF2
               end if;
               if (data_reg1 = "11111010") then      -- receive 0xFA?
                error <='0';
               else
                error <='1';
               end if;

         elsif (data_out = "11110010") then          -- sent 0xF2?
```

```
              state <= host_dc;
              data_out <= "11110100";                -- 0xF4
              if (data_reg1 = "11111010") then       -- receive 0xFA?
               error <='0';
              else
               error <='1';
              end if;
              if (data_reg2 = "00000000") then       -- receive 0x00?
               int_mouse <= '0';                     -- mouse standard
              else
               int_mouse <='1';                      -- intellimouse
              end if;

       elsif(data_out = "11110100") then             -- sent 0xF4?
             if(data_reg1 = "11111010") then         -- receive 0xFA?
              error <= '0';
             else
              error<='1';
             end if;
             data_out<=(others=>'0');
             state <= idle;
             -- other condition example 0xF5...
       end if;
       count <= (others => '0');
    end if;

when idle =>

       bit_parity <= '1';
       clk_cmd <= '0';
       data_cmd <= '0';
       bit_count <= "000001";
       if (data_read = '0') then
        state <= device_hc;
       end if;

 when others =>
 state <= idle;

end case;

.....
```

Appendice C

x_y_movement.vhd

```
.....
       if (clk25'event and clk25 = '1')then
        count <= count +'1';
        if (count = 10) then
         count <=(others=>'0');
         if (bus_mouse /= mouse ) then
         mouse(31 downto 0) <= bus_mouse(31 downto 0);
       if (bus_mouse(4) = '0') then                          -- positive movement to right
                    if (x_position + bus_mouse(15 downto 8) > 640 ) then
                          x_position <= "01010000000";
                    else
                          x_position <= x_position + bus_mouse(15 downto 8);
                    end if;
       else                                                  -- negative movement to left
                    if (not(bus_mouse(15 downto 8)) + '1' > x_position ) then
                          x_position <= (others => '0');
                    else
                          x_position <= x_position - (not(bus_mouse(15 downto 8)) + '1');
                    end if;
             end if;
        if (bus_mouse(5) = '0') then                         -- movement positive upward
             if (bus_mouse(23 downto 16) > y_position) then
                    y_position <= (others => '0');
             else
                    y_position <= y_position - bus_mouse(23 downto 16);
             end if;
        else                                                 -- negative movement down
             if(not(bus_mouse(23 downto 16)) + '1' + y_position > 480) then
                    y_position <= "0111100000";
             else
                    y_position <= y_position + (not(bus_mouse(23 downto 16)) + '1');
             end if;
       end if;
             if(bus_mouse(27) = '0') then
                    z_position <= z_position + bus_mouse(26 downto 24);
             else
                    z_position <= z_position - (not(bus_mouse(26 downto 24)) + '1');
             end if;
        end if;
       end if;
       end if;
 x_count <= x_position;
 y_count <= y_position;
 rgb <= z_position;
.....
```

Appendice D

interface_vga.vhd

```
.....
  if (clk_25'event and clk_25 ='1') then
   if(h_count=799)then
        h_count<=(others =>'0');
        if (en = '1') then
         en <= '0';
        end if;
       else
        h_count <= h_count + '1';
       end if;
       if (v_count >= 0 and v_count <= 479) then            -- screen color
        red_signal <= rgb(0);
        blue_signal <=rgb(1);
        green_signal <=rgb(2);
       end if;
       if( v_count - y_count >= -10 and v_count - y_count <= 10 and -- pointer
             h_count - x_count >= -10 and h_count - x_count <= 10) then
        if((v_count - y_count >= 0 and v_count - y_count <= 0) or
                    (h_count - x_count >= 0 and h_count - x_count <= 0))then
                    red_signal <= '1';
                    green_signal <='0';
                    blue_signal<='0';
        end if;
       end if;
       if(h_count >= 659 and h_count <= 755) then                -- sincro horizontal
        h_sync <= '0';
       else
        h_sync <= '1';
       end if;

       --reset
       if(v_count >= 524 and h_count <= 699)then
        v_count <= (others => '0');
       elsif(h_count = 699) then
        v_count <= v_count +1;
       end if;
       if(v_count >= 493 and v_count <= 494) then                -- sincro vertical
        v_sync <= '0';
       else
        v_sync <= '1';
       end if;

       if(h_count <=639) then
        horizontal_en <='1';
       else
        horizontal_en <='0';
       end if;

       if(v_count <= 479) then
        vertical_en <= '1';
       else
        vertical_en <= '0';
       end if;
       red <= red_signal and video_en;
       blue <= blue_signal and video_en;
       green <= green_signal and video_en;

  end if;
.....
```

Ringraziamenti

E’ mio desiderio ringraziare tutte le persone che mi hanno permesso di arrivare fin qui, in particolare la mia famiglia Luigino, Sabrina e Andrea che mi hanno sostenuto moralmente, i miei nonni, la fidanzata Jessica per essermi stata accanto in ogni momento, Serena, Daniela, Graziano e i mie colleghi di università Andrea e Stefan.
Inoltre volevo ringraziare il mio relatore Daniele Vogrig per i consigli ricevuti per realizzare questa tesi e per la grande professionalità che ha dimostrato.

Bibliografia

- Slides corso “Elettronica dei sistemi digitali” a cura del professore D.Vogrig;
- Testo “Progettazione Automatica di Circuiti e Sistemi Elettronici”;
- Testo “FPGA Prototyping by VHDL Examples” ;
- Sito web http://www.computer-engineering.org/;
- Sito web della Digilent www.digilentinc.com;
- Sito web della Xilinx www.xilinx.com;
- Sito web www.parthmehta.in;
- Sito web www.delucagiovanni.com.