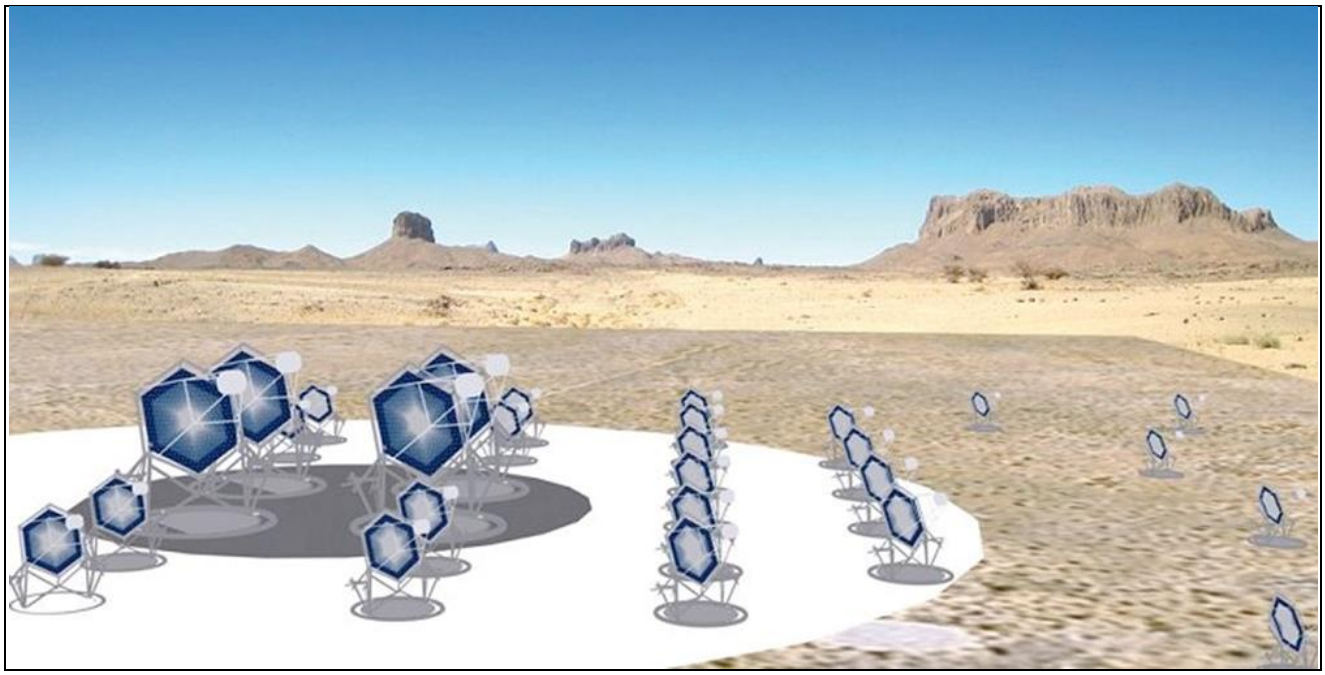
OSSERVATORIO ASTROFISICO DI CATANIA

Aggiornamenti Firmware dell'FPGA per la Front-End Electronics della Camera del Telescopio ASTRI-SST-2M - V1



Osservatorio Astrofisico di Catania

D. MARANO[(1)], S. GAROZZO[(1)], G. BONANNO[(1)], A. GRILLO[(1)], G. ROMEO[(1)], M. C. TIMPANARO[(1)], M. RAPISARDA[(1)]

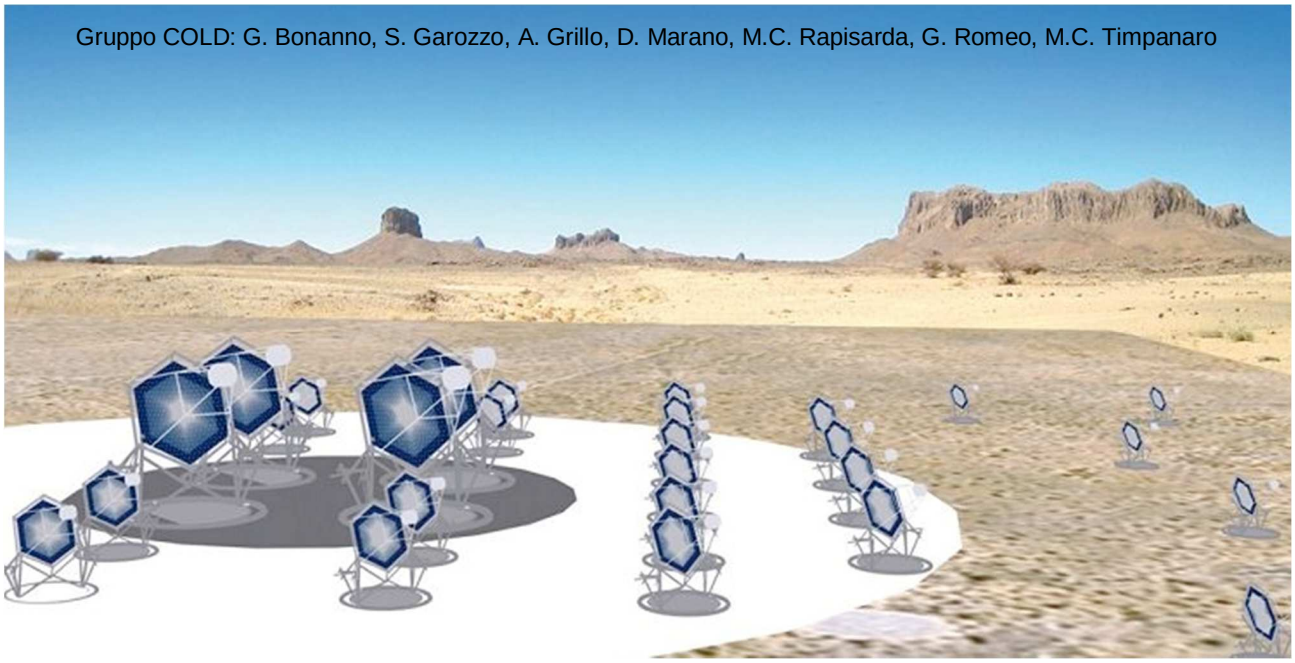
(1) INAF – Osservatorio Astrofisico di Catania

Rapporti interni e tecnici
N. 6/2016

INAF - Osservatorio Astrofisico di Catania
Via Santa Sofia, 78 I-95123 Catania, Italy Tel.: +39- 095-7332 111 Fax: +39-095-330592
Sede "Mario G.Fracastoro" (Etna) – Tel +39-095-911580 Fax+39-095-916184
www.oact.inaf.it - oacatania@oact.inaf.it

Aggiornamenti Firmware dell'FPGA per la Front-End Electronics della Camera del Telescopio ASTRI-SST-2M

Gruppo COLD: G. Bonanno, S. Garozzo, A. Grillo, D. Marano, M.C. Rapisarda, G. Romeo, M.C. Timpanaro



Prepared by:	Name: Davide Marano Salvatore Garozzo	Signature:		Date:	25/09/2016
Reviewed by:	Name: Giovanni Bonanno	Signature:		Date:	25/09/2016
Approved by:	Name: Giovanni Bonanno	Signature:		Date:	25/09/2016

TABLE OF CONTENTS

DISTRIBUTION LIST

ASTRI mailing list	astri@brera.inaf.it
AIV-CAM mailing list	aivcam@brera.inaf.it
Giovanni Pareschi	giovanni.pareschi@brera.inaf.it
Patrizia Caraveo	pat@lambrate.inaf.it
Salvo Scuderi	scuderi@oact.inaf.it
Mauro Fiorini	fiorini@lambrate.inaf.it
Rachele Millul	rachele.millul@brera.inaf.it
Rodolfo Canestrari	rodolfo.canestrari@brera.inaf.it
Osvaldo Catalano	osvaldo.catalano@iasf-palermo.inaf.it
Stefano Vercellone	stefano@ifc.inaf.it
Maria Concetta Maccarrone	mcm@ifc.inaf.it
Giovanni La Rosa	larosa@ifc.inaf.it
Francesco Russo	russo@ifc.inaf.it
Giovanni Bonanno	gbo@oact.inaf.it
Davide Marano	davide.marano@oact.inaf.it
Salvatore Garozzo	salvatore.garozzo@oact.inaf.it
Alessandro Grillo	agrillo@oact.inaf.it
Giuseppe Romeo	giuseppe.romeo@oact.inaf.it
Domenico Impiombato	domenico.impiombato@ifc.inaf.it
Giuseppe Sottile	sottile@ifc.inaf.it
Salvatore Giarrusso	jerry@ifc.inaf.it
Carmelo Gargano	gargano@ifc.inaf.it
Pierluca Sangiorgi	sangiorgi@ifc.inaf.it

DOCUMENT HISTORY

Version	Date	Modification
1.0	Date	first version
		update

LIST OF ACRONYMS

OACT	Osservatorio Astrofisico di Catania
IFC	Istituto di Astrofisica Spaziale e Fisica Cosmica di Palermo
COLD	Catania astrophysical Observatory Laboratory for Detectors
SiPM	Silicon Photo-Multiplier
MPPC	Multi Pixel Photon Counter
SST-2M	Small-Size Telescope Dual-Mirror
PDM	Photon Detection Module
ASIC	Application Specific Integrated Circuit
FEE	Front-End Electronics
BEE	Back-End Electronics
FPGA	Field Programmable Gate Array
CITIROC	Cherenkov Imaging Telescope Integrated Read-Out Chip
ILA	Integrated Logic Analyzer
VHDL	Very-high speed integrated circuits Hardware Description Language
HW	Hardware
SW	Software
SDK	Software Development Kit
GPIO	General Purpose Input/Output

REFERENCE DOCUMENTS

[R1] Artix-7 Series Datasheet, http://www.xilinx.com/support/documentation

[R2] ASTRI-TEC-IASFPA-3200-XXX: “PDM Manager: Architecture of the Firmware of the FPGA-FE”, 2015.

[R3] ASTRI-TN-IASFPA-3200-024: “Detection Technique and Performance of the ASTRI SST-2M Camera”, 2014.

[R4] ASTRI-SPEC-IASFPA-3200-009: “FPGA Board Architectural and Design Specification”, 2013.

[R5] CITIROC datasheet, WEEROC, http://www.weeroc.com/asic-products/citiroc-1.

1. INTRODUZIONE

Il presente documento descrive in dettaglio i vari aggiornamenti associati allo sviluppo dell'architettura del firmware FPGA per la FEE della camera ASTRI, allo scopo di tener traccia delle varie modifiche effettuate dalla prima versione rilasciata in INAF dalla ditta Mindway s.r.l. nel dicembre 2015 () fino all'implementazione definitiva dell'ultima *release* a fine settembre 2016 ().

Dalla versione originaria, si sono rese necessarie modifiche sia alla parte hardware che al comparto software, allo scopo di risolvere problematiche riscontrate dai test, ottimizzare le funzionalità di alcuni blocchi ed inserire nuovi moduli strutturali in grado di eseguire le operazioni richieste.

Per ognuna delle modifiche descritte è stata realizzata una implementazione hardware su sistema di sviluppo Xilinx VIVADO, un software in linguaggio C su sistema di sviluppo Xilinx SDK, un file di probe relativo all'ILA Chipscope (con estensione .ltx), e due file di programmazione binari, uno per la memoria RAM dell'FPGA (con estensione .bit) e l'altro per la memoria FLASH integrata nella scheda FPGA della PDM (con estensione .mcs). Questi ultimi tre file, unitamente ad una sintetica descrizione testuale, sono stati inseriti, per ognuna delle versioni del firmware, in un *repository* condiviso su un server di rete (https://www.dropbox.com/home/INAF/CATANIA).

L'organizzazione di questo rapporto tecnico prevede la struttura seguente. Nella sezione 2 viene brevemente illustrata l'architettura del firmware, ed il funzionamento di base dei principali moduli HW in esso implementati. La sezione 3 descrive gli strumenti di sviluppo, le piattaforme HW e SW, nonché l'apparato sperimentale adottato per le procedure di test. Nella sezione 3 vengono riportati nel dettaglio i vari aggiornamenti del firmware, che coinvolgono sia la parte HW che la parte SW. Infine, la sezione 5 riassume l'elenco dei comandi relativi alla PDM e le tabelle di configurazione aggiornate in seguito alle modifiche effettuate.

2. ARCHITETTURA DEL FIRMWARE

L'architettura del firmware dell'FPGA di *front-end* comprende essenzialmente:

- un microprocessore MicroBlaze (uBlaze) integrato;
- i moduli HW funzionali alle specifiche tecniche;
- le tabelle di parametri/segnali che gestiscono il funzionamento dei singoli moduli;
- i collegamenti dai pin dell'FPGA verso il mondo esterno (ASIC, convertitori ADC, multiplexer degli HK, interfacce UART ed USB, BEE).

L'FPGA utilizzata per il progetto della FEE è una Xilinx Artix-7 100T, avente package di tipo FGG-484 (23×23mm^2), speed -2, e un numero di I/O pins pari a 285 [R1].

Il diagramma a blocchi dell'architettura del firmware è mostrato in Fig. 1, dove in rosso sono rappresentati i segnali di clock, in marrone i segnali di interfacciamento con i registri e le GPIO del processore, e in verde i pin relativi ai banchi dell'FPGA.

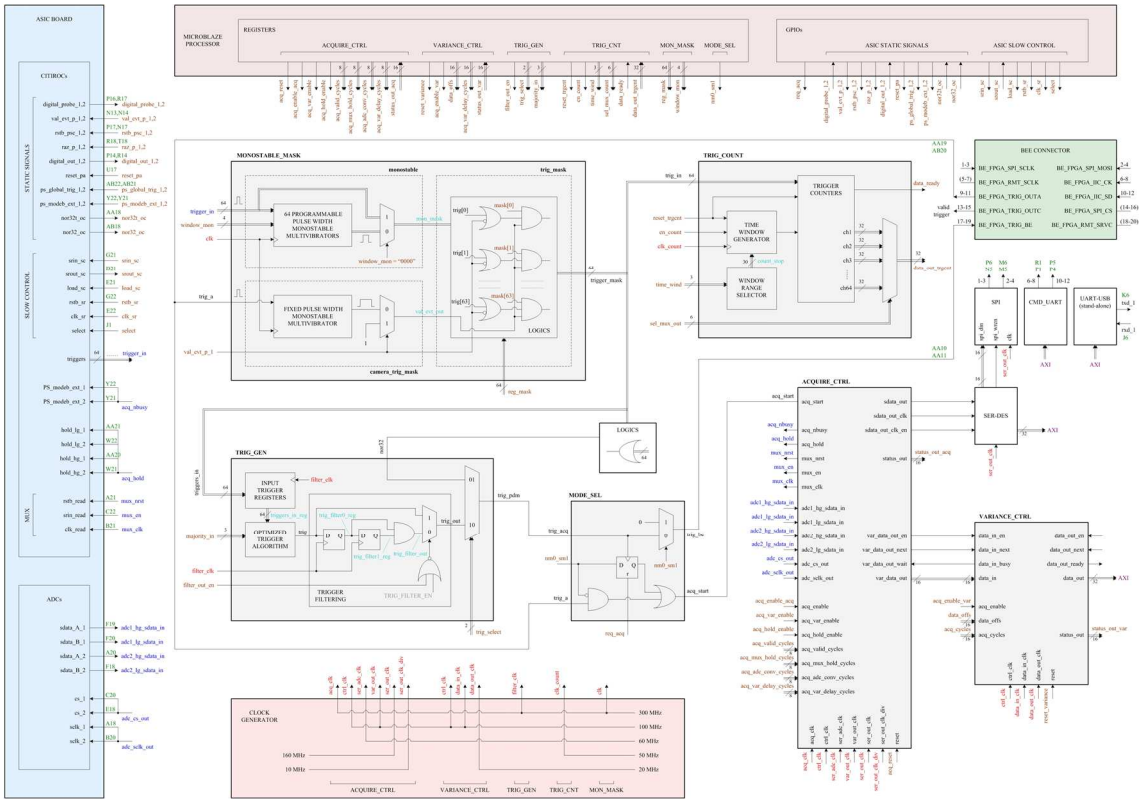


Fig. 1. Diagramma a blocchi dell'architettura del firmware dell'FPGA di front-end.

I principali moduli funzionali che stanno alla base del firmware del progetto sono:

- *acquire_ctrl* (modulo per il controllo delle acquisizioni);
- *variance_ctrl* (modulo per il controllo dei dati di varianza);

- *trig_gen* (modulo per la generazione del trigger di PDM);
- *trig_count* (modulo per il conteggio dei trigger);
- *monostable_mask* (modulo per la gestione dei trigger);
- *mode_sel* (modulo per la selezione del modo operativo).

Completano il quadro dell'architettura una serie di registri e di porte GPIO, unitamente ad alcuni blocchi di generazione dei segnali di clock. Infine, è presente un modulo HW dedicato, l'*xadc*, adibito alla conversione anologico/digitale dei 10 sensori di temperatura presenti nella scheda SiPM, nonchè alla conversione della corrente e della temperatura relative alla scheda FPGA.

Per la gestione dei dati in uscita, indipendentemente dalla configurazione usata, si fa uso di una zona di memoria condivisa come "data buffer". Per ogni tipologia di dati viene assegnata un'area di memoria. Ogni singolo modulo firmware scrive i dati prodotti nei "data buffer" a loro assegnati. I dati scientifici e di varianza vengono memorizzati in dei buffer di memoria dedicati, mentre i dati di conteggio dei trigger e di *housekeeping* vengono conservati in dei registri. Tali dati vengono quindi letti tramite bus AXI dal processore che gestisce la trasmissione verso la BEE.

Nella modalità di funzionamento *standalone* (ovvero con gestione interna dei trigger), i dati vengono prelevati dalle memorie e trasmesse tramite interfaccia UART. Diversamente, nella modalità di funzionamento definitiva (con collegamento diretto alla BEE), i dati degli eventi scientifici non vengono prelevati dalla memoria, bensì spediti direttamente alla BEE tramite interfaccia SPI, mentre i dati degli eventi di varianza, di conteggio dei trigger e di *housekeeping* vengono letti dalle rispettive memorie e trasferiti verso la BEE tramite interfaccia UART.

In particolare, nella modalità operativa *standalone*, per consentire il *debug* del firmware tramite software di controllo, è stata prevista una modalità di trasferimento di tutti i dati tramite porta USB e collegamento al PC.

Nei paragrafi successivi di questa sezione vengono sinteticamente descritti i principali moduli funzionali dell'architettura del firmware.

2.1 Processore MicroBlaze

Il cuore dell'architettura del firmware dell'FPGA è costituito da un microprocessore integrato, il MicroBlaze (uBLAZE), caratterizzato da una frequenza di clock nominale di 100MHz, da due blocchi di memoria RAM per dati e istruzioni, rispettivamente di 1Mbit e 2Mbit, e da un bus di collegamento di classe *Advanced eXtensible Interface* (AXI) per la comunicazione con le periferiche.

Il processore MicroBlaze all'interno del firmware si occupa prevalentemente di:

- gestire il protocollo di comunicazione verso la BEE e verso la porta USB;
- interpretare i comandi provenienti dalla BEE o dalla porta USB;
- gestire lo *slow control* degli ASIC, ovvero trasferire la tabella di configurazione degli ASIC dalla memoria locale ai due CITIROC della scheda di *front-end*;
- gestire lo *slow control* dell'FPGA, ovvero trasferre i parametri ricevuti da comando verso i moduli firmware corrispondenti;
- raccogliere i dati di *housekeeping*
- raccogliere i dati prodotti e spedirli verso la BEE (ad eccezione dei dati degli eventi scientifici provenienti dal modulo *acquire*) oppure verso la porta USB (in questo caso anche gli stessi dati degli eventi scientifici).

I principali segnali che coinvolgono i registri e le GPIO del processore sono illustrati in Fig. 2. Sostanzialmente, i registri sono adibiti alla gestione dei parametri di configurazione e dei dati di *output* dei moduli strutturali, mentre le GPIO consentono un interfacciamento agevole con le linee di uscita verso gli ASIC.

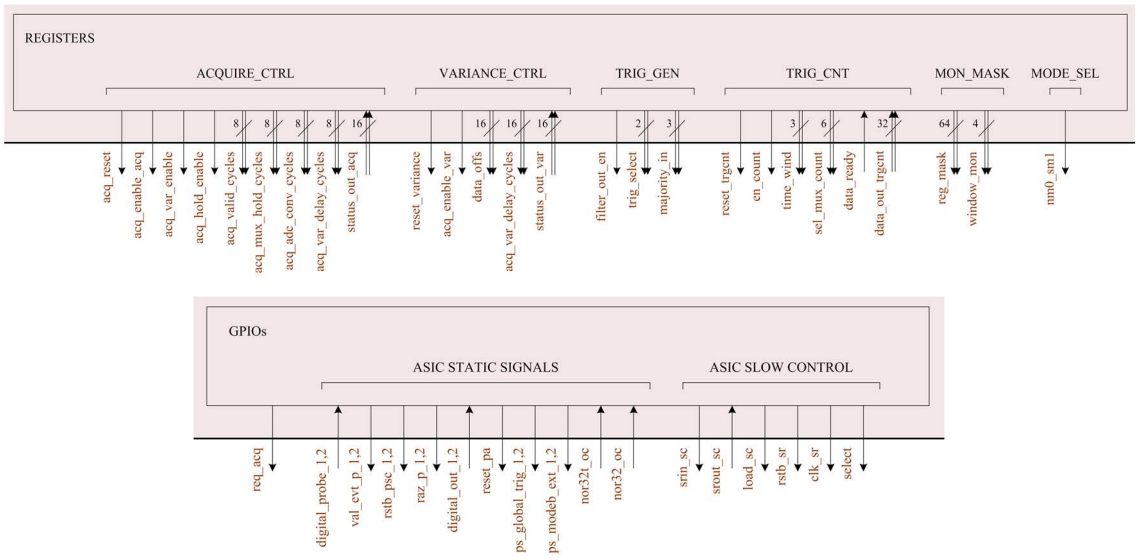


Fig. 2. Schematizzazione dei blocchi di gestione dei registri e delle GPIO del processore.

2.2 Modulo di acquisizione degli eventi

Il modulo *acquire* è il blocco HW più complesso tra tutti i moduli presenti all'interno del diagramma a blocchi dell'FPGA, in quanto gestisce l'acquisizione dei dati provenienti dagli ASIC (siano essi dati di varianza o di eventi scientifici).

La schematizzazione dell'entità strutturale del modulo è rappresentata in Fig. 3.

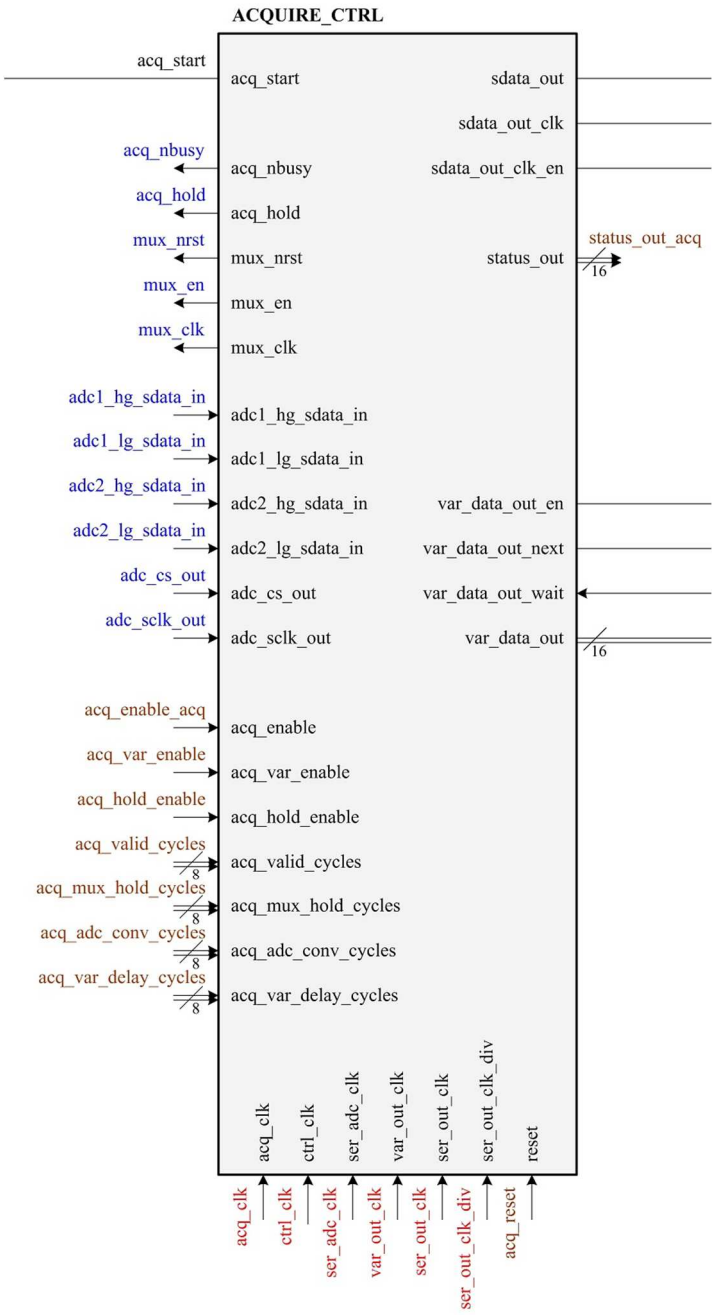


Fig. 3. Schematizzazione dell'entità relativa al modulo acquire.

Il modulo *acquire* si interfaccia direttamente con i CITIROC e gli ADCs della scheda di *front-end*, e, all'interno dell'FPGA, con il modulo della varianza, con il microprocessore uBLAZE (da cui riceve parametri e linee di controllo) e, infine, con i buffer di uscita assegnati, dove scaricherà i dati dei segnali scientifici o di calibrazione.

I parametri di configurazione passati al modulo *acquire* dal microprocessore sono elencati di seguito:

- **acq_enable**: abilita/disabilita l'acquisizione (sia eventi scientifici che di varianza).
- **acq_var_enable**: abilita/disabilita l'acquisizione degli eventi di varianza.
- **acq_hold_enable**: abilita/disabilita la modalità di funzionamento *single shot*.
- **acq_valid_cycles**: misurato in cicli del clock *acq_clk* a 300MHz, rappresenta il ritardo necessario al raggiungimento del picco dei segnali analogici delle due catene di *slow-shaper* degli ASIC.
- **acq_mux_hold_cycles**: misurato in cicli del clock *ctrl_clk* a 100MHz, rappresenta il tempo necessario a far assestare il dato da convertire, in seguito alla selezione del canale da parte dei *mux* degli ASIC.
- **acq_adc_conv_cycles**: misurato in cicli del clock *ctrl_clk* a 100MHz, rappresenta il tempo necessario ad effettuare la conversione completa di un canale.
- **acq_var_delay_cycles**: misurato in cicli del clock *ctrl_clk* a 100MHz, rappresenta il ritardo per il riavvio automatico dell'acquisizione di varianza.

I segnali prodotti in uscita dal modulo *acquire* riguardano il trasferimento dei dati di varianza al modulo *variance* (dati paralleli a 16 bit), ed il trasferimento in seriale dei dati scientifici verso la BEE.

Il codice sorgente del modulo *acquire* è stato realizzato da Francesco Russo (IASF Palermo) su piattaforma ALTERA, e il relativo *porting* su sistema di sviluppo Xilinx è stato effettuato presso OACT, dove si è inoltre sviluppato un opportuno *testbench* per la verifica funzionale del modulo tramite simulatore *iSim*.

Le modifiche firmware che hanno interessato il modulo *acquire* sono descritte nelle sezioni 4.2, 4.4, 4.9, 4.12, 4.13, 4.15.

2.3 Modulo di campionamento della varianza

Il modulo *variance* è adibito al calcolo e all'accumulazione degli eventi di varianza dei 64 canali degli ASIC, relativamente alla sola catena elettronica ad alto guadagno.

La schematizzazione dell'entità strutturale del modulo è rappresentata in Fig. 4.

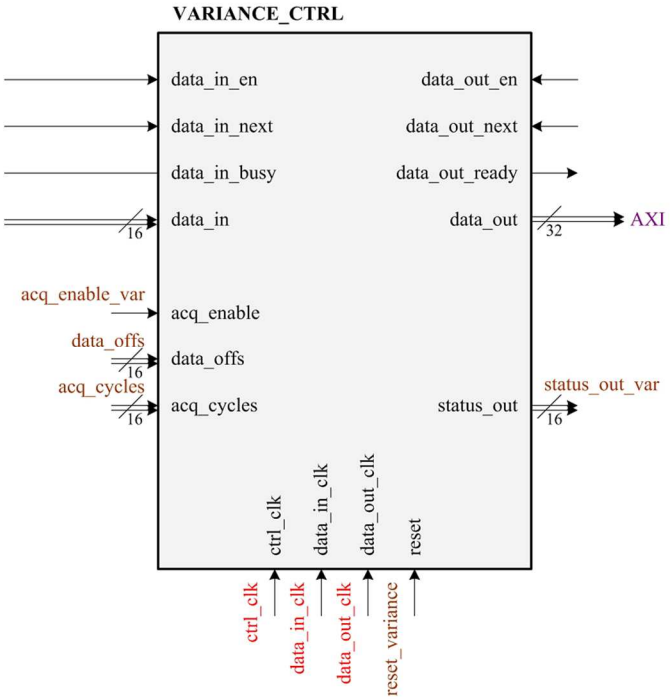


Fig. 4. Schematizzazione dell'entità relativa al modulo variance.

Questo blocco riceve dal modulo *acquire* i 64 valori (uno per ogni pixel) di ogni singolo campionamento. Successivamente, per ogni pixel, l'algoritmo implementato nel modulo sottrae al dato ricevuto un unico valore fisso per tutti i pixel (passato come parametro), corrispondente alla *baseline* media, quindi esegue la somma fra il "valore ottenuto" e il contenuto del registro delle somme, conservando il risultato nel registro delle somme (uno per ciascun pixel). Inoltre, esegue la somma fra il quadrato del "valore ottenuto" e il registro dei quadrati (uno per ogni pixel), conservando il risultato nel registro dei quadrati. Dopo aver sommato un determinato numero di campioni (dettato al modulo da un parametro ricevuto tramite comando), il blocco scarica infine i 2*64 registri (somma e somma dei quadrati) nei buffer di uscita assegnati.

I dati in uscita al modulo di varianza, siano essi somme che somme di quadrati, vengono trasferiti in uscita verso il bus di sistema a 32 bit, e seguono un ordinamento crescente per pixel; più specificamente, i primi 64 valori vengono prelevati dai registri delle somme, e i successivi 64 valori dai registri delle somme dei quadrati.

I parametri di configurazione passati al modulo *variance* dal MicroBlaze sono elencati di seguito:

- **acq_enable_var**: abilita/disabilita la raccolta dei dati di varianza.
- **acq_cycles**: fornisce il numero di campioni *N* di varianza da accumulare.
- **data_offs**: fornisce il valore di *offset* per i dati di varianza (stesso valore a 16 bit per tutti i 64 pixel della PDM).

I dati di uscita del modulo, opportunamente processati, consentono la determinazione del livello di *Night Sky Background* del cielo durante le osservazioni notturne.

Il codice sorgente del modulo *variance* è stato realizzato da Francesco Russo (IASF Palermo) su piattaforma ALTERA, ed il relativo *porting* su sistema di sviluppo Xilinx e è stato effettuato presso OACT, dove si è inoltre sviluppato un opportuno *testbench* per la verifica funzionale del modulo tramite simulatore *iSim*.

Le modifiche al firmware che interessano il modulo *variance* sono descritte nelle sezioni 4.8, 4.15.

2.4 Modulo di generazione del trigger

Il modulo *trig_gen* è dedicato alla formazione del trigger di PDM a partire dai 64 trigger provenienti dagli ASIC, e include un apposito algoritmo di scansione dei canali atto alla rilevazione delle adiacenze logiche tra più pixel.

La schematizzazione dell'entità strutturale del modulo è rappresentata in Fig. 5.

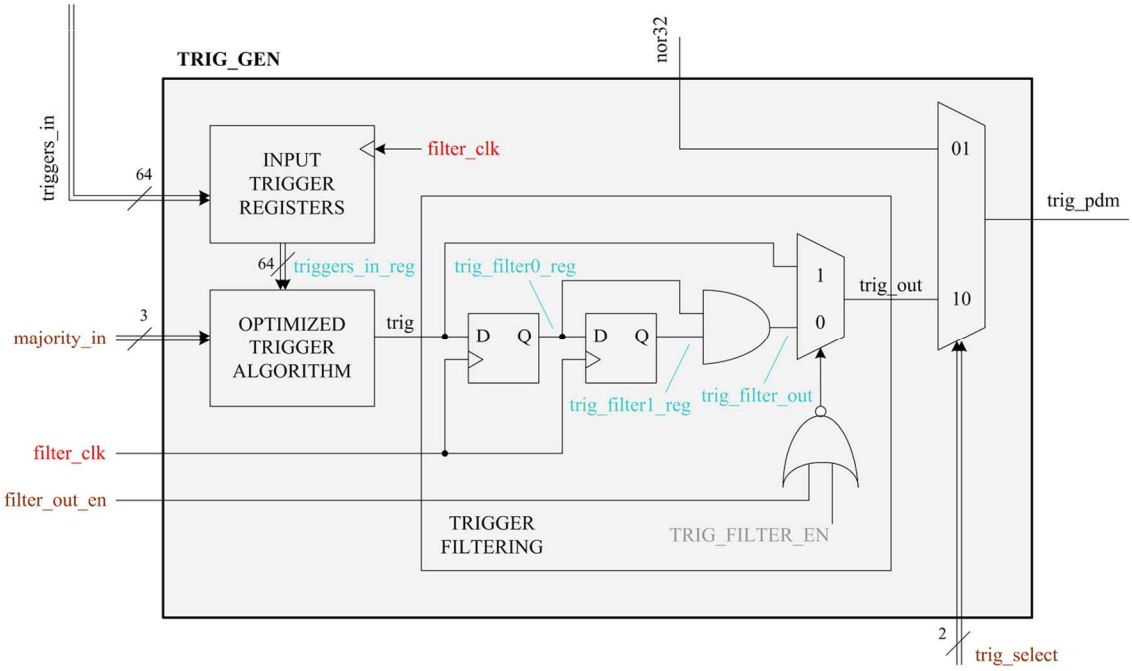


Fig. 5. Diagramma a blocchi del modulo trig_gen.

Il modulo è costituito da una sezione di registrazione di trigger in ingresso, da un blocco dedicato alla realizzazione dell'algoritmo, da una sezione di filtraggio di eventuali segnali spuri dovuti all'asincronismo nell'arrivo dei trigger, e da un multiplexer in uscita per la selezione del segnale prodotto dall'algoritmo oppure della OR logica tra i 64 trigger provenienti dai discriminatori degli ASIC.

Il modulo di generazione del trigger riceve in ingresso i 64 segnali provenienti dai discriminatori dei due CITIROC e, attraverso un algoritmo interno a scansione settoriale, produce in uscita un segnale logico (il cosiddetto trigger di PDM) da inviare alla BEE.

I parametri di configurazione passati al modulo *trig_gen* dal MicroBlaze sono elencati di seguito:

- **trig_select**: seleziona in uscita il segnale prodotto dall'algoritmo di trigger o il segnale costituito dalla OR logica dei 64 trigger dei discriminatori degli ASIC; in alternativa, è altresì possibile porre in alta impedenza l'uscita del modulo.
- **filter_out_en**: abilita/disabilita il filtro in uscita al modulo per la riduzione dei *glitch* dovuti all'arrivo asincrono dei segnali di trigger.

- **majority_in**: numero di pixel adiacenti accesi (utilizzato all'interno dell'algoritmo per la generazione del PDM trigger in base alle adiacenze richieste).

In condizioni nominali, quando la PDM è collegata alla BEE, il segnale di uscita al modulo viene inviato direttamente al connettore FEE-BEE presente nella scheda FPGA.

Il codice sorgente del modulo *trig_gen* è stato realizzato da Francesco Russo (IASF Palermo) su piattaforma ALTERA, ed è stato rifinito, ottimizzato e trasferito su ambiente Xilinx presso OACT, dove si è inoltre provveduto allo sviluppo di un opportuno *test-bench* per la verifica funzionale del modulo tramite simulatore *iSim*.

Le modifiche al firmware che interessano il modulo *trig_gen* sono descritte nella sezione 4.5.

2.5 Modulo di conteggio dei trigger

Il modulo *trig_cnt* è dedicato al conteggio degli impulsi di trigger prodotti in uscita ai 64 discriminatori degli ASIC. Esso è formato da 64 contatori a 32 bit abilitati in una finestra temporale ben definita (variabile in un *range* compreso tra 125ms e 16s).

La schematizzazione dell'entità strutturale del modulo è rappresentata in Fig. 6.

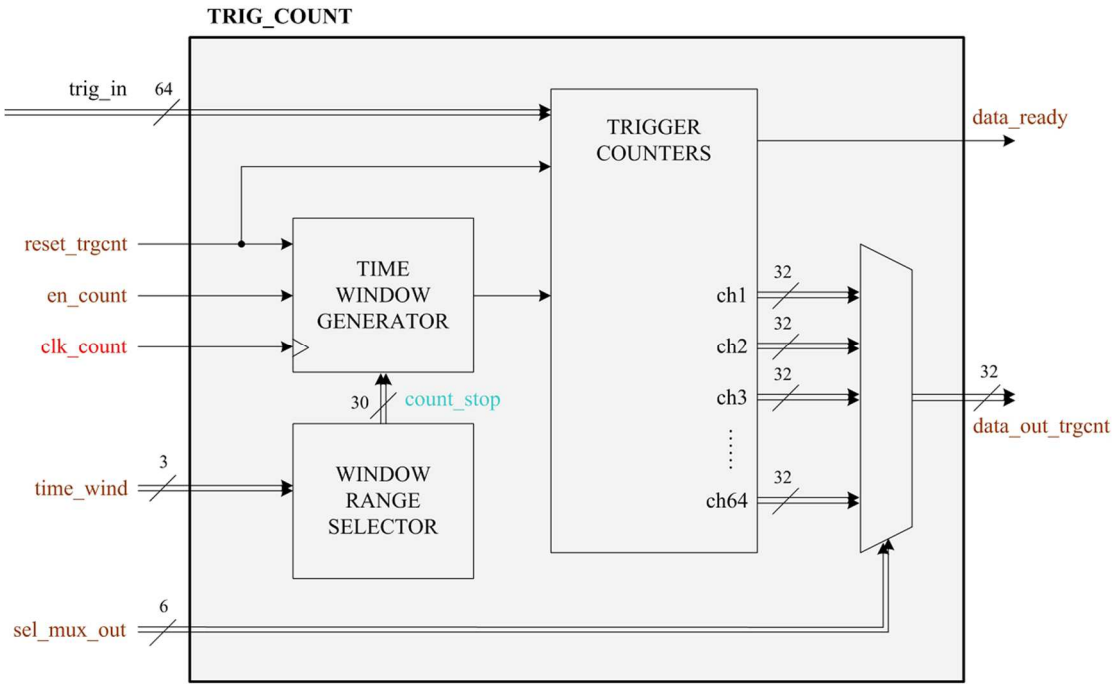


Fig. 6. Diagramma a blocchi del modulo trig_count.

Esso riceve in ingresso i 64 segnali provenienti dai discriminatori degli ASIC e, per ogni canale (pixel), conta i trigger ricevuti in una certa finestra temporale, passata come parametro. Alla fine della finestra temporale impostata, i 64 contatori vengono automaticamente resettati.

I parametri di configurazione del modulo *trig_count* gestiti dal microprocessore sono elencati di seguito:

- **en_count**: segnale che fornisce l'inizio della finestra temporale e che abilita l'avvio dell'operazione di conteggio dei trigger. La larghezza di tale impulso dev'essere superiore ad un periodo del clock "clk_count" per garantire il corretto funzionamento del modulo.
- **reset_trgcnt**: segnale di reset sincrono/asincrono per i 64 contatori di trigger; questo segnale azzera i contatori sia dopo una sessione di conteggio (comportamento sincrono) sia durante una operazione di conteggio (comportamento asincrono). La larghezza di tale impulso dev'essere superiore ad un periodo del clock "clk_count" per garantire il corretto funzionamento del modulo.

- **time_wind**: 3 bit di selezione per la finestra temporale di conteggio. I possibili valori assunti dal parametro seguono la relazione $0.125s \times 2^n$, dove *n* è il valore decimale del parametro; di conseguenza, la finestra temporale selezionabile può variare da un minimo di 0.125s fino ad un massimo di 16s.
- **data_ready**: segnale di uscita che indica la fine dei conteggi dei contatori e la relativa disponibilità dei dati di conteggio.
- **data_out_trgcnt**: segnale di conteggio dei dati relativo al canale selezionato.
- **sel_mux_out**: 6 bit di selezione per i dati di conteggio dei 64 canali degli ASIC.

I dati di uscita del modulo, opportunamente processati, consentono la costruzione delle cosiddette *staricase functions* al variare della soglia dei discriminatori degli ASIC.

Il codice sorgente del modulo *trig_cnt* è stato realizzato su sistema di sviluppo Xilinx presso OACT, dove si è inoltre provveduto allo sviluppo di un opportuno *testbench* per la verifica funzionale del modulo tramite simulatore *iSim*.

2.6 Modulo di gestione della modalità di funzionamento

Il modulo *mode_sel* è adibito alla gestione della modalità di funzionamento dell'FPGA. Le due modalità consentite sono: *stand-alone mode* e *BEE mode*. Nella prima modalità il segnale che produce l'avvio dell'acquisizione nel modulo *acquire* ("acq_start") è rappresentato dal trigger di PDM in uscita al modulo di generazione dei trigger (*trig_gen*), mentre nella seconda modalità l'avvio dell'acquisizione viene segnato dalla ricezione di un segnale di trigger esterno proveniente dalla BEE (sulla linea "trig_a").

La schematizzazione dell'entità strutturale del modulo è rappresentata in Fig. 7.

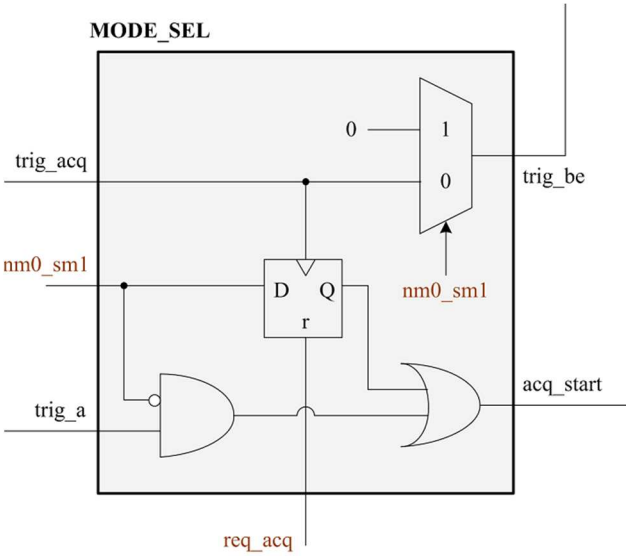


Fig. 7. Diagramma a blocchi del modulo mode_sel.

I due parametri di configurazione del modulo *mode_sel* gestiti dal microprocessore sono elencati di seguito:

- **nm0_sm1**: determina la modalità di funzionamento dell'FPGA ("1" logico per modalità *stand-alone* e "0" logico per modalità *BEE*).
- **req_acq**: nella modalità di funzionamento *stand-alone*, rappresenta il segnale di richiesta dei dati da parte del processore, che abilita il modulo alla ricezione dei PDM trigger per l'avvio dell'acquisizione.

Il codice sorgente del modulo *mode_sel*, inizialmente sviluppato da Mindway, è stato perfezionato e finalizzato presso OACT. La verifica funzionale del modulo è stata testata direttamente su scheda FPGA, tramite l'ausilio dell'ILA Chipscope.

Le modifiche al firmware che interessano il modulo *mode_sel* sono descritte nelle sezioni 4.3, 4.15.

2.7 Modulo di generazione monostabili e mascheratura trigger

Il modulo *monostable_mask* è interamente dedicato al processamento dei 64 segnali di trigger in uscita ai discriminatori degli ASIC. Esso consente sia di migliorare l'efficienza dell'algoritmo di trigger nell'individuazione delle adiacenze tra pixel, che di agevolare i test sui moduli di generazione e di conteggio dei trigger.

La schematizzazione dell'entità strutturale del modulo è rappresentata in Fig. 8.

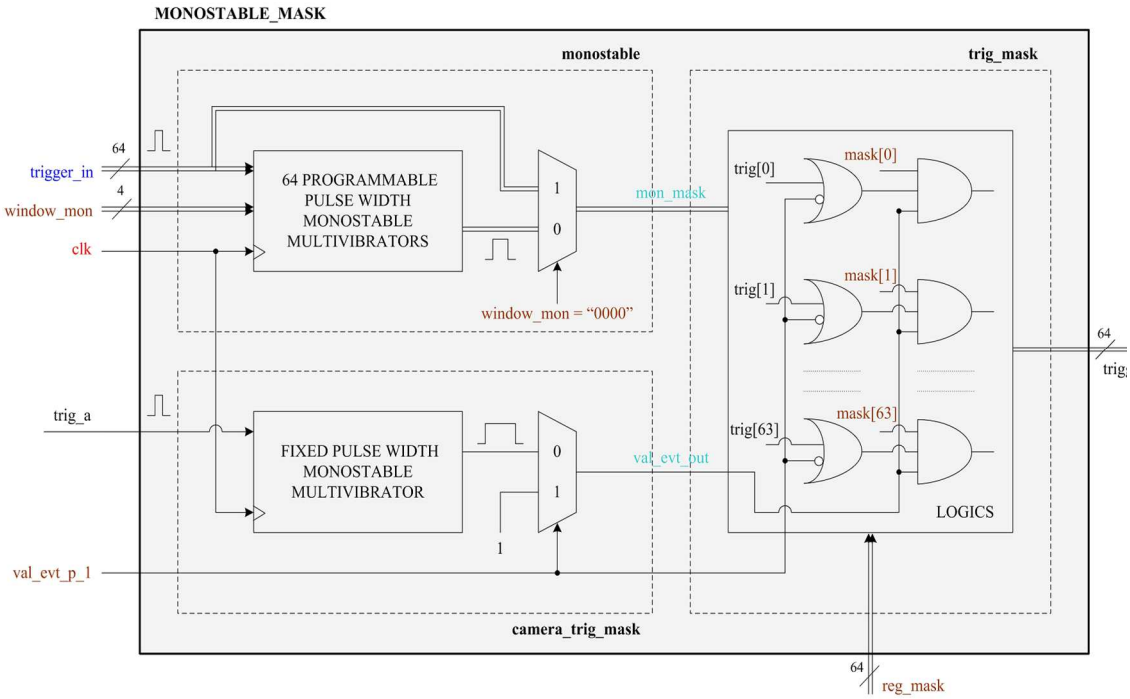


Fig. 8. Diagramma a blocchi del modulo monostable_mask.

Il modulo riceve in ingresso i trigger provenienti dai 2 CITIROC e fornisce in uscita 64 segnali di trigger (opportunamente modulati) che andranno a pilotare, in parallelo, sia il modulo di generazione del trigger di PDM, sia il modulo di conteggio dei trigger.

L'entità strutturale racchiude al suo interno tre componenti, ciascuno dei quali è dedicato alla realizzazione di una particolare funzione logica:

- *monostable*: realizza 64 multivibratori monostabili a larghezza di impulso programmabile per i 64 segnali di trigger.
- *trig_mask*: implementa una mascheratura logica dei segnali di trigger in uscita al blocco precedente.
- *camera_trig_mask*: consente la selezione del trigger esterno con durata fissa a 100ns oppure i 64 trigger interni.

I parametri di configurazione del modulo *monostable_mask* gestiti dal microprocessore sono elencati di seguito:

- **window_mon**: determina la durata dei segnali di trigger modulati.
- **reg_mask**: abilita/disabilita i segnali di trigger in uscita al modulo.
- **val_evt_p1**: consente la selezione dei trigger interni o del trigger esterno ("1" logico per selezionare i trigger provenienti dagli ASIC, "0" logico per selezionare il trigger proveniente dalla BEE).

Il codice sorgente del modulo *monostable_mask*, comprensivo dei 3 componenti interni, è stato sviluppato e implementato su sistema di sviluppo Xilinx presso OACT, dove si è inoltre provveduto alla realizzazione di un opportuno *testbench* per la verifica funzionale tramite simulatore *iSim* di ciascuno dei tre componenti interni al modulo nonché del modulo *monostable_mask* nella sua interezza.

Le modifiche al firmware che interessano il modulo *monostable_mask* vengono descritte nelle sezioni 4.6, 4.10, 4.11.

2.8 Generatori di clock

I segnali di clock che viaggiano all'interno dell'FPGA vengono generati da appositi *core block* integrati e programmabili (*Clocking Wizard*), e vengono successivamente smistati ai vari componenti e moduli funzionali che li richiedono.

Una schematizzazione semplificata dei vari moduli di generazione dei clock nella versione definitiva del progetto è illustrata in Fig. 9.

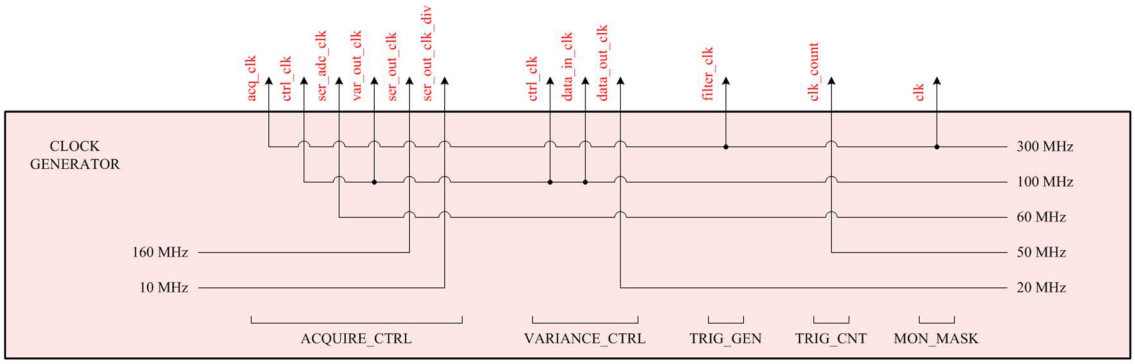


Fig. 9. Schematizzazione del modulo di generazione dei clock.

I segnali di clock prodotti in uscita ai blocchi *clock generator* sono dettagliati di seguito, a seconda del modulo funzionale di riferimento.

1) Modulo *acquire*:

- **acq_clk** (300MHz): campiona il segnale di inizio acquisizione nel modulo *acquire*.
- **ctrl_clk** (100MHz): governa il funzionamento generale della macchina a stati integrata all'interno del modulo *acquire*.
- **ser_adc_clk** (60MHz): scandisce l'acquisizione dei dati degli ADC.
- **var_out_clk** (100MHz): gestisce la comunicazione dei dati, prelevati dagli appositi buffer interni al modulo *acquire*, verso il modulo di varianza.
- **ser_out_clk** (160MHz): gestisce la comunicazione dei dati verso la memoria esterna o verso la BEE; allo stato attuale, il generatore fornisce in uscita per questo clock una frequenza reale di 150MHz.
- **ser_out_clk_div** (10MHz): rappresenta il clock di validazione dei dati di uscita; sono richiesti 64 cicli di questo clock per trasmettere un blocco di 64 dati in uscita.

2) Modulo *variance*:

- **ctrl_clk** (100MHz): detta il funzionamento generale della macchina a stati integrata all'interno del modulo *variance*.
- **data_in_clk** (100MHz): scandisce la comunicazione con il buffer di memoria dei dati di varianza in ingresso provenienti dal controllore delle acquisizioni.

- **data_out_clk** (20MHz): scandisce la comunicazione con il buffer di memoria dei dati di varianza in uscita verso il microprocessore.

3) Modulo *trig_gen*:

- **filter_clk** (300MHz): gestisce la sezione di filtraggio dei *glitch* nel modulo *trig_gen*; il clock agisce previa abilitazione del filtro tramite apposito parametro di configurazione trasmesso dal microprocessore (“filter_out_en”).

4) Modulo *trig_count*:

- **clk_count** (50MHz): rappresenta gli *step* temporali (20ns) del contatore che determina la larghezza della finestra di acquisizione per i conteggi di trigger.

5) Modulo *monostable_mask*:

- **clk** (300MHz): determina gli *step* temporali (3.3ns) legati alla larghezza degli impulsi di trigger nel modulo *monostable_mask*.

3. STRUMENTI DI SVILUPPO E SET-UP SPERIMENTALE

Gli strumenti utilizzati per lo sviluppo del firmware dell'FPGA comprendono:

- la piattaforma HW "VIVADO 15.2" (che integra un simulatore *iSim* e un analizzatore di stati logici ILA Chipscope per le verifiche su scheda);
- la piattaforma SW "Software Development Kit" (SDK), basata su ambiente di sviluppo integrato Eclipse, per la gestione di microprocessori embedded.

I vari file sorgenti dell'intero progetto dell'FPGA sono racchiusi in tre direttori: una prima cartella ("inaf_fe") include i file di sintesi, implementazione e programmazione del *design*; un secondo archivio ("inaf_fe_src") contiene il repositorio dei singoli componenti del progetto, richiamati all'interno del diagramma a blocchi dell'FPGA; infine, una terza cartella ("inaf_sdk") conserva i sorgenti del codice C utilizzati per la parte SW di controllo del microprocessore dell'FPGA.

Tutti i test dell'FPGA in modalità *stand-alone* (con trasmissione e ricezione tramite porta USB) che hanno permesso la verifica puntuale di ogni singola porzione di firmware, nonché tutte le procedure di calibrazione effettuate sulle 37 PDM del piano focale sono state rese possibili grazie allo sviluppo di un elaborato ed efficiente software di controllo di tipo multimediale, ideato, realizzato, perfezionato e collaudato in IASF Palermo da Osvaldo Catalano. La versione ultima di tale applicativo (utilizzabile su sistemi operativi successivi a Windows 7) con il quale sono stati eseguiti i vari test riportati nel presente report, è denominata "PDM_ver4" ed è disponibile nel direttorio online sul server condiviso dal team ASTRI. La schermata di avvio dell'applicazione è illustrata in Fig. 10.

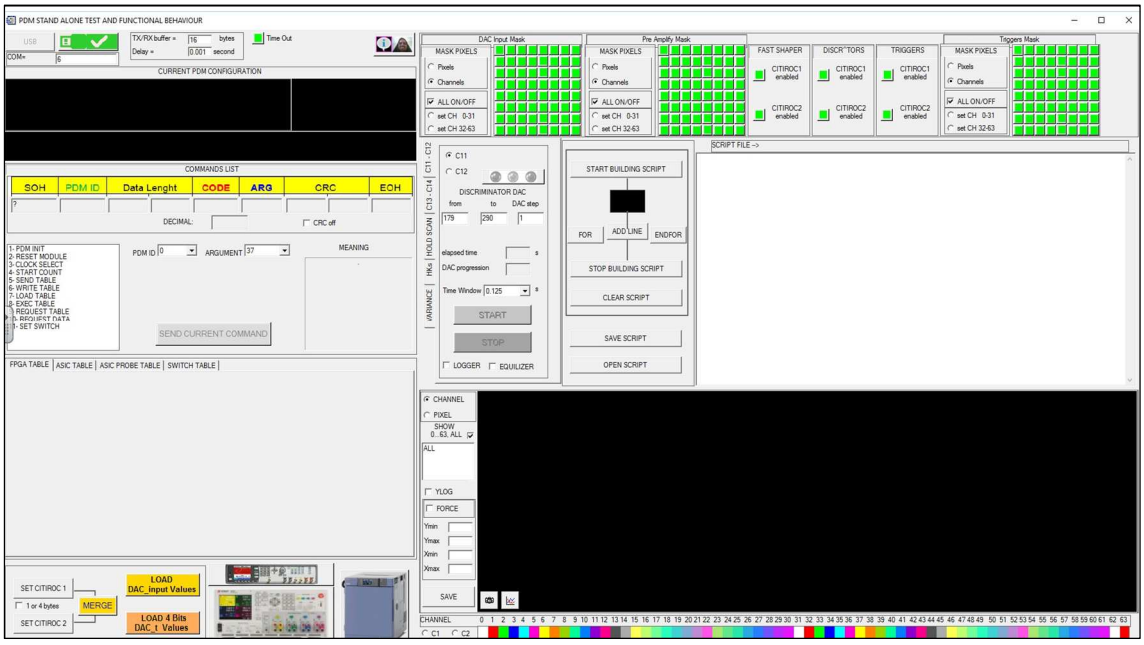


Fig. 10. Interfaccia principale del software di controllo dell'FPGA.

Il *set-up* sperimentale con cui è stato verificato il firmware dell'FPGA è rappresentato in Fig. 11. In particolare, un alimentatore Keysight N6000B (che include un mainframe e 4 moduli funzionali) è stato usato per fornire le quattro alimentazioni necessarie sia per la scheda FPGA (7.0V) che per la scheda ASIC ed i sensori di temperatura della scheda SiPM (3.6V, -3.3V, 5.6V). La tensione di alimentazione dei SiPM viene altresì fornita da un secondo generatore di tensione, un Agilent 6655A, in grado di produrre tensioni fino a 120V. Per eseguire i test sul sistema completo, è stato collegato ad uno dei due connettori della scheda ASIC un sensore MPPC Hamamatsu da 3×3mm^2, opportunamente coperto per le misure in *dark*, come illustrato in figura.

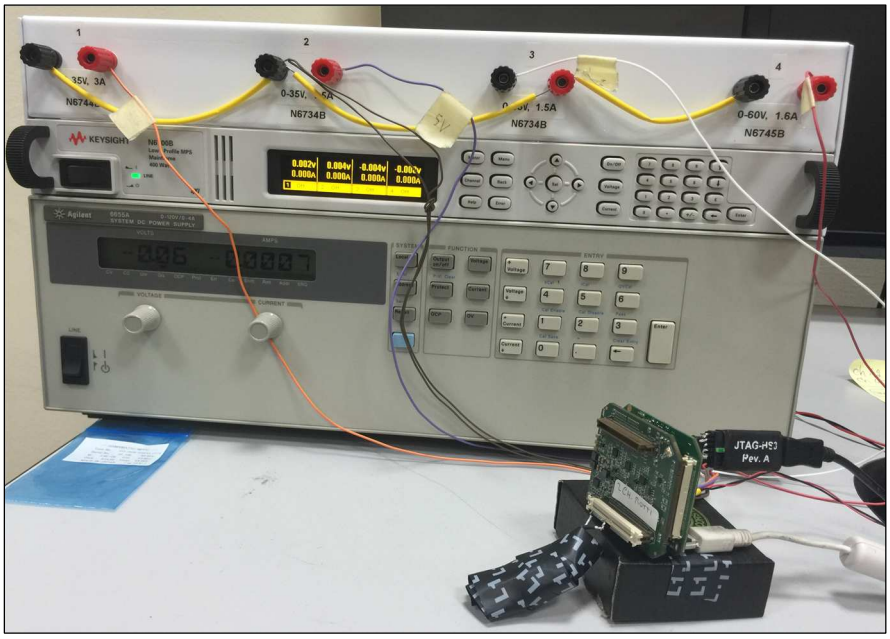


Fig. 11. Apparato sperimentale utilizzato per testare il firmware dell'FPGA.

La procedura di programmazione del firmware all'interno dell'FPGA può avvenire secondo due modalità distinte:

- programmazione nella RAM temporanea (.bit);
- programmazione nella memoria *flash* integrata su scheda (.mcs).

Entrambe le modalità avvengono in ambiente VIVADO tramite tool *hardware manager*, attraverso il quale si stabilisce un collegamento con l'FPGA.

Il file per la programmazione in RAM o su memoria *flash* viene generato a partire da un file di implementazione HW (.bit) in aggiunta al file SW relativo al codice (.elf), qualora sia presente il microprocessore. In Fig. 12 e in Fig. 13 vengono illustrati rispettivamente i comandi per programmare e "flashare" il dispositivo (device: xc7a100t). In assenza di software del microprocessore, il file di programmazione .mcs in memoria *flash* può anche essere generato a partire dal file .bit relativo alla sola implementazione.

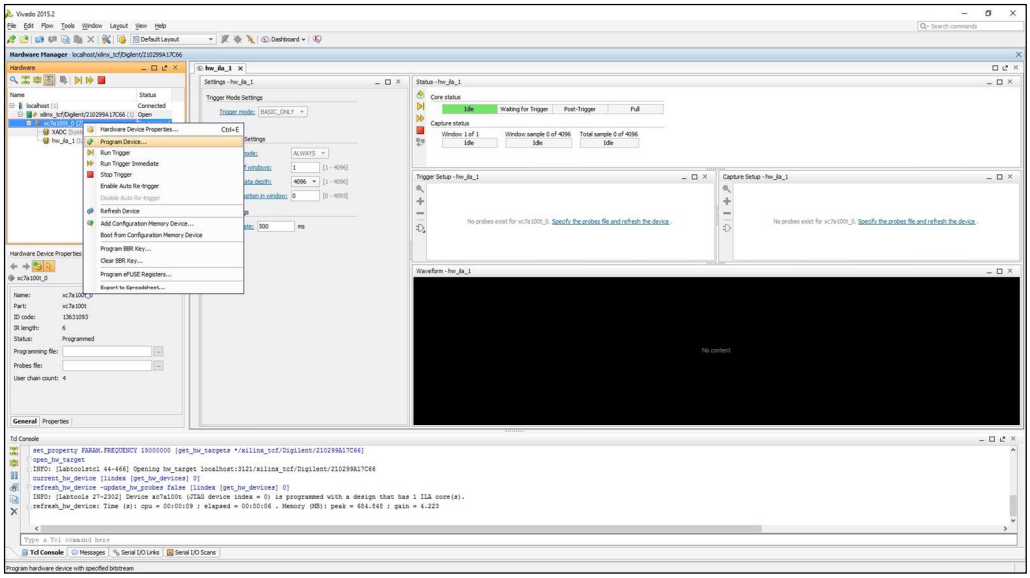


Fig. 12. Interfaccia hardware manager di VIVADO per la programmazione dell'FPGA.

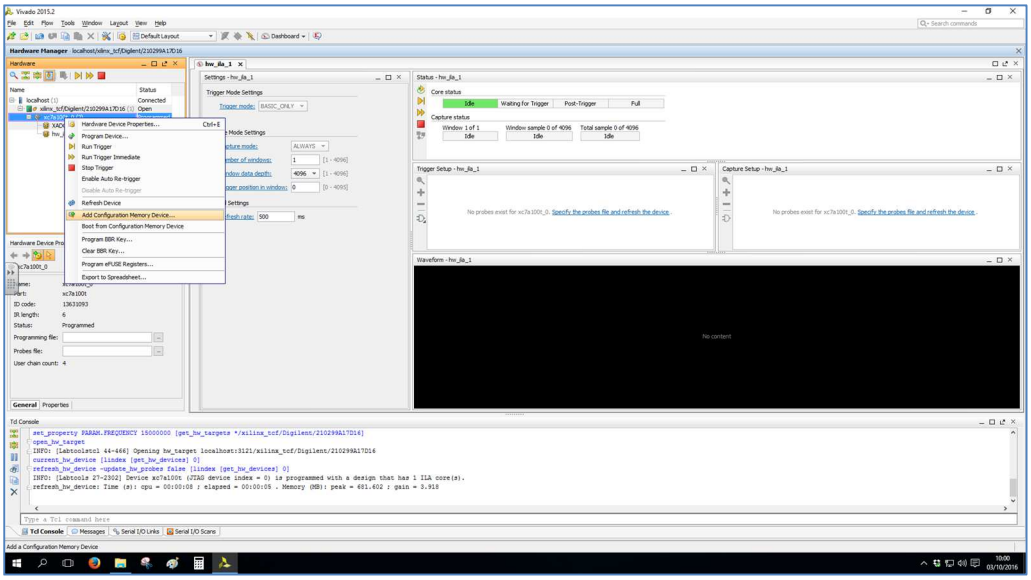


Fig. 13. Interfaccia hardware manager di VIVADO per la programmazione dell'FPGA.

La sezione successiva include un elenco dettagliato dei vari aggiornamenti firmware effettuati sia sulla parte HW in VIVADO che sul codice C in SDK.

()

4. AGGIORNAMENTI FIRMWARE DELL'FPGA

In questa sezione sono descritti in dettaglio i vari aggiornamenti firmware dell'FPGA. Il diagramma a blocchi del progetto completo è illustrato in Fig. 14.

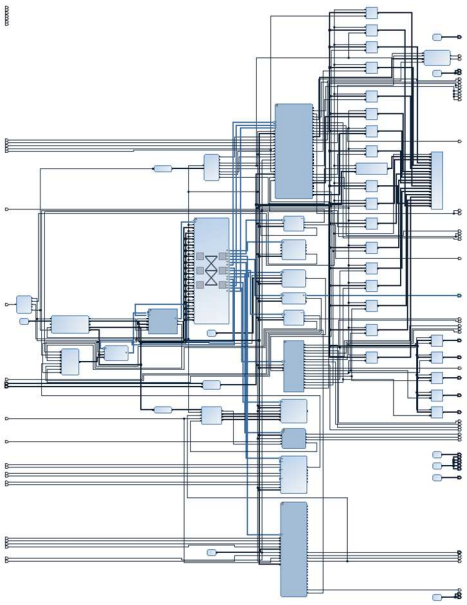


Fig. 14. Diagramma a blocchi del progetto completo dell'FPGA di front-end.

Esso comprende i blocchi legati alla gestione del processore MicroBlaze, inclusi GPIO (*inaf_gpio*) e registri (*inaf_fee_registers*), i moduli relativi alla trasmissione dei dati verso la BEE, vari blocchi *ila_sampler* di campionamento dei dati per l'ILA Chipscope, e la gerarchia *inaf_fee_core_block*, che rappresenta il cuore del progetto e racchiude i principali moduli funzionali della FEE.È presente, inoltre, il blocco *mw_fee_core_block* per la gestione dei dati di *housekeeping* degli ASIC CITIROC.

Il diagramma a blocchi dei principali moduli funzionali realizzati all'interno della struttura *inaf_fee_core_block* è rappresentato nella Fig. 15. Esso comprende al suo interno i blocchi fondamentali precedentemente descritti:

- *acquire_ctrl*
- *variance_ctrl*
- *trig_gen*
- *trig_cnt*
- *mode_sel*
- *monostable_mask*

In Fig. 16 viene illustrata l'allocazione delle risorse complessive utilizzate dall'FPGA.

Nelle pagine seguenti vengono riportati tutti gli aggiornamenti del firmware che hanno coinvolto i vari moduli HW nonché le sezioni del SW per la gestione del processore.

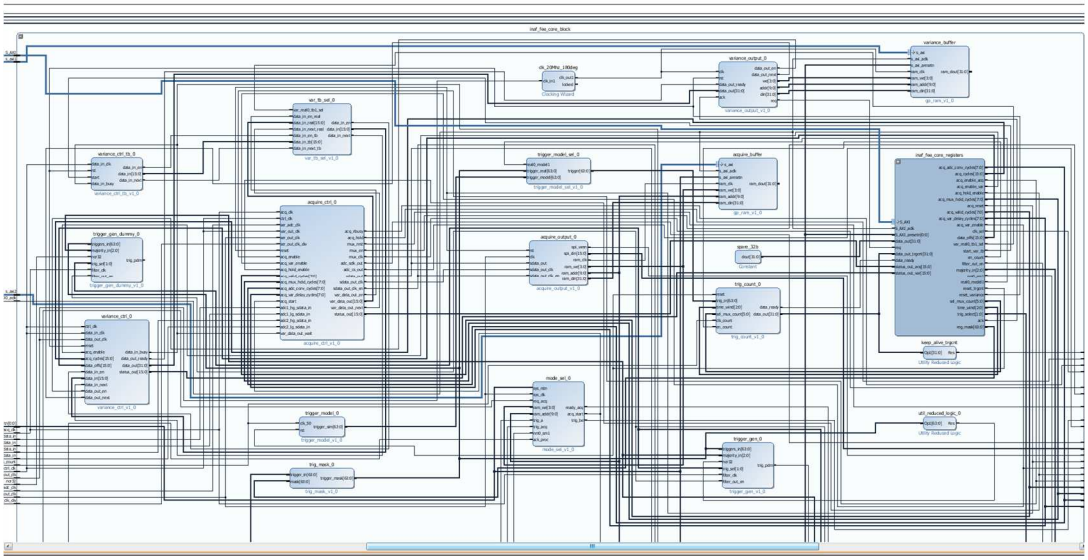


Fig. 15. Diagramma a blocchi dei principali moduli funzionali dell'FPGA realizzati all'interno della gerarchia inaf_fee_core_block.

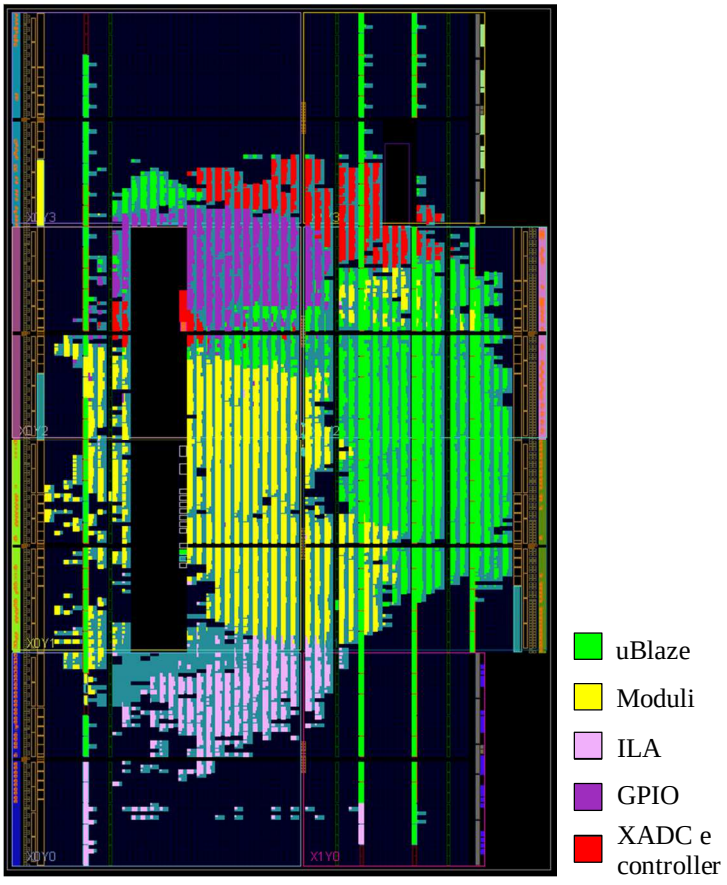


Fig. 16. Allocazione delle risorse complessive del progetto all'interno dell'FPGA Artix-7.

4.1 Acquisizione delle temperature dei sensori SiPM ()

Allo scopo di garantire la stabilizzazione dei segnali di selezione dei 10 sensori di temperatura della scheda SiPM, vengono utilizzati nel codice SW dei ritardi generati mediante semplici procedure di acquisizione a vuoto (modulo helloworld.c in SDK):

```
XsysMon_GetStatus(SysMonInstPtr);
      while ((XsysMon_GetStatus(SysMonInstPtr) & XSM_SR_EOS_MASK) !=
XSM_SR_EOS_MASK);
```

Dai test effettuati sulle acquisizioni delle temperature dei sensori si è riscontrata una non corretta selezione di alcuni sensori. Pertanto, sono state aggiunte ulteriori acquisizioni a vuoto all'interno delle procedure di acquisizione per stabilizzare il valore del selettore del multiplexer che gestisce la scansione delle temperature.

Inoltre, allo scopo di ridurre i tempi necessari ad una corretta selezione dei 10 sensori di temperatura, è stato seguito il codice Grey durante il passaggio tra un selettore e il successivo, in maniera tale da minimizzare le variazioni sugli ingressi di selezione del multiplexer.

In Fig. 17 sono illustrate graficamente, attraverso il software di controllo dell'FPGA, le acquisizioni dei dati di *housekeeping*, tra i quali figurano anche i valori degli ADC per le 10 temperature dei SiPM.

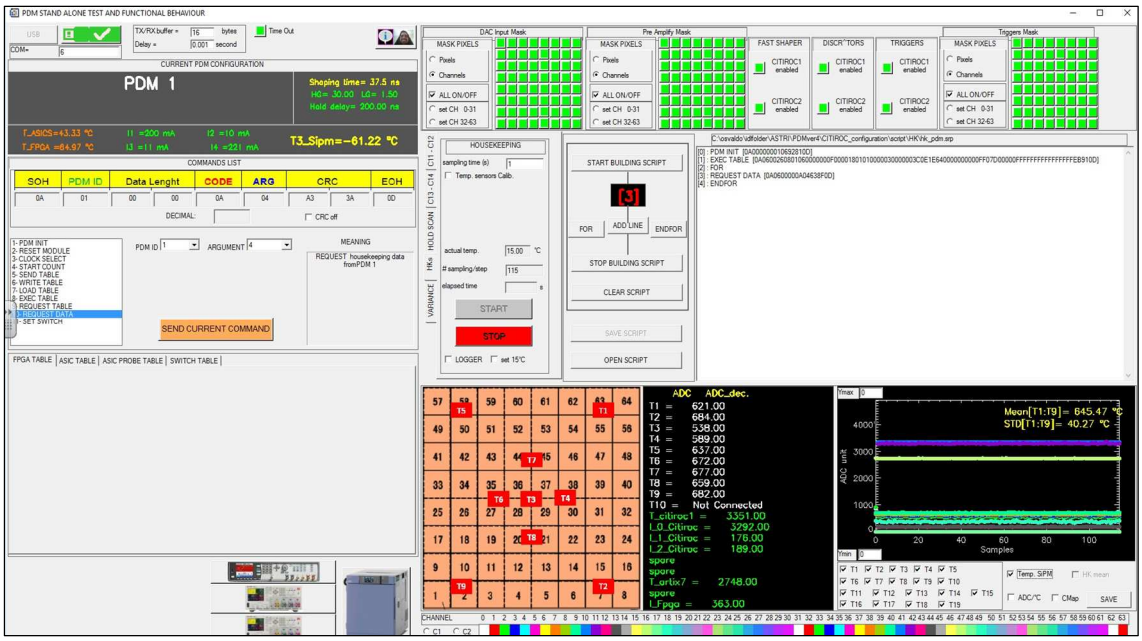


Fig. 17. Acquisizioni dei dati di housekeeping tramite il software di controllo.

4.2 Temporizzazione della RAM per l'invio dati scientifici ()

I blocchi HW all'interno dell'FPGA che gestiscono l'acquisizione dei dati scientifici ed il loro trasferimento verso la BEE sono fondamentalmente quattro: il modulo *acquire*, che sovrintende al processo di lettura dei dati, al loro ordinamento per pixel ed alla loro trasmissione seriale; i moduli *acquire_output* e *acquire_buffer*, addetti alla ricezione e alla memorizzazione dei dati scientifici prodotti; infine, il modulo *mode_sel*, che stabilisce la modalità di acquisizione e fornisce al uBLAZE il segnale di fine trasmissione.

In particolare, quest'ultimo blocco funge da interfaccia tra l'utente esterno e il modulo di acquisizione dati nella modalità di funzionamento *stand alone* dell'FPGA di *front-end*, ed è descritto in HW dal seguente codice VHDL:

```
if sys_rstn = '0' then
      rst_at <= '1';
      int_state <= x"0";
      req_acq_p1 <= '0';
      ready_acq <= '0';
elsif rising_edge(sys_clk) then
      req_acq_p1 <= req_acq;
      if int_state = x"0" then
            if (req_acq = '1' and req_acq_p1 = '0') then
                  int_state <= x"1";
                  rst_at <= '0';
            else
                  int_state <= x"0";
                  rst_at <= '1';
            end if;
      elsif int_state = x"1" then
            if (ram_we = x"F") and (ram_addr = "0001111111") then
                  int_state <= x"2";
                  ready_acq <= '1';
            else
                  int_state <= x"1";
            end if;
      elsif int_state = x"2" then
            if ack_proc = '1' then
                  ready_acq <= '0';
                  rst_at <= '1';
                  int_state <= x"0";
            else
                  ready_acq <= '1';
                  int_state <= x"2";
            end if;
      end if;
end if;
```

I dati scientifici vengono richiesti tramite il comando REQUEST DATA (evento scientifico) [R1], attivando il segnale "req_acq". Da questo momento la ricezione di un trigger determina l'abilitazione del segnale "acq_start" nel modulo "ACQUIRE", la conseguente attivazione del *peak detector* e la successiva conversione analogico-digitale dei segnali

relativi ai 32 canali per ciascun ASIC, sia per gli HG che per gli LG (in totale 128 dati a 12 bit).

I segnali prodotti vengono spediti dal modulo *acquire* in serie, ed attraverso il modulo *acquire_output* vengono parallelizzati e caricati in memoria (*acquire_buffer*). Quando il valore dell'indirizzo di memoria è pari a 127, tutti i dati sono stati memorizzati e pertanto possono essere letti dal processore e spediti in uscita tramite UART. Lo schema a blocchi che descrive tale processo è rappresentato in Fig. 18.

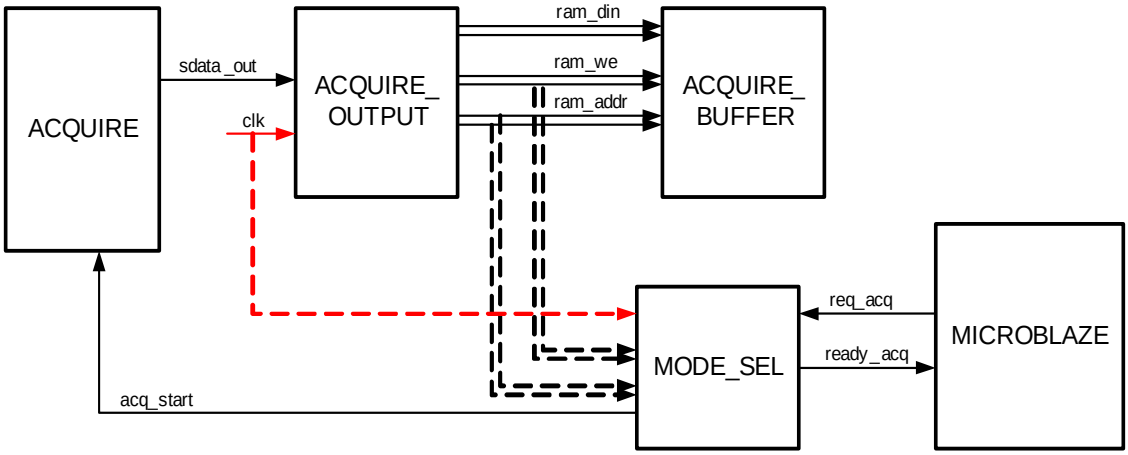


Fig. 18. Schema a blocchi che descrive il processo di memorizzazione dei dati scientifici.

Il blocco *mode_sel* si occupa di verificare se la memoria è stata caricata con tutti i 128 dati scientifici. In particolare, quando si verifica la condizione relativa all'`if` nel seguente stralcio di codice:

```
if (ram_we = x"F") and (ram_addr = "0001111111") then
            int_state <= x"2";
            ready_acq <= '1';
      else
            int_state <= x"1";
end if;
```

allora viene attivato il segnale `ready_acq`, che informa il MicroBlaze che può iniziare la lettura dalla memoria.

Analizzando le uscite del modulo acquire su ILA Chipscope, si è potuto verificare che i 128 dati scientifici provenienti dal modulo di acquisizione vengono caricati correttamente all'interno del buffer di memoria *acquire_buffer*, ma il segnale di fine memorizzazione, `ready_acq`, resta ancorato al valore 0. In sostanza, non viene verificata la condizione sopracitata.

In Fig. 19 è possibile notare che una frequenza del clock `sys_clk` pari a 100MHz per la macchina a stati presente nel modulo *mode_sel* non consente la verifica della condizione precedente.

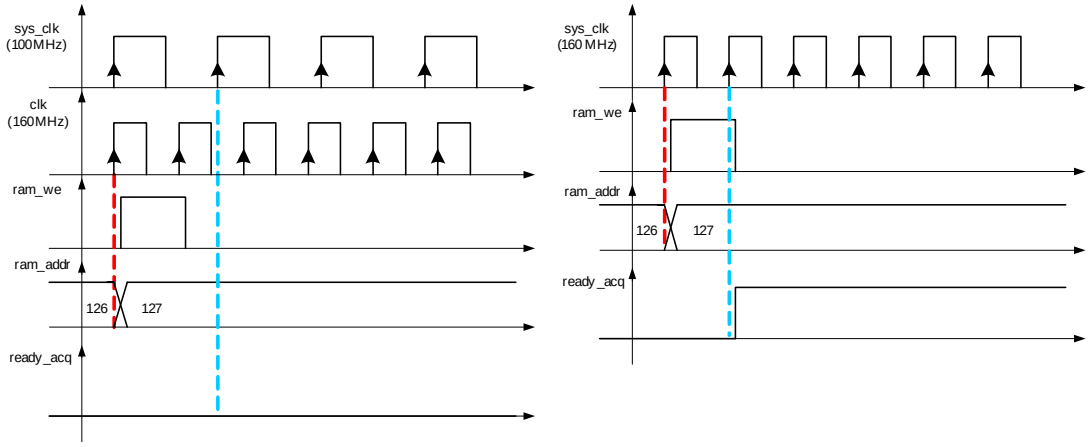


Fig. 19. Diagramma temporale prima (sinistra) e dopo (destra) la modifica effettuata.

Pertanto, per risolvere il problema è stata aumentata la frequenza del clock `sys_clk` dal valore originario di 100MHz al valore di 160MHz, con cui vengono scanditi i dati scientifici in uscita al modulo *acquire*. Tale modifica HW è illustrata nello schematico di Fig. 20. Nella precedente versione del firmware, il segnale `sys_clk` era collegato al clock "S_AXI_aclock" (100MHz); in seguito alla modifica, esso risulta dunque collegato al segnale di clock "ser_out_clk" (160MHz).

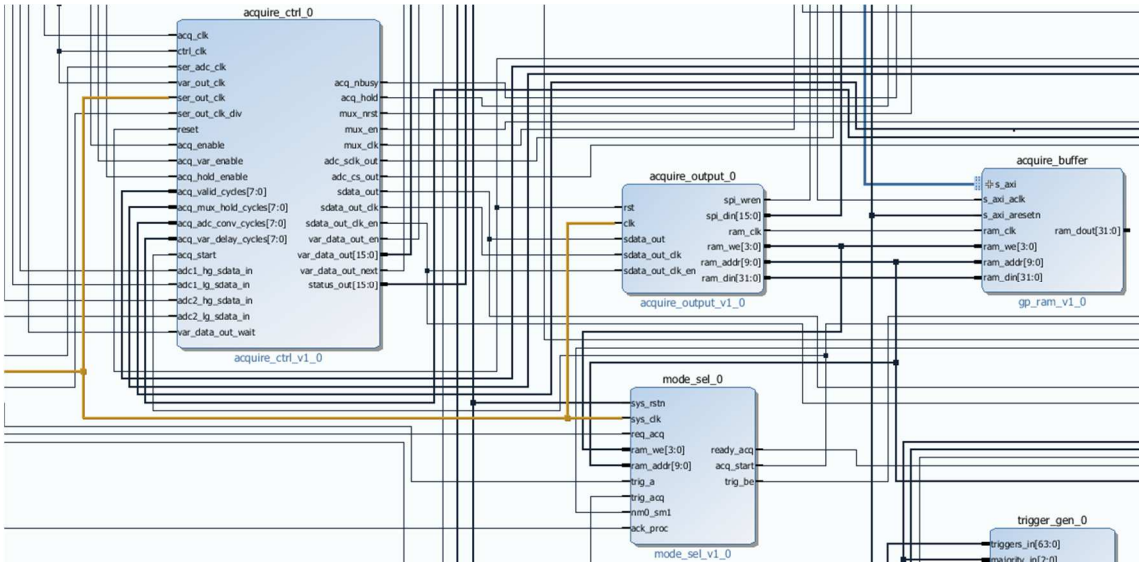


Fig. 20. Schematizzazione (linea gialla) del collegamento dei segnali di clock nei moduli acquire, acquire_output e mode_sel nella nuova versione del firmware.

In seguito a questa variazione, eseguendo il comando REQUEST DATA (evento scientifico) [R1], i dati prodotti dal modulo *acquire* vengono visualizzati correttamente.

4.3 Disabilitazione del segnale di trigger BEE in modalità SA ()

La modalità di gestione del trigger per l'avvio dell'acquisizione è denominata "autotrigger". Il modulo HW che governa tale modalità è il *mode_sel*, schematizzato in Fig. 21.

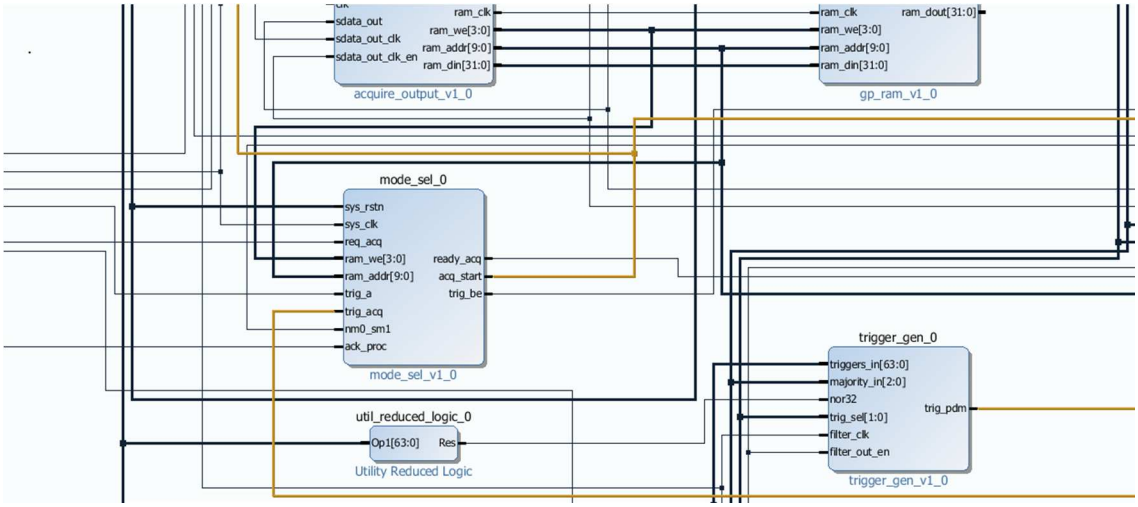


Fig. 21. Schematico del modulo mode_sel per la gestione della modalità di autotrigger.

Il diagramma a blocchi che illustra il principio di funzionamento del modulo *mode_sel* per la gestione del trigger, nella precedente versione del firmware, è riportato in Fig. 22.

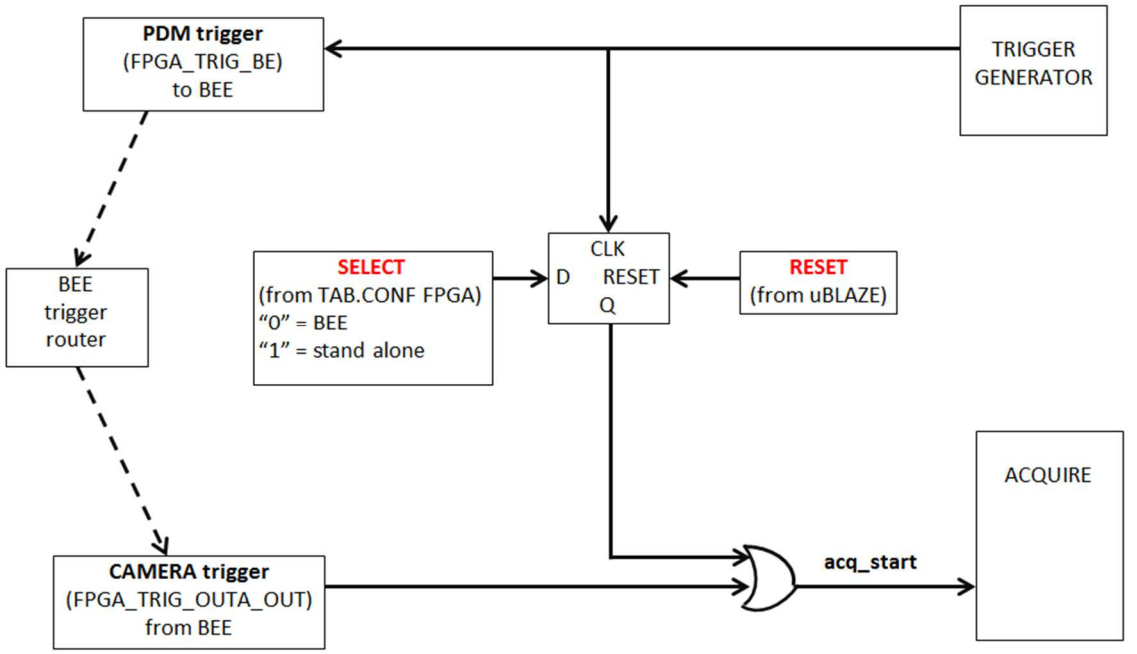


Fig. 22. Diagramma a blocchi del modulo mode_sel nella precedente versione del firmware.

In condizioni nominali, quando la PDM è collegata alla BEE, il segnale SELECT (associato alla porta "nm0_sm1" dello schematico in Fig. 21 ed immagine diretta del parametro "stand alone" nella tabella di configurazione dell'FPGA [R1]) verrà posto a "0". In questa modalità l'uscita Q del flip-flop resterà sempre a "0", mentre il segnale di camera trigger proveniente dalla BEE (sul pin FPGA_TRIG_OUTA_OUT del connettore della scheda FPGA) attiverà, attraverso una porta logica OR, il segnale "acq_start" del modulo *acquire*, abilitando il modulo all'acquisizione dati.

Di conseguenza, in questo caso (modalità BEE), sarà il trigger proveniente dalla BEE a riportare la linea di camera trigger a "0" dopo aver letto tutto il blocco dati di un evento scientifico (per tutte le 37 PDM). Il ritorno a "0" della linea "acq_start" riabilita il modulo *acquire* a ricevere il trigger successivo.

D'altro canto, per attivare la modalità di funzionamento *stand alone*, bisognerà comandare ad "1" il parametro "stand alone" nella tabella di configurazione dell'FPGA. A questo punto, il uBLAZE porrà ad "1", in sequenza, prima la linea RESET e subito dopo la linea SELECT. Così, il flip-flop sarà pronto a ricevere internamente un segnale di trigger che porti l'uscita Q ad "1" ma viene tenuto forzatamente a "0" dal segnale di RESET mantenuto attivo.

Successivamente, alla ricezione di un comando REQUEST DATA (evento scientifico), il uBLAZE riporterà la linea di RESET a "0". Ciò farà sì che al primo fronte di salita del segnale "PDM trigger", l'uscita Q del FF andrà ad "1", attivando (attraverso la porta logica OR) il segnale "acq_start" del modulo *acquire*. La linea Q resterà alta fino a quando il uBLAZE, dopo aver letto tutto il blocco dati di un evento ed averlo trasmesso al PC esterno attraverso la porta USB, riporterà la linea di RESET stabilmente ad "1", in attesa di un nuovo comando REQUEST DATA (evento scientifico).

L'assegnazione delle due uscite del modulo *mode_sel* relative ai segnali di trigger verso la BEE e verso il modulo *acquire* sono stabilite dalle seguenti righe di codice:

```
trig_be : out STD_LOGIC;
acq_start : out STD_LOGIC;
trig_be <= trig_acq when (nm0_sm1 = '0') else '0';
acq_start <= Q or trig_a;
```

dove `trig_a` è il camera trigger proveniente dalla BEE, mentre `trig_acq` è il trigger di PDM in ingresso al blocco.

Ora, nella modalità di funzionamento *stand alone*, visualizzando i segnali di input e di output del modulo *mode_sel* su ILA Chipscope si è riscontrato, in talune situazioni, che la linea di trigger proveniente dalla BEE permane nello stato logico alto, impedendo il corretto funzionamento del blocco. Infatti, qualora il segnale di camera trigger dovesse trovarsi ad "1", l'uscita della porta OR diverrebbe insensibile ai fronti di salita del trigger di PDM in inglesso al flip-flop, e l'acquisizione dei dati scientifici resterebbe bloccata.

Allo scopo di rimediare al suddetto inconveniente, durante il funzionamento del modulo in modalità *stand alone* (SELECT=1) il segnale di trigger che proviene dal pin del con-

nettore di collegamento FEE-BEE è stato disattivato. Più specificamente, mediante una modifica funzionale al codice VHDL del modulo *mode_sel*, è stata inserita una parte di logica combinatoria aggiuntiva in grado di inibire l'ingresso della porta OR relativo al camera trigger durante la modalità di funzionamento *stand alone*. Il nuovo diagramma a blocchi comprensivo della modifica apportata è illustrato in Fig. 23.

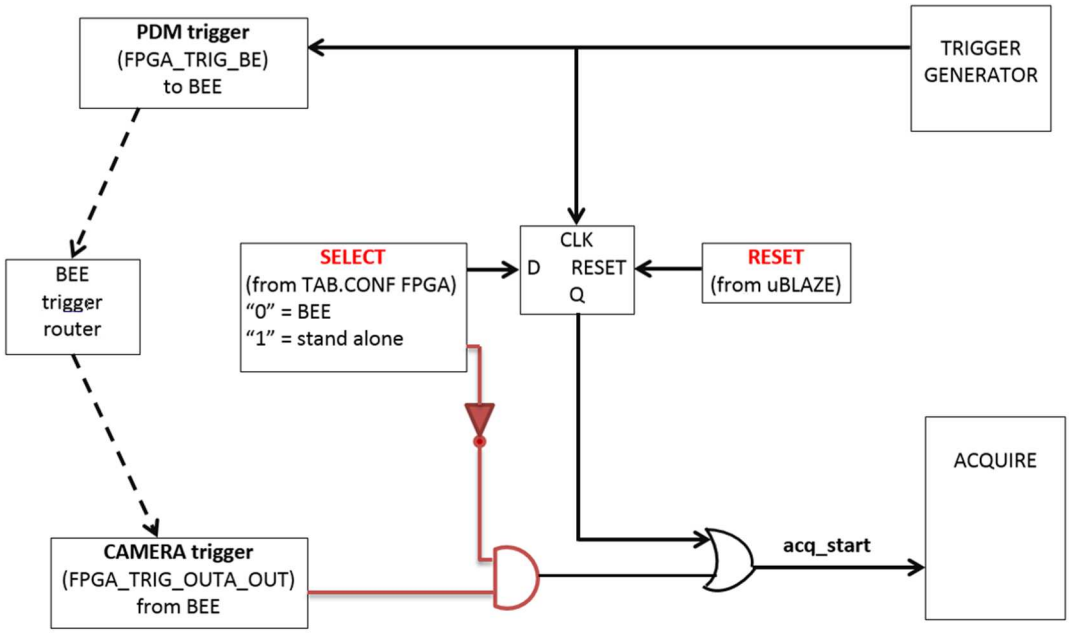


Fig. 23. Diagramma a blocchi del modulo mode_sel nella nuova versione del firmware.

In virtù della logica aggiuntiva evidenziata in rosso, il valore del segnale SELECT viene opportunamente invertito e posto in ingresso ad una porta AND, ottenendo l'esclusione del segnale proveniente dalla BEE nella modalità di funzionamento *stand alone*, anche qualora il pin del connettore di collegamento FEE-BEE dovesse trovarsi ad "1".

La variante circuitale rappresentata in Fig. 23 è stata realizzata in HW attraverso la seguente modifica al codice VHDL del modulo *mode_sel*:

```
trig_be   <= trig_acq when (nm0_sm1 = '0') else '0';
acq_start <= Q or (trig_a and (not nm0_sm1));
```

Nel caso in cui il segnale di SELECT viene posto a "0" (in modalità BEE), l'acquisizione dati viene correttamente attivata dal segnale di trigger inviato dalla BEE e posto nel secondo ingresso della porta AND.

4.4 Modifica sull'ordine dei dati scientifici dal modulo *acquire* ()

Dalle simulazioni iSIM sul modulo *acquire*, si è potuto constatare un problema legato alla sequenza dei 128 dati scientifici generati durante la trasmissione seriale verso la BEE (64 dati HG e 64 dati LG, entrambi ordinati per pixel). In particolare, con la precedente versione del codice, si verifica una inversione degli ultimi due dati seriali di ciascuna sequenza, ovvero il 64-esimo dato HG ed il 64-esimo dato LG risultano invertiti. Ciò è dovuto ad uno sfasamento associato al segnale di clock `clk_out`, che scandisce l'invio dei dati scientifici in uscita dal modulo *acquire*.

Il codice VHDL legato alla trasmissione dei dati seriali verso la BEE è il seguente:

```
signal data_buf_cnt_out : integer range 0 to (ACQ_CYCLE_NUM*2)-1;

data_reg_mux <= data_a_out when (data_buf_cnt_out<64) else data_b_out;
data_reg <= data_reg_mux when ((clk_out'event and clk_out='1') and
data_buf_out_reg_en='1');

      process(clk_out, data_buf_cnt_out_en)
      begin
            if (data_buf_cnt_out_en='0') then
            elsif (clk_out'event and clk_out='1') then
                  data_a_out<=data_a_buf(data_buf_cnt_out);
            end if;
      end process;

      process(clk_out, data_buf_cnt_out_en)
      begin
            if (data_buf_cnt_out_en='0') then
            elsif (clk_out'event and clk_out='1') then
                  data_b_out<=data_b_buf(data_buf_cnt_out);
            end if;
      end process;

data_buf_cnt_out<=0 when (data_buf_cnt_out_en='0') else
data_buf_cnt_out+1 when ((clk_out'event and clk_out='1') and
data_buf_cnt_out_stop='0');
```

Nello specifico, il segnale `data_buf_cnt_out` rappresenta una variabile con cui vengono conteggiati i dati da trasmettere, e può assumere valori interi da 0 a 127 (la costante `ACQ_CYCLE_NUM` è fissata a 64). I dati da spedire in seriale vengono prelevati da due buffer a 16 bit (uno per i dati LG, `data_a_buf`, ed uno per gli HG, `data_b_buf`), ed assegnati rispettivamente ai segnali `data_a_out` e `data_b_out` sul fronte di salita del clock `clk_out`. Ad ogni fronte di salita di `clk_out`, la variabile `data_buf_cnt_out` con cui vengono prelevati i dati viene incrementata, sicché i primi 64 dati HG presenti in `data_a_out` vengono accodati ai successivi 64 dati LG presenti in `data_b_out` ed assegnati al segnale `data_reg_mux`, da cui vengono prelevati al successivo fronte di Salita del clock, posti nel segnale `data_reg` e spediti in uscita in seriale verso la BEE. In Fig. 24 è illustrata la simulazione funzionale post-implementation relativa all'inizio della trasmissione seriale. Il segnale "sdata_out_clk_en" abilita la trasmissione dei bit di ogni singolo dato. La *radix* dei dati visualizzati è impostata in *unsigned decimal*.

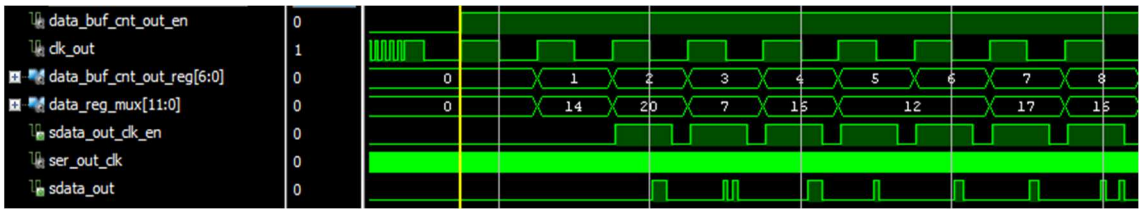


Fig. 24. Simulazione funzionale post-implementation tramite il tool iSIM relativa all'inizio della trasmissione seriale dei dati nella precedente versione del codice.

La variabile `data_buf_cnt_out` che scandisce il conteggio rappresenta, altresì, l'indice dei due buffer `data_a_buf` e `data_b_buf` da cui vengono prelevati i dati paralleli ad ogni fronte di salita di `clk_out` (all'interno dei due `process` del codice). Tale variabile viene utilizzata in parallelo anche per determinare la condizione con cui accodare al segnale `data_reg_mux` i 64 dati HG ed i 64 dati LG:

```
data_reg_mux <= data_a_out when (data_buf_cnt_out<64) else data_b_out;
```

Più specificamente, per valori della variabile `data_buf_cnt_out` compresi fra 0 e 63 vengono assegnati a `data_reg_mux` i 64 dati HG, mentre per valori della variabile superiori a 64 vengono accodati i restanti 64 dati LG.

Tuttavia, a causa della sequenzialità delle istruzioni nel codice, nello stesso momento in cui ogni singolo dato viene prelevato dai due buffer (sul fronte di salita di `clk_out`) la variabile `data_buf_cnt_out` viene incrementata. Di conseguenza, la condizione che stabilisce l'accodamento dei dati in `data_reg_mux` viene riferita al valore successivo della variabile `data_buf_cnt_out`, cosicché sul fronte di salita del clock i dati vengono caricati correttamente nei 2 buffer di uscita, ma non vengono correttamente accodati in uscita a causa dello *shift* operato in parallelo sulla variabile di conteggio. Ciò comporta un disallineamento dell'ultimo dato di ciascuna delle due sequenze.

Ad esempio, con riferimento alla Fig. 24, sul fronte di salita del clock `clk_out` immediatamente successivo a quello indicato dalla linea gialla, la variabile `data_buf_cnt_out` presenta il valore iniziale "0" e così vengono trasferiti rispettivamente in `data_a_out` e `data_b_out` i primi dati delle sequenze HG ed LG, ovvero i valori contenuti in `data_a_buf(0)` ed in `data_b_buf(0)`. Simultaneamente, tuttavia, la variabile del contatore viene incrementata al valore "1" e la condizione legata all'inserimento del primo dato di uscita in `data_reg_mux` viene a dipendere da questo nuovo valore della variabile, e farà sì che venga correttamente inserito (e trasferito in uscita al successivo fronte del clock) il primo dato immagazzinato in `data_a_out`, poiché la prima condizione `data_buf_cnt_out<64` viene verificata, in quanto `1<64`. Tale *shift* nella variabile di conteggio viene progressivamente mantenuto durante le successive letture dei dati. Non appena si giunge al 64-esimo dato, ai segnali `data_a_out` e `data_b_out` vengono rispettivamente assegnati i dati `data_a_buf(63)` e `data_b_buf(63)` ma, nello stesso istante, la variabile di conteggio si porta al valore 64, e dunque la condizione `64<64` non risulta più verificata. Questo comporta che il 64-esimo dato della prima sequenza HG contenuto in `data_a_buf(63)` non venga correttamente inserito in `data_reg_mux`,

a cui verrà assegnato invece il valore `data_b_buf(63)`, che conterrà dunque l'ultimo dato dell'altra sequenza (LG).

Analogamente, lo *shift* nella variabile di conteggio verrà ad alterare anche il 128-esimo dato seriale trasmesso, ovvero il 32-esimo dato della sequenza LG. Infatti, in virtù della assegnazione iniziale sulla variabile di conteggio

```
signal data_buf_cnt_out : integer range 0 to (ACQ_CYCLE_NUM*2)-1;
```

non appena `data_buf_cnt_out` raggiungerà il valore 128, verrà automaticamente resettata al valore "0". Ciò implica che il 64-esimo dato della seconda sequenza LG contenuto in `data_b_buf(127)`, non venga correttamente inserito in `data_reg_mux`, poiché la prima condizione `data_buf_cnt_out<64` viene ora nuovamente verificata, in quanto `0<64`. Così, a `data_reg_mux` verrà invece assegnato il valore memorizzato in `data_a_buf(127)`, che corrisponde al primo dato della precedente sequenza HG.

Nelle simulazioni riportate in Fig. 25 è possibile riscontrare entrambe le inversioni degli ultimi dati delle sequenze HG ed LG. In particolare, come si evince nel pannello superiore, in corrispondenza del valore 64 della variabile `data_buf_cnt_out` viene assegnato in `data_reg_mux` il valore errato ("1") corrispondente al primo dato della sequenza LG, anziché il valore corretto relativo al 32-esimo dato HG (16), che viene invece inserito nell'ultima posizione della sequenza LG, come si può osservare nel pannello inferiore, ovvero quando la variabile di conteggio `data_buf_cnt_out` viene azzerata. I dati presenti in `data_reg_mux`, preordinati per pixel, vengono trasferiti all'uscita seriale "sdata_out" al successivo fronte di salita del segnale di clock. Naturalmente, tale inversione si verifica ad ogni trasferimento dei dati scientifici verso la BEE.

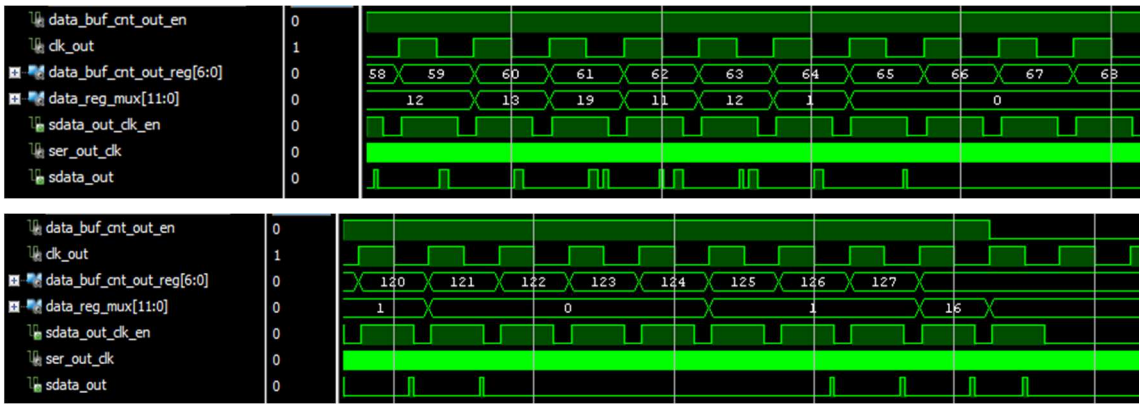


Fig. 25. Simulazione funzionale post-implementation tramite il tool iSIM relativa alla fine della trasmissione seriale dei 64 dati HG (pannello superiore) e dei 64 dati LG (pannello inferiore) nella precedente versione del codice.

Vale la pena notare che, poiché la variabile `data_buf_cnt_out` assume 128 valori, l'indice dei due buffer `data_a_buf` e `data_b_buf` (che coincide con tale variabile) assumerà di volta in volta i medesimi valori nonostante i dati contenuti nei due buffer sia-

no soltanto 64 per ogni evento (64 nel buffer HG e 64 nel buffer LG). Questo comporta che i dati immagazzinati nei buffer vengano ad essere scritti una seconda volta non appena la variabile di conteggio abbia superato il 64-esimo valore. In altre parole, con riferimento al buffer HG, i relativi 64 dati verranno memorizzati in `data_a_buf(0)`, `data_a_buf(1)`,…, `data_a_buf(63)` e, successivamente, anche in `data_a_buf(64)`, `data_a_buf(65)`,…, `data_a_buf(127)`. Discorso analogo per i dati LG.

Per risolvere l'inconveniente delle due inversioni di dati HG ed LG è stato modificato opportunamente il codice VHDL del modulo *acquire*, relativamente alle due istruzioni che coinvolgono rispettivamente la condizione su `data_buf_cnt_out` all'interno della struttura `when`, e l'assegnazione iniziale della medesima variabile tramite la costante `ACQ_CYCLE_NUM`. In particolare, per evitare la prima inversione sull'ultimo dato HG è stata effettuata la seguente modifica al codice sorgente:

```
data_reg_mux <= data_a_out when (data_buf_cnt_out<65) else data_b_out;
data_reg <= data_reg_mux when ((clk_out'event and clk_out='1') and
data_buf_out_reg_en='1');
```

In virtù della suddetta variazione, per quanto discusso in precedenza, la condizione su `data_buf_cnt_out` continua ad essere verificata anche durante la lettura del 64-esimo dato del buffer HG (contenuto in `data_a_buf(63)`), che verrà pertanto correttamente trasferito in `data_reg_mux`, e dunque successivamente trasmesso al segnale di uscita "sdata_out", poiché la condizione `data_buf_cnt_out<64` viene verificata anche sul 64-esimo dato di uscita, in quanto `64<65`.

Per evitare che venga alterato anche l'ultimo dato della catena LG è stata effettuata una seconda modifica alla riga di codice che coinvolge la definizione della variabile di conteggio, incrementando a 128 il suo massimo valore intero ottenibile:

```
signal data_buf_cnt_out : integer range 0 to (ACQ_CYCLE_NUM*2);
```

In tal modo, l'azzeramento della variabile verrà effettuato al raggiungimento del valore 129, consentendo la corretta assegnazione a `data_reg_mux` del dato seriale contenuto in `data_a_buf(127)` ed inserito in `data_b_out`, che rappresenta l'ultimo dato della sequenza LG da trasmettere.

Le simulazioni riportate in Fig. 26 e relative alla nuova implementazione funzionale del modulo *acquire* mostrano come venga evitata l'inversione dei dati durante la trasmissione seriale verso la BEE. In particolare, come si evince nel pannello superiore, in corrispondenza del valore 64 della variabile `data_buf_cnt_out` viene assegnato in `data_reg_mux` il valore corretto (16), relativo all'ultimo dato della sequenza HG. Analogamente, in corrispondenza del valore 128 di `data_buf_cnt_out` (ovvero prima che la variabile venga azzerata), viene inserito in `data_reg_mux` il valore corretto ("1"), relativo al 32-esimo dato della sequenza LG, garantendo il corretto trasferimento dell'intera catena di dati scientifici. Come prima, i dati presenti in `data_reg_mux` vengono trasferiti all'uscita seriale "sdata_out" al successivo fronte di salita del segnale di clock.

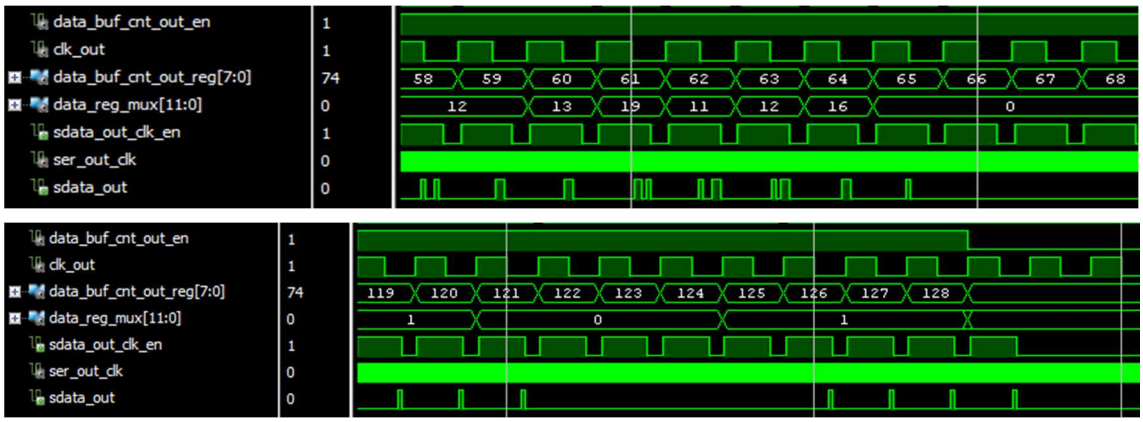


Fig. 26. Simulazione funzionale post-implementation tramite il tool iSIM relativa alla fine della trasmissione seriale dei 64 dati HG (pannello superiore) e dei 64 dati LG (pannello inferiore) nella versione modificata del codice.

Le suddette modifiche effettuate al codice sorgente del modulo *acquire* per risolvere i problemi descritti nel presente paragrafo non hanno alcuna influenza sulla modalità di trasmissione dei dati di varianza in *free-running* in assenza di trigger, in quanto coinvolgono esclusivamente le istruzioni legate alla trasmissione seriale verso la BEE, e mantengono pertanto inalterato il flusso di dati paralleli verso il modulo *variance*.

4.5 Modifiche all'algoritmo di generazione del trigger di PDM ()

La generazione del segnale di trigger di PDM viene effettuata dal modulo *trig_gen*, che realizza un algoritmo di controllo delle adiacenze sui 64 segnali di trigger provenienti dai 2 CITIROC. Rispetto alla versione originaria dell'algoritmo, che includeva esclusivamente il modulo di generazione del trigger, è stata prevista ed implementata inizialmente una sezione logica aggiuntiva che prevede la possibilità di assegnare al trigger di PDM il segnale proveniente dall'algoritmo di generazione oppure il segnale costituito semplicemente dalla OR logica dei 64 trigger provenienti dagli ASIC.

Il diagramma a blocchi della versione iniziale del modulo *trig_gen* con tale sezione aggiuntiva è rappresentato in Fig. 27, dove è stato sostanzialmente introdotto in uscita un multiplexer comandato dal parametro di configurazione aggiuntivo "trig_sel".

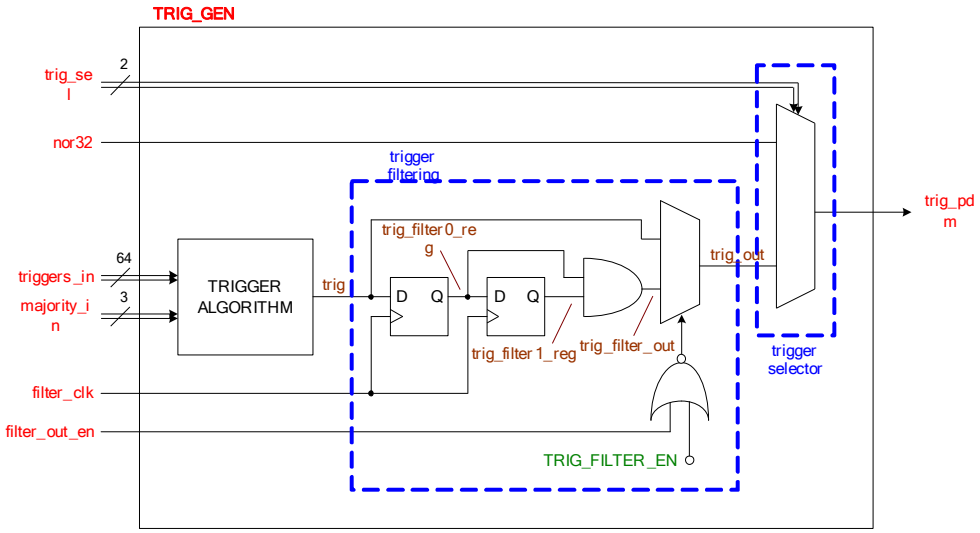


Fig. 27. Diagramma a blocchi iniziale relativo al modulo di generazione del trigger di PDM.

Il trigger di PDM in uscita al modulo è legato ai valori dei bit assegnati al parametro di configurazione "trig_sel" in accordo alla tabella seguente:

trig_sel	trig_pdm
00	*high-Z*
01	nor32
10	trigger
11	*high-Z*

Il codice VHDL dell'algoritmo di trigger, comprensivo della sezione aggiuntiva per la selezione del trigger in uscita, è riportato nelle seguenti righe:

```
entity trigger_gen is
   port (
        triggers_in    : in std_logic_vector(63 downto 0);
        majority_in    : in std_logic_vector(2 downto 0);
        nor32          : in std_logic;
        trig_sel       : in std_logic_vector(1 downto 0); -- SELECTOR
        filter_clk     : in std_logic;
        filter_out_en  : in std_logic;
        trig_pdm       : out std_logic
        );
end entity trigger_gen;

architecture arc_00 of trigger_gen is

constant TRIG_FILTER_EN : std_logic := '0';

type array16_type    is array(integer range <>) of
                        std_logic_vector(15 downto 0);
type ch_sorter_type  is array(integer range <>) of
                        integer range 0 to 63;

signal ch_sorter_buf : ch_sorter_type(0 to 63) :=
      (
             2,  0,  7,  1, 56, 58, 61, 63,
             6,  4,  5,  3, 60, 62, 57, 59,
            11,  9,  8, 10, 55, 53, 52, 54,
            15, 13, 12, 14, 51, 49, 48, 50,
            18, 16, 17, 19, 46, 44, 45, 47,
            22, 20, 21, 23, 42, 40, 41, 43,
            27, 25, 30, 28, 33, 35, 36, 38,
            31, 29, 26, 24, 37, 39, 32, 34
      );

signal trig             : std_logic;
signal trig_out         : std_logic; -- TRIGGER ALGORITHM OUTPUT
signal majority         : integer range 0 to 7;
signal trig_filter0_reg : std_logic;
signal trig_filter1_reg : std_logic;
signal trig_filter_out  : std_logic;

begin

trig_out <= trig when (TRIG_FILTER_EN='0' or filter_out_en='0')
                 else trig_filter_out;

majority <= conv_integer(majority_in);

-- TRIGGER FILTER ***************************************************

trig_filter0_reg <= trig when (filter_clk'event and filter_clk='1');
trig_filter1_reg <= trig_filter0_reg when
                   (filter_clk'event and filter_clk='1');
trig_filter_out  <= trig_filter1_reg and trig_filter0_reg;
```

```
-- TRIGGER SELECT ******************************************************

process (trig_sel, nor32, trig_out)
begin
case trig_sel is
   when "10"   => trig_pdm <= trig_out;
   when "01"   => trig_pdm <= nor32;
   when others => trig_pdm <= 'Z'; -- HIGH IMPEDANCE
end case;
end process;

-- TRIGGER ALGORITHM ***************************************************

process (triggers_in, majority)

type chains_type is array(integer range <>)
                    of std_logic_vector(24 downto 0);
type pixels_type is array(integer range <>)
                    of integer range -13 to 13;
type results_type is array(integer range <>)
                    of integer range -6 to 6;

variable chains         : chains_type(0 to 63);
variable chains_offsets : results_type(0 to 8);
variable pixels_offsets : pixels_type(0 to 8);

variable pixel_chain  : integer range 0 to 63;
variable pixel_offset : integer range 0 to 143;
variable chains_count : integer range 0 to 25;
variable chain_offset : integer range 0 to 24;

variable ch_sorter_ptr_i : integer range 0 to 63;
variable ch_sorter_ptr_o : integer range 26 to 117;

variable pixels  : std_logic_vector(143 downto 0);  -- MATRICE 12 x 12
variable trigger : std_logic;

begin

      pixels:= (others=>'0');

      for i in 0 to 7 loop

            for k in 0 to 7 loop

                  ch_sorter_ptr_i:= (i * 8) + k;
                  ch_sorter_ptr_o:= (i * 12) + k + 26;
                  pixels(ch_sorter_ptr_o):=
                  triggers_in(ch_sorter_buf(ch_sorter_ptr_i));

            end loop;

      end loop;
```

```
pixels_offsets(0):= 0;
pixels_offsets(1):= 1;
pixels_offsets(2):= 13;
pixels_offsets(3):= 12;
pixels_offsets(4):= 11;
pixels_offsets(5):= -1;
pixels_offsets(6):= -13;
pixels_offsets(7):= -12;
pixels_offsets(8):= -11;

chains_offsets(0):= 0;
chains_offsets(1):= 1;
chains_offsets(2):= 6;
chains_offsets(3):= 5;
chains_offsets(4):= 4;
chains_offsets(5):= -1;
chains_offsets(6):= -6;
chains_offsets(7):= -5;
chains_offsets(8):= -4;

for i in 0 to 7 loop

      for k in 0 to 7 loop

            pixel_chain:= (i * 8) + k;
            pixel_offset:= (i * 12) + k + 26;

            chains(pixel_chain):= (others=>'0');

            for j in 0 to 8 loop

                  for t in 0 to 8 loop

            chain_offset:=
            12 + chains_offsets(j) + chains_offsets(t);

if ((
pixels(pixel_offset) and
pixels(pixel_offset + pixels_offsets(j)) and
pixels(pixel_offset + pixels_offsets(j) + pixels_offsets(t)))
= '1') then
       chains(pixel_chain)(chain_offset):='1';
end if;

                  end loop;

            end loop;

      end loop;

end loop;

trigger:='0';
```

```
      for i in 0 to 63 loop

            chains_count:= 0;

            for j in 0 to 24 loop

                  chains_count:=
                  chains_count + conv_integer(chains(i)(j));

            end loop;

            if (chains_count > majority) then
                  trigger:='1';
                  exit;
            end if;

      end loop;

      trig <= trigger;

end process;

end architecture;
```

Nella simulazione su iSim del modulo *trig_gen* rappresentata in Fig. 28 è possibile evidenziare il diverso comportamento del segnale di trigger di PDM al variare del parametro di configurazione `trig_sel`.

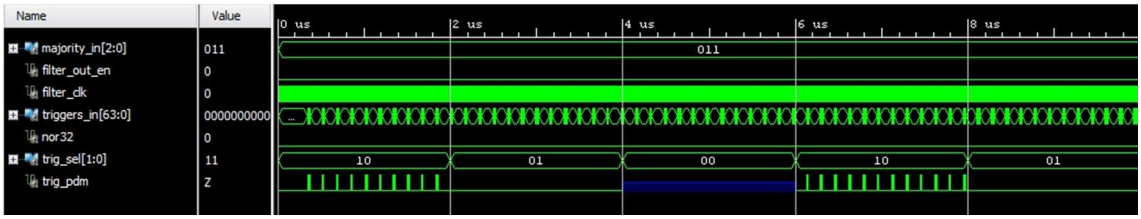


Fig. 28. Simulazione su iSim degli impulsi di trigger del modulo trig_gen al variare del parametro di configurazione "trig_sel".

Successivamente alla versione del modulo *trig_gen* discussa in questa sezione, sono state effettuate ulteriori modifiche funzionali allo scopo di ottimizzare l'algoritmo di generazione delle adiacenze, rendere attivabile la sezione di *trigger_filtering* per il filtraggio dei *glitch* provocati dall'algoritmo, allineare il numero richiesto di adiacenze logiche al parametro di configurazione `majority_in`, ed eliminare infine i *glitch* spuri in uscita al modulo sul segnale di trigger di PDM.

L'attivazione della sezione di filtraggio del modulo *trig_gen* avviene attraverso la costante TRIG_FILTER_EN, inizialmente settata a "0" come valore di *default*. Modificando ad "1" il valore di tale costante nella seguente riga:

```
constant TRIG_FILTER_EN : std_logic := '1';
```

è possibile abilitare l'attivazione del filtro per i *glitch* tramite il parametro di configurazione `filter_out_en`, come illustrato nello schema a blocchi della Fig. 27.

L'allineamento del numero richiesto di adiacenze logiche al parametro di configurazione `majority_in` è stato effettuato modificando la seguente condizione logica nella sezione del codice relativa all'attivazione del segnale `trigger`:

```
if (chains_count >= majority) then
      trigger:='1';
      exit;
end if;
```

Precedentemente a tale modifica, il trigger di PDM veniva generato se il numero di pixel adiacenti che scattano è superiore al valore decimale impostato dal parametro di configurazione `majority_in` addizionato di una unità (ad esempio, se "majority_in"=3, il trigger di PDM si porta a "1" se vi sono almeno 4 pixel adiacenti). In seguito alla suddetta modifica, il segnale di trigger relativo all'algoritmo scatta allorquando il numero di trigger nei pixel adiacenti eguaglia esattamente il valore fornito in ingresso al modulo dal parametro di configurazione `majority_in` (ad esempio, se "majority_in"=3, il trigger di PDM si porta a "1" se vi sono almeno 3 pixel adiacenti).

A tal proposito, vale la pena rimarcare che, in base a come è strutturato l'algoritmo di scansione delle adiacenze (effettuato all'interno di sottomatrici di larghezza 5×5), assegnando al parametro `majority_in` a 3 bit i valori più significativi da 6 a 7 (che equivarrebbe ad attivare il trigger di PDM in presenza di trigger su 6-7 pixel adiacenti), allora il segnale di trigger in uscita all'algoritmo potrebbe non portarsi ad "1" qualora i 6-7 pixel adiacenti della PDM non dovessero trovarsi all'interno di una sottomatrice 5×5. Nella Fig. 29 sono rappresentati due possibili eventualità nelle quali non si produrrebbe in uscita un segnale di trigger pur verificandosi un numero di adiacenze tra pixel attivi pari al valore settato dal parametro di configurazione `majority_in`.

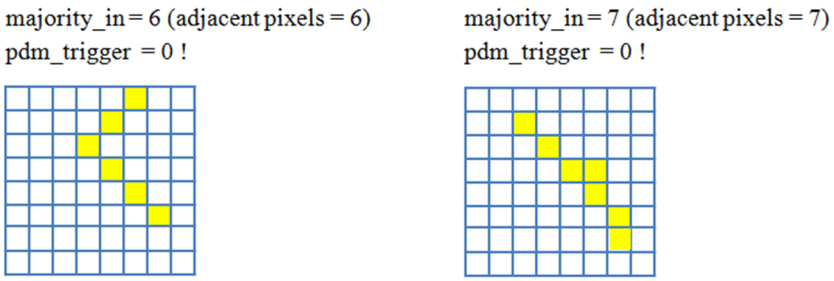


Fig. 29. Rappresentazione di due casistiche di adiacenze tra pixel attivi (6 sulla sinistra e 7 sulla destra) che non produrrebbero alcun trigger di PDM a causa del fatto che le adiacenze richieste non si verificano all'interno di una sottomatrice di larghezza 5×5.

Inoltre, per completezza di informazioni, se al parametro `majority_in` si dovesse assegnare il valore binario corrispondente allo "0" decimale (eventualità possibile, sebbene non plausibile), il che implicherebbe una condizione di adiacenze tra pixel pari a "0", allora il trigger in uscita resterebbe ancorato ad "1" (come verificato in simulazione).

Per quanto concerne le modifiche legate all'ottimizzazione dell'algoritmo, descriviamo brevemente la metodologia di riconoscimento delle adiacenze tra pixel, per come è stata concepita la modalità di generazione del trigger, da cui verranno successivamente derivate le considerazioni che hanno portato ad una ottimizzazione dell'architettura del modulo *trig_gen*.

L'algoritmo di generazione del trigger implementa uno specifico criterio di ricerca per il riconoscimento delle adiacenze spaziali tra pixel. Più specificamente, gli 8×8 bit, associati ai 64 pixel di PDM, che rappresentano le 64 occorrenze di trigger (un "1" logico in presenza di trigger, ed uno "0" logico in assenza di trigger) sono simmetricamente disposti all'interno di una struttura matriciale di larghezza 12×12 come illustrato in Figura 16 (a sinistra), riempiendo le posizioni al di fuori dei 64 pixel di PDM con i valori "0". Questa matrice più grande è suddivisa in settori di larghezza 5×5 sovrapposti tra loro con passo di 1 pixel, in modo tale che ciascuna casella relativa ai 64 pixel di PDM costituisca il pixel centrale del relativo settore. Per un dato settore 5×5, la ricerca di pixel attivi in posizioni attigue è ottenuta controllando pixel per pixel l'eventuale adiacenza di caselle attigue rispetto al pixel centrale e, di volta in volta, le successive eventuali adiacenze rispetto a ciascuna delle 8 celle contigue al pixel centrale, come schematizzato in Figura 16 (a destra) per un singolo settore 5×5. Il vantaggio di questa procedura di scansione delle adiacenze tra pixel risiede nel fatto che esso si presta ad essere facilmente descritto in HW mediante una serie di sequenze cicliche (implementate in codice VHDL dai tipici costrutti `for`/`loop`).

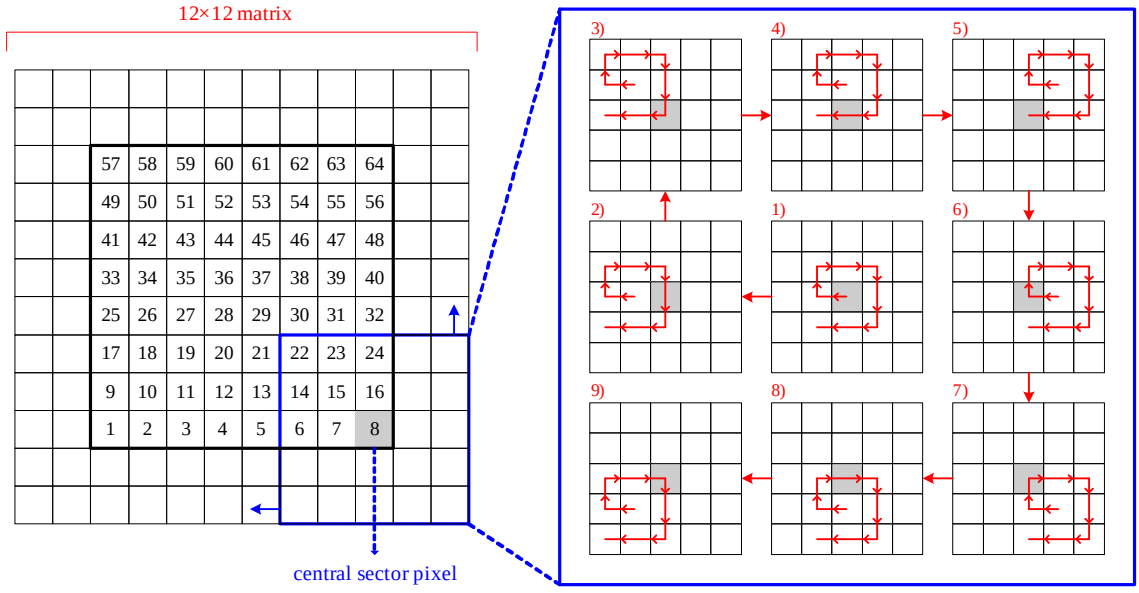


Fig. 30. Rappresentazione schematica della tecnica di scansione adoperata per il controllo delle adiacenze tra pixel in un singolo settore 5×5.

I risultati di ogni potenziale adiacenza fra trigger in ogni settore 5×5 rispetto al suo pixel centrale vengono successivamente memorizzati in 64 buffer di 5×5 elementi, uno per ogni settore, in cui le singole celle conterranno un valore logico "1" ogni qualvolta si ve-

rifichi un percorso di adiacenze logico rispetto al relativo pixel centrale. Viceversa, se il pixel centrale di un dato settore presenta un valore logico "0", il relativo buffer non presenterà alcun "1" logico in nessun elemento. In tal modo, viene rilevata l'adiacenza tra pixel nell'intera matrice 8×8 della PDM, a partire dal pixel centrale di un dato settore ed indipendentemente dal numero di pixel adiacenti attivi, a condizione che i *pattern* di pixel contigui che scattano siano circoscritti nell'ambito di un settore 5×5.

Contemporaneamente ai processi di scansione dei pixel e di scrittura nei relativi buffer, 64 contatori, uno per ogni settore 5×5, provvedono al conteggio del numero progressivo di valori logici "1" in un dato buffer di settore, per stabilire se viene verificata una determinata coincidenza di pixel adiacenti. A tale scopo, viene usato il segnale `majority`, collegato al relativo parametro di configurazione a 3-bit, in modo tale che un trigger di PDM venga generato quando il numero di valori logici "1" in ogni buffer di settore raggiunge il numero di adiacenze desiderato.

Il seguente stralcio di codice VHDL descrive le istruzioni, incluse in un doppio ciclo `for` all'interno del `process` principale dell'algoritmo, che vengono utilizzate sia per generare i cicli di scansione dei 25 pixel in uno qualsiasi dei 64 settori, che per scrivere i relativi dati di adiacenza nei corrispondenti buffer di settore:

```
for j in 0 to 8 loop

      for t in 0 to 8 loop

      chain_offset:=
      12 + chains_offsets(j) + chains_offsets(t);

      if ((
      pixels(pixel_offset) and
      pixels(pixel_offset + pixels_offsets(j)) and
      pixels(pixel_offset + pixels_offsets(j) + pixels_offsets(t)))
      = '1') then
            chains(pixel_chain)(chain_offset):='1';
      end if;

      end loop;

end loop;
```

in cui `pixels` è la matrice 12×12 dei 64 pixel di PDM, `chains` è l'elemento di memoria 5×5, `pixel_offset` e `12` rappresentano rispettivamente i pixel centrali del dato settore 5×5 e del relativo buffer di memoria, `chain_offset` indica la posizione del pixel all'interno del buffer di settore, mentre `pixels_offsets` e `chains_offsets` sono rispettivamente gli indici di vettore del settore matriciale e del relativo buffer, le cui variazioni dinamiche all'interno del ciclo permettono di eseguire la scansione di un intero settore come in Figura 16, alla ricerca di trigger adiacenti rispetto al pixel centrale.

La realizzazione fisica della suddetta procedura di scansione comporta la creazione da parte del sintetizzatore HW di VIVADO di 81 blocchi logici di tipo AND per ogni settore di PDM da 5×5, come schematicamente rappresentato in Fig. 31.

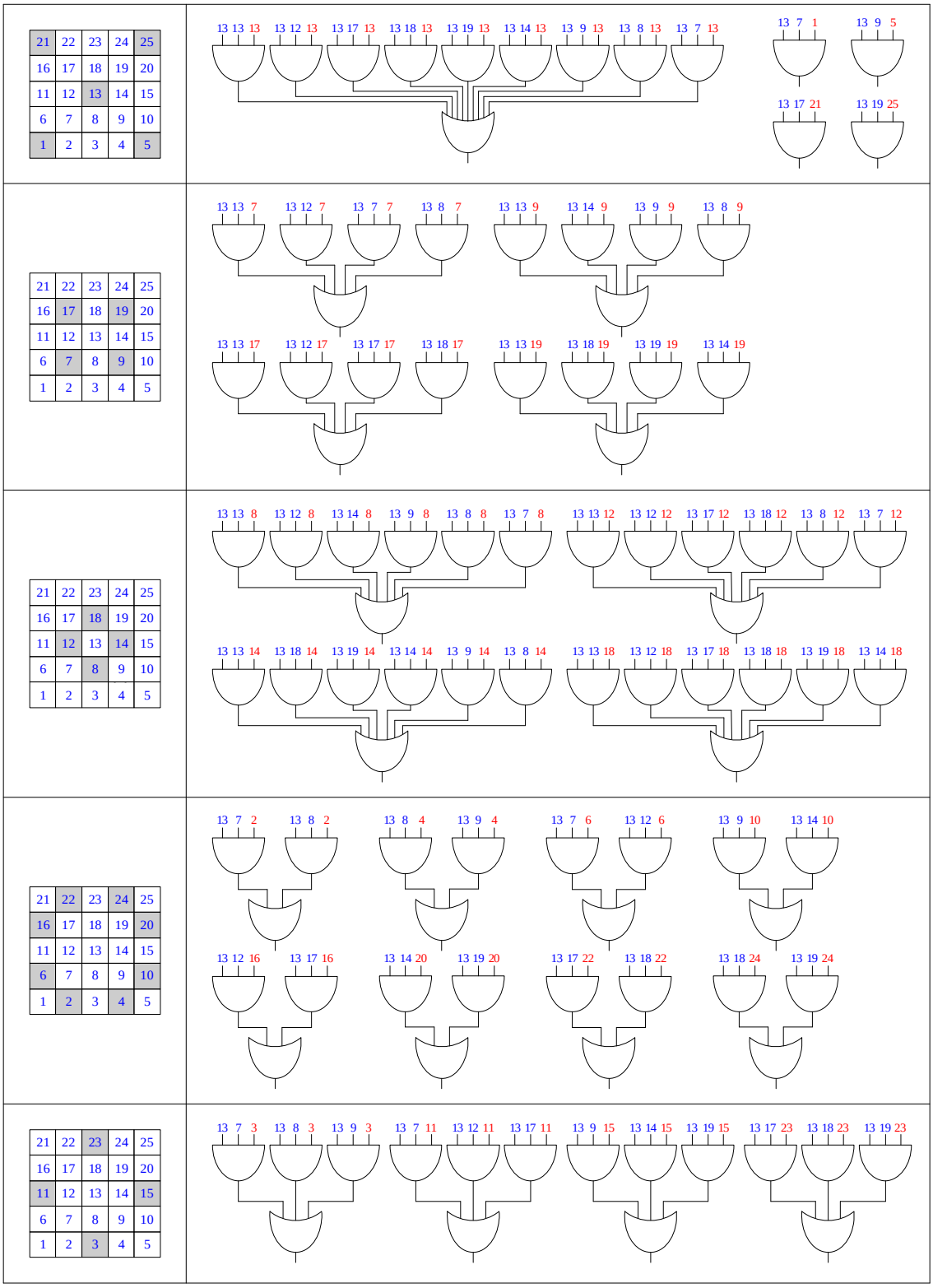


Fig. 31. Sviluppo HW della procedura di scansione a doppio ciclo per un settore di PDM da 5×5.

Infatti, ogni cella di settore scandita attraverso il doppio ciclo `for` viene implementata in HW da una porta logica AND a 3 ingressi, e la scrittura dei dati su ciascuna delle 25 locazioni del relativo buffer viene eseguita attraverso l'OR logico dei segnali di tutte le possibili combinazioni di ingressi delle porte AND relative a quella cella (che, di fatto, rappresentano tutte le possibili adiacenze logiche), a seconda della posizione specifica nel buffer di settore. In particolare, nella Fig. 31, i segnali rossi all'ingresso delle porte AND a destra rappresentano i corrispondenti pixel scanditi in grigio nel settore 5×5 a sinistra; le 21 uscite delle porte OR, unitamente alle uscite delle 4 AND logiche collegate ai pixel 1, 5, 21 e 25, rappresentano i dati memorizzati nel buffer di settore.

La logica rappresentata in figura viene dunque naturalmente ripetuta per ciascuno degli 81 settori da 5×5 necessari a ricoprire i 64 pixel di una intera PDM.

Ora, nonostante la gran praticità della tecnica di scansione precedentemente discussa, una certa ridondanza logica viene inevitabilmente generata nell'implementazione HW, a causa del fatto che, durante la scansione delle adiacenze, la maggior parte dei 25 pixel di settore vengono controllati più di una volta all'interno dello stesso *loop*, implicando pertanto una superflua occupazione delle risorse allocate. Ciò, in definitiva, potrebbe portare ad un aumento dei *glitch* spuri in uscita all'algoritmo e ad un possibile ritardo nella identificazione delle adiacenze.

Alla luce delle considerazioni di cui sopra, e in base alle configurazioni logiche illustrate in Fig. 31, può essere derivata una struttura più compatta ed efficiente, che eviti la ridondanza nell'utilizzo di logica combinatoria allo scopo di migliorare l'efficienza del sistema. In particolare, ogni singolo pixel dell'*i*-esimo settore di PDM da 5×5 può essere mappato direttamente nella posizione relativa dell'*i*-esimo buffer di settore secondo la medesima configurazione evidenziata in Fig. 31, ma con il vantaggio di eseguire una notevole riduzione della logica combinatoria associata. Pertanto, i cicli di scansione per determinare le adiacenze tra pixel in un dato settore PDM possono essere sostituiti da 25 istruzioni di assegnazione logica del tutto equivalenti (in considerazione delle possibili adiacenze ottenibili in ognuna delle 25 celle di settore), ottenendo così gli stessi risultati utilizzando un'architettura logica più compatta.

La Fig. 32 illustra, per un determinato settore di PDM, la logica semplificata per la scrittura dei trigger adiacenti nel buffer di settore pertinente. Confrontando le due figure, è possibile osservare la riduzione nel numero di porte logiche impiegate per ciascun pixel relativo ad un settore di PDM: in Fig. 31 sono necessari 81 blocchi logici, uno per pixel, contro i 25 richiesti dall'architettura semplificata in Fig. 32. Anche in questo caso, i segnali di ingresso in blu si riferiscono alle 25 caselle del settore da 5×5, mentre i segnali di uscita in rosso corrispondono ai 25 dati digitali memorizzati nel relativo buffer.

In virtù della seguente ottimizzazione, una stessa struttura logica di base viene utilizzata per ciascun pixel di settore. Così, i pixel evidenziati nel primo pannello verticale della Fig. 32 sono sempre contigui al pixel centrale e vengono semplicemente mappati tramite una porta AND a due ingressi, essendo il pixel centrale uno dei due ingressi della porta; il pixel 3 evidenziato in rosso nel secondo pannello risulta essere contiguo al pixel centrale attraverso i pixel 7, 8 e 9, ed è quindi mappato da una porta AND a tre ingressi, dove i primi due ingressi sono i pixel 3 e 13, mentre il terzo ingresso è l'uscita di una porta OR tra pixel 7, 8 e 9; analogamente, il pixel 1 evidenziato in rosso nell'ultimo pannello è contiguo al pixel centrale soltanto attraverso il pixel 7, e viene quindi map-

pato da una porta AND a tre ingressi (pixel 1, 7 e 13); infine, il dato logico nel pixel 13 è direttamente memorizzato nella locazione centrale del relativo buffer di settore.

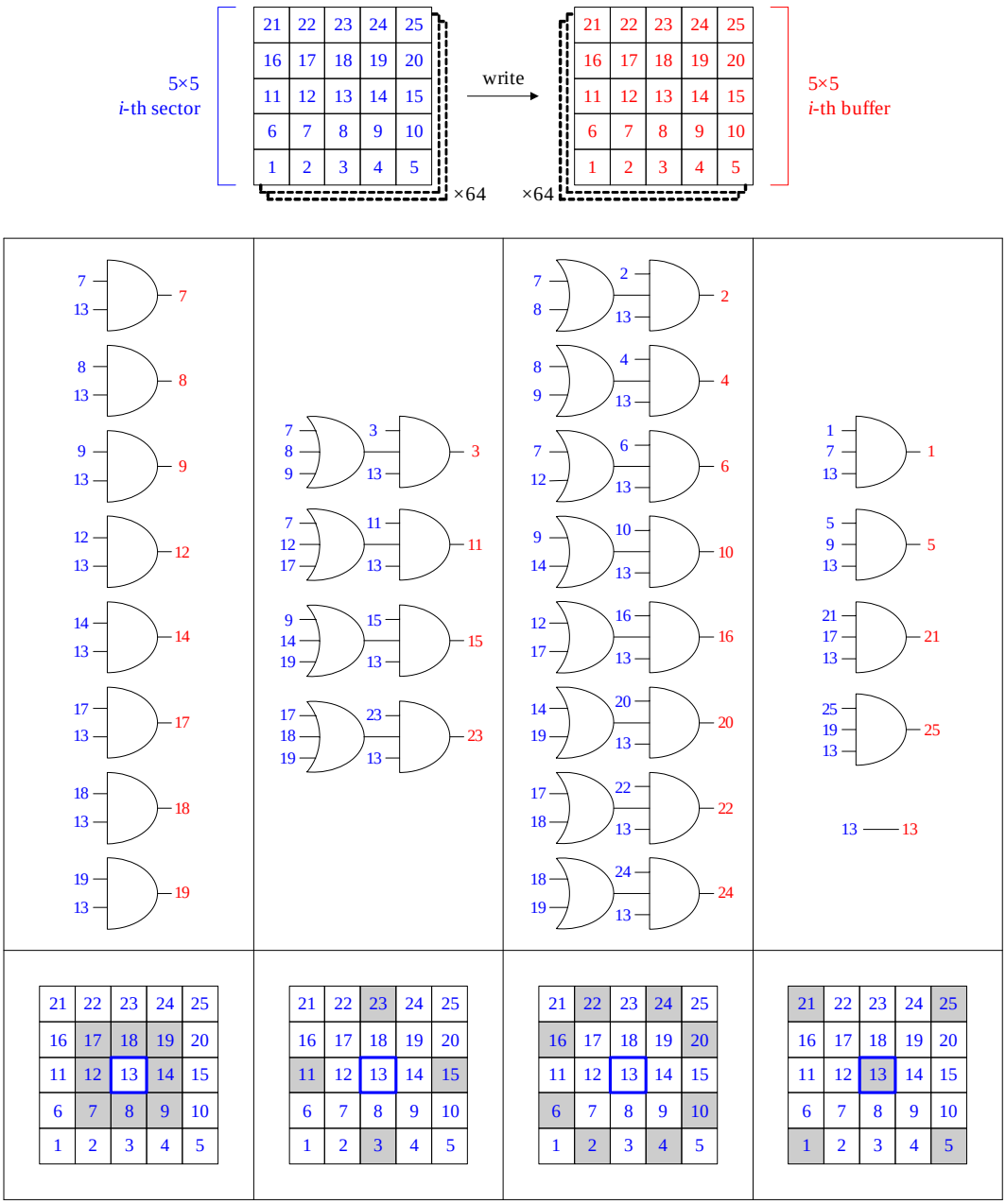


Fig. 32. Sviluppo HW ottimizzato dell'algoritmo di trigger per un settore di PDM da 5×5.

In aggiunta a tale ottimizzazione, che ha comportato la sostituzione del doppio ciclo interno del codice VHDL con 25 semplici assegnazioni logiche, una per pixel (riducendo l'occupazione logica di un singolo settore da 5×5 di un fattore 81:25), è stata prevista

anche la registrazione dei trigger in ingresso per rendere sincrona l'individuazione dei trigger in ingresso al modulo con conseguente eliminazione dei *glitch*. Il campionamento dei trigger in ingresso è stato realizzato tramite l'introduzione di 64 flip-flop di tipo D con l'aggiunta delle seguenti righe di codice nella sezione dell'algoritmo:

```
-- REGISTRAZIONE TRIGGER IN INGRESSO
signal triggers_in_reg  : std_logic_vector(63 downto 0);

-- REGISTRAZIONE TRIGGER IN INGRESSO
triggers_in_reg <= triggers_in when
                   (filter_clk'event and filter_clk='1');

for i in 0 to 7 loop

      for k in 0 to 7 loop

            ch_sorter_ptr_i:= (i * 8) + k;
            ch_sorter_ptr_o:= (i * 12) + k + 26;
            pixels(ch_sorter_ptr_o):=
             triggers_in_reg(ch_sorter_buf(ch_sorter_ptr_i));
            -- TRIGGER REGISTRATI

      end loop;

end loop;
```

In Fig. 33 sono riportati i risultati delle simulazioni relative al modulo di generazione del trigger di PDM nella versione originale (pannello superiore) e nella versione ottimizzata con registrazione dei trigger in ingresso (pannello inferiore). Come si evince dai grafici, i *glitch* in uscita vengono praticamente rimossi nella versione ottimizzata del modulo, mentre sono presenti in abbondanza nella versione originale, nonostante l'attivazione della sezione di *glitch filterning*.

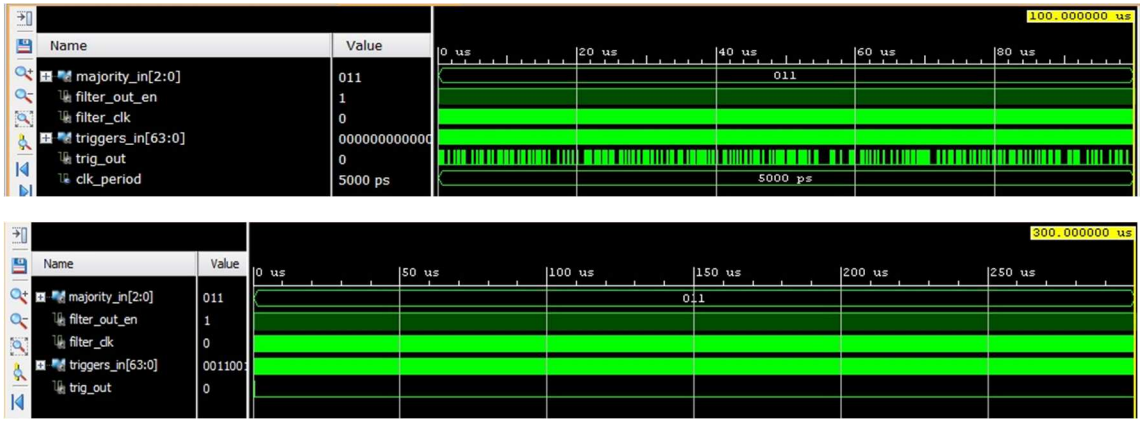


Fig. 33. Simulazioni su iSim del trigger di PDM. La versione originale dell'algoritmo presenta notevoli impulsi di glitch (pannello superiore); la versione ottimizzata dell'algoritmo con registrazione dei trigger rimuove completamente i glitch (pannello inferiore).

Allo scopo di verificare direttamente su scheda FPGA l'assenza di *glitch* in uscita al modulo *trig_gen*, è stata implementata una modifica allo schema a blocchi dell'FPGA, per consentire un conteggio diretto dei trigger di PDM nella modalità di generazione del trigger (ovvero settando ad "10" il parametro di configurazione "trig_sel"). In particolare, una delle 64 linee di trigger collegate al modulo *trig_count* è stata rimpiazzata dal segnale di trigger di PDM. In Fig. 34 sono riportati i risultati dei test eseguiti.

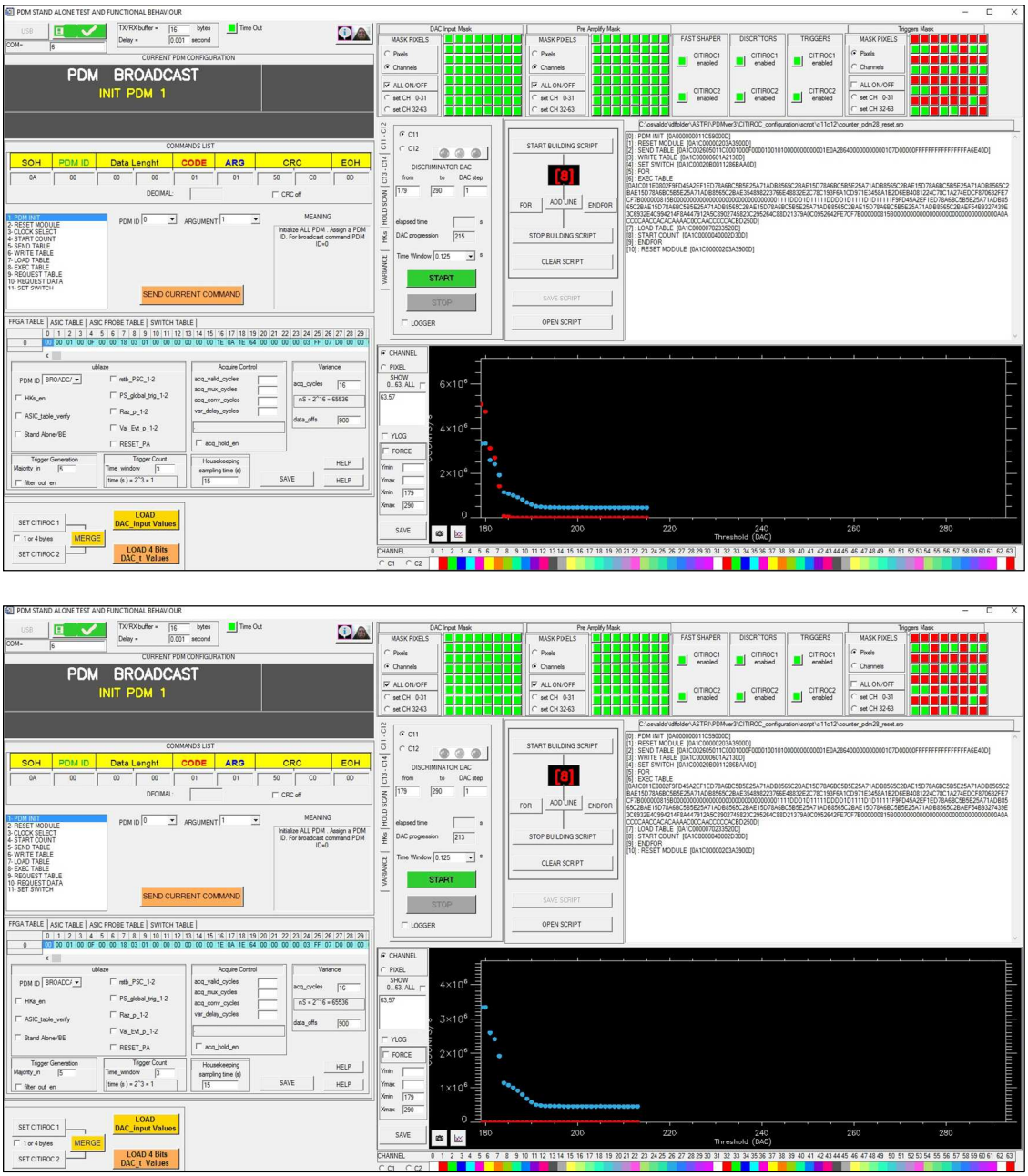


Fig. 34. Verifica su scheda FPGA della rimozione di glitch nella versione ottimizzata del modulo trig_gen con registrazione dei trigger in ingresso (sotto) rispetto alla versione originale (sopra).

In particolare, è stata settata una “majority” pari a 3 ed è stata abilitata una configurazione di pixel attivi dove non viene mai verificata un’adiacenza fisica fra 3 pixel. In queste condizioni, la presenza di eventuali segnali spuri in uscita all’algoritmo sarebbero da imputarsi ai *glitch* generati all’interno del modulo *trig_gen*. In entrambi i grafici della figura, i punti rossi rappresentano i *glitch* contati e quelli azzurri i conteggi relativi ad un SiPM Hamamatsu utilizzato per altre misure. Le adiacenze sono verificate sul rumore dei canali (osservabili in entrambi i grafici soltanto a soglie basse). Come si evince dallo *screenshot* nel pannello inferiore, nella versione ottimizzata del modulo con registrazione dei trigger in ingresso non sono presenti impulsi spuri in uscita (i punti rossi indicano sempre un conteggio nullo).

Il codice VHDL relativo alla sezione dell’algoritmo ottimizzato con inclusa la registrazione dei trigger in ingresso per l’eliminazione dei *glitch* è riportato nelle seguenti righe, dove sono state commentate le sezioni eliminate dalla versione originale:

```
-- OPTIMIZED TRIGGER ALGORITHM ************************************

process (triggers_in, majority)

type chains_type is array(integer range <>)
                    of std_logic_vector(24 downto 0);
--type pixels_type is array(integer range <>)
                      of integer range -13 to 13;
--type results_type is array(integer range <>)
                      of integer range -6 to 6;

variable chains           : chains_type(0 to 63);
--variable chains_offsets : results_type(0 to 8);
--variable pixels_offsets : pixels_type(0 to 8);

variable pixel_chain   : integer range 0 to 63;
variable pixel_offset  : integer range 0 to 143;
variable chains_count  : integer range 0 to 25;
--variable chain_offset : integer range 0 to 24;

variable ch_sorter_ptr_i : integer range 0 to 63;
variable ch_sorter_ptr_o : integer range 26 to 117;

variable pixels  : std_logic_vector(143 downto 0);  -- MATRICE 12 x 12
variable trigger : std_logic;

begin

      pixels:= (others=>'0');

      for i in 0 to 7 loop

            for k in 0 to 7 loop

                  ch_sorter_ptr_i:= (i * 8) + k;
                  ch_sorter_ptr_o:= (i * 12) + k + 26;
```

```
                  pixels(ch_sorter_ptr_o):=
                  triggers_in_reg(ch_sorter_buf(ch_sorter_ptr_i));
                  -- TRIGGER REGISTRATI

            end loop;

      end loop;

--    pixels_offsets(0):= 0;
--    pixels_offsets(1):= 1;
--    pixels_offsets(2):= 13;
--    pixels_offsets(3):= 12;
--    pixels_offsets(4):= 11;
--    pixels_offsets(5):= -1;
--    pixels_offsets(6):= -13;
--    pixels_offsets(7):= -12;
--    pixels_offsets(8):= -11;

--    chains_offsets(0):= 0;
--    chains_offsets(1):= 1;
--    chains_offsets(2):= 6;
--    chains_offsets(3):= 5;
--    chains_offsets(4):= 4;
--    chains_offsets(5):= -1;
--    chains_offsets(6):= -6;
--    chains_offsets(7):= -5;
--    chains_offsets(8):= -4;

      for i in 0 to 7 loop

            for k in 0 to 7 loop

                  pixel_chain:= (i * 8) + k;
                  pixel_offset:= (i * 12) + k + 26;

chains(pixel_chain)(12):= pixels(pixel_offset);

chains(pixel_chain)(6) := pixels(pixel_offset - 13) and
                          pixels(pixel_offset);
chains(pixel_chain)(7) := pixels(pixel_offset - 12) and
                          pixels(pixel_offset);
chains(pixel_chain)(8) := pixels(pixel_offset - 11) and
                          pixels(pixel_offset);
chains(pixel_chain)(11):= pixels(pixel_offset - 1)  and
                          pixels(pixel_offset);
chains(pixel_chain)(13):= pixels(pixel_offset + 1)  and
                          pixels(pixel_offset);
chains(pixel_chain)(16):= pixels(pixel_offset + 11) and
                          pixels(pixel_offset);
chains(pixel_chain)(17):= pixels(pixel_offset + 12) and
                          pixels(pixel_offset);
chains(pixel_chain)(18):= pixels(pixel_offset + 13) and
                          pixels(pixel_offset);
```

```
chains(pixel_chain)(1) := pixels(pixel_offset - 25) and
      (pixels(pixel_offset - 13) or pixels(pixel_offset - 12)) and
       pixels(pixel_offset);
chains(pixel_chain)(3) := pixels(pixel_offset - 23) and
      (pixels(pixel_offset - 12) or pixels(pixel_offset - 11)) and
       pixels(pixel_offset);
chains(pixel_chain)(5) := pixels(pixel_offset - 14) and
      (pixels(pixel_offset - 13) or pixels(pixel_offset - 1))  and
       pixels(pixel_offset);
chains(pixel_chain)(9) := pixels(pixel_offset - 10) and
      (pixels(pixel_offset - 11) or pixels(pixel_offset + 1))  and
       pixels(pixel_offset);
chains(pixel_chain)(15):= pixels(pixel_offset + 10) and
      (pixels(pixel_offset + 11) or pixels(pixel_offset - 1))  and
       pixels(pixel_offset);
chains(pixel_chain)(19):= pixels(pixel_offset + 14) and
      (pixels(pixel_offset + 13) or pixels(pixel_offset + 1))  and
       pixels(pixel_offset);
chains(pixel_chain)(21):= pixels(pixel_offset + 23) and
      (pixels(pixel_offset + 11) or pixels(pixel_offset + 12)) and
       pixels(pixel_offset);
chains(pixel_chain)(23):= pixels(pixel_offset + 25) and
      (pixels(pixel_offset + 12) or pixels(pixel_offset + 13)) and
       pixels(pixel_offset);

chains(pixel_chain)(2) := pixels(pixel_offset - 24) and
      (pixels(pixel_offset - 13) or pixels(pixel_offset - 12)
    or pixels(pixel_offset - 11)) and pixels(pixel_offset);
chains(pixel_chain)(10):= pixels(pixel_offset - 2)  and
      (pixels(pixel_offset - 13) or pixels(pixel_offset - 1)
    or pixels(pixel_offset + 11)) and pixels(pixel_offset);
chains(pixel_chain)(14):= pixels(pixel_offset + 2)  and
      (pixels(pixel_offset - 11) or pixels(pixel_offset + 1)
    or pixels(pixel_offset + 13)) and pixels(pixel_offset);
chains(pixel_chain)(22):= pixels(pixel_offset + 24) and
      (pixels(pixel_offset + 11) or pixels(pixel_offset + 12)
    or pixels(pixel_offset + 13)) and pixels(pixel_offset);

chains(pixel_chain)(0) := pixels(pixel_offset - 26) and
       pixels(pixel_offset - 13) and pixels(pixel_offset);
chains(pixel_chain)(4) := pixels(pixel_offset - 22) and
       pixels(pixel_offset - 11) and pixels(pixel_offset);
chains(pixel_chain)(20):= pixels(pixel_offset + 22) and
       pixels(pixel_offset + 11) and pixels(pixel_offset);
chains(pixel_chain)(24):= pixels(pixel_offset + 26) and
       pixels(pixel_offset + 13) and pixels(pixel_offset);

--                chains(pixel_chain):= (others=>'0');

--                for j in 0 to 8 loop

--                      for t in 0 to 8 loop
```

```
--                      chain_offset:=
--                      12 + chains_offsets(j) + chains_offsets(t);

--    if ((
--    pixels(pixel_offset) and
--    pixels(pixel_offset + pixels_offsets(j)) and
--    pixels(pixel_offset + pixels_offsets(j) + pixels_offsets(t)))
--    = '1') then
--           chains(pixel_chain)(chain_offset):='1';
--    end if;

--                      end loop;

--                end loop;

            end loop;

      end loop;

      trigger:='0';

      for i in 0 to 63 loop

            chains_count:= 0;

            for j in 0 to 24 loop

                  chains_count:=
                  chains_count + conv_integer(chains(i)(j));

            end loop;

            if (chains_count >= majority) then
                  trigger:='1';
                  exit;
            end if;

      end loop;

      trig <= trigger;

end process;
```

Il diagramma a blocchi relativo alla nuova versione del modulo *trig_gen*, che comprende l'ottimizzazione dell'algoritmo di generazione e la registrazione dei trigger in ingresso, è rappresentato nella Fig. 35.

Vale la pena rimarcare in questa sede che il segnale di ingresso "nor32", inizialmente collegato alla relativa uscita degli ASIC, rappresenta in questa nuova versione la OR logica delle 64 linee di trigger in uscita al modulo integrativo *monostable_mask*, implementato all'interno del diagramma a blocchi dell'FPGA ed analizzato in dettaglio nella relativa sezione di questo report.

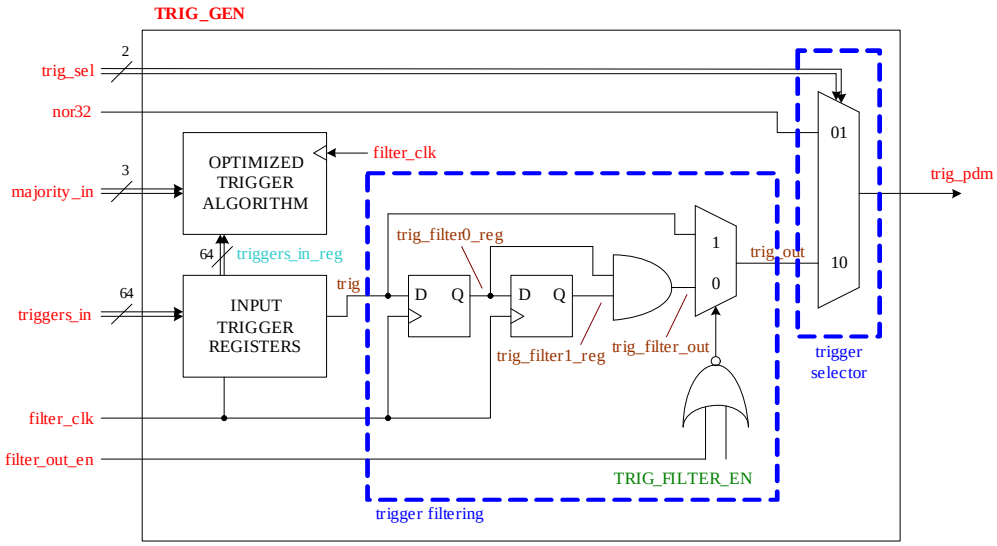


Fig. 35. Diagramma a blocchi aggiornato relativo al modulo di generazione del trigger di PDM.

La seguente tabella riassume il funzionamento dei segnali di input/output relativi alla versione aggiornata dell'entità *trig_gen*.

SIGNAL	DIRECTION	BITS	CONNECTION	DESCRIPTION
triggers_in	in (active H)	64	from CITIROC	outputs of the 64 discriminator signals from ASICs
nor32	in (OC)	1	from CITIROC	nor32 (output of the 64 triggers from monostable multivibrators)
majority_in	in	3	from uBlaze	trig_pdm is high when number of adj. pixels ≥ majority_in (0-7)
trig_sel	in	2	from uBlaze	trigger selector: 10=trigger; 01=nor32; 00/11=high Z
filter_clk	in	1	from uBlaze	clock for filter section (simulated at 200MHz)
filter_out_en	in (active H)	1	from uBlaze	signal enabling trigger filtering (2 clock periods delay)
trig_pdm	out (active H)	1	to BEE FPGA	PDM trigger output signal

4.6 Realizzazione di un blocco HW di mascheratura dei trigger ()

Allo scopo di agevolare i test sui moduli di generazione e di conteggio dei trigger, è stato realizzato un blocco HW aggiuntivo per consentire una mascheratura logica dei 64 segnali di trigger. Nel diagramma a blocchi dell'FPGA, come rappresentato in Fig. 36, questo nuovo modulo è stato frapposto tra i 64 trigger provenienti dall'ASIC e i blocchi di conteggio e di generazione dei trigger, e richiede in ingresso un segnale di configurazione a 64 bit proveniente dal registro MicroBlaze.

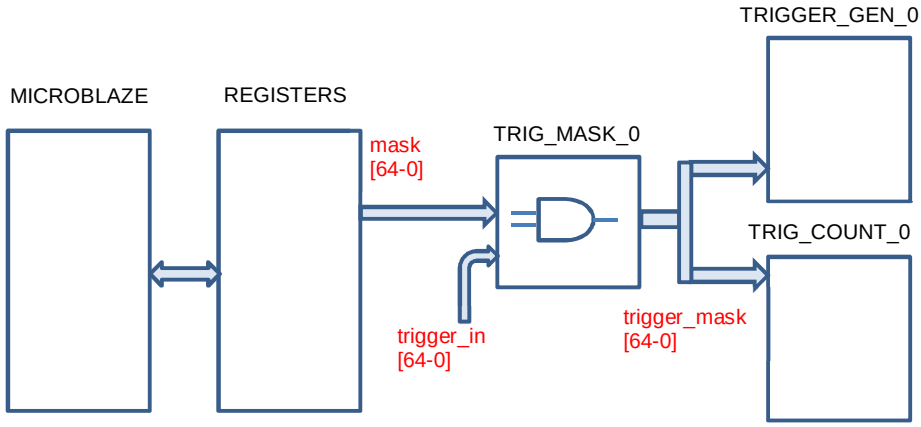


Fig. 36. Schematizzazione dei collegamenti inerenti il blocco di mascheratura dei trigger.

Il nuovo blocco *trig_mask* è costituito dal prodotto logico (AND) tra i corrispondenti bit dei segnali *trigger_mask* e *trigger_in*. Attraverso questo modulo è possibile attivare o disattivare singolarmente ciascuno dei 64 trigger. In Fig. 39 è illustrato il dettaglio del diagramma a blocchi dell'FPGA con il nuovo modulo inserito.

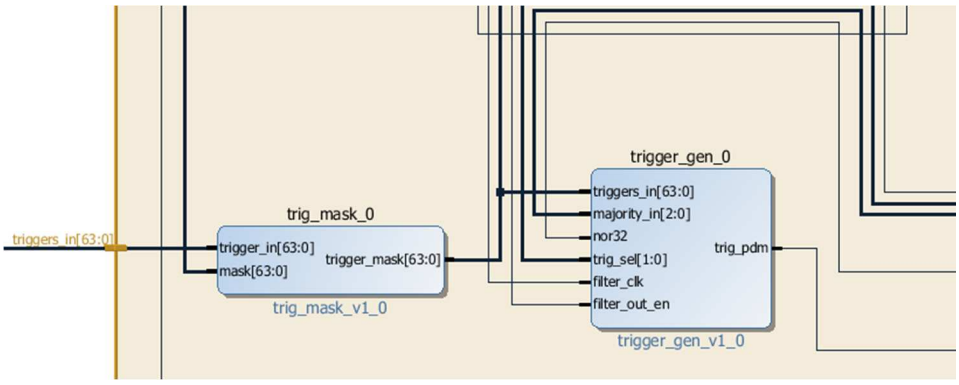


Fig. 37. Porzione del diagramma a blocchi dell'FPGA con il modulo di mascheratura dei trigger.

Nel firmware, il nuovo blocco *trig_mask* è costituito fondamentalmente dalla seguente istruzione VHDL:

```
trigger_mask <= trigger_in and mask;
```

dove `trigger_in` è il vettore dei trigger di ingresso, `trigger_mask` è il vettore delle uscite mascherate dei trigger, mentre `mask` rappresenta il segnale di configurazione a 64 bit proveniente dai registri del MicroBlaze.

Per la generazione del parametro di configurazione `trigger_mask` a 64 bit sono stati utilizzati in HW due banchi di registri liberi a 32 bit già presenti all'interno del blocco *inaf_fee_core_registers*, come illustrato in Fig. 38.

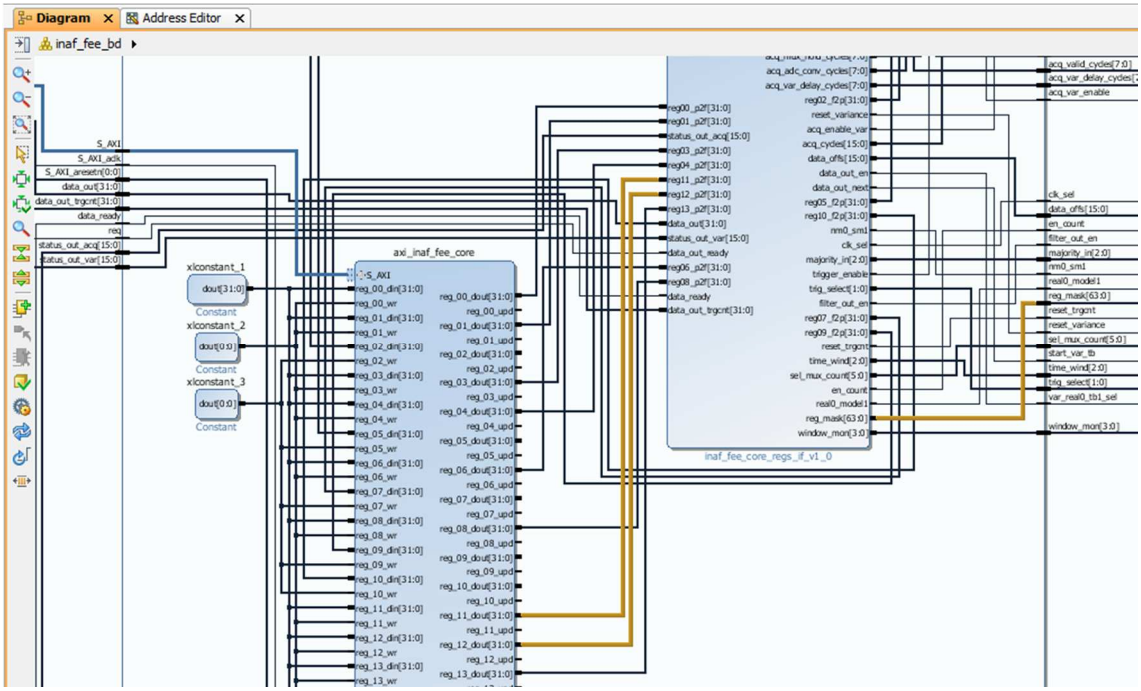


Fig. 38. Diagramma a blocchi dei registri comandati dal MicroBlaze, in cui è evidenziato il nuovo segnale di configurazione per la mascheratura dei trigger.

In particolare, nell'entità del blocco che definisce i segnali di configurazione dei registri gestiti dal processore è stata aggiunta la definizione seguente:

```
reg_mask : out std_logic_vector(63 downto 0);
```

e sono state altresì inserite nel corpo della struttura le due seguenti assegnazioni:

```
-- mask
   reg_mask(63 downto 32) <= reg11_p2f;
   reg_mask(31 downto 0)  <= reg12_p2f;
```

relativamente ai due banchi liberi di registri a 32 bit concatenati nel segnale di uscita `reg_mask` a 64 bit, che andrà dunque a configurare il modulo *trig_mask*.

Per quanto concerne il comparto SW gestito dal MicroBlaze, sono state inserite alcune istruzioni e modificate alcune *routine* in vari moduli in linguaggio C.

Anzitutto, considerato che nella tabella FPGA sono stati aggiunti altri 64 bit per il registro `mask`, si è resa necessaria una modifica alla *command line interface* (modulo SW cli.c su SDK) sul controllo della lunghezza dei bit della tabella FPGA dal valore “20” in esadecimale (32 in decimale) al valore “26” in esadecimale (38 in decimale):

```
case 3: { //read LENGTH 2/2
...
   checkLength = ((*Cli).ReadBuff[1] * 256) + (*Cli).ReadBuff[2];
      if( (checkLength == 0x0000) | (checkLength == 0x0026) |
          (checkLength == 0x011E) | (checkLength == 0x0038) |
          (checkLength == 0x0002) | (checkLength == 0x001E)) {
          int_state = 4; // modificata checkLength da 0x0020 in 0x0026
       }
...}
```

Inoltre, in considerazione del numero aggiuntivo di byte nella tabella di configurazione dell’FPGA, è stata aumentata la dimensione del relativo vettore nella struttura dichiarativa relativa alla *command line interface* (modulo SW global.h su SDK):

```
typedef struct
{
...
unsigned char FpgaTable[40]; // aumentata la dimensione da 32 byte a 40 byte
unsigned char AsicTable[286];
unsigned char ProbeAsicTable[56];
unsigned char SwitchTable[2];
...
} Cli_struct;
```

Infine, sono state introdotte due variabili di riferimento per i due banchi di registri da 32 bit nella struttura dichiarativa del codice (modulo SW global.h su SDK),utilizzate successivamente per l’assegnazione dei byte della tabella dell’FPGA:

```
typedef struct
{
...
unsigned long reg2_mask; // fpga_table_byte_30_31_32_33
unsigned long reg1_mask; // fpga_table_byte_34_35_36_37
...
} Pdm_struct;
```

dove i commenti in verde accanto alle definizioni specificano la posizione dei byte della tabella dell’FPGA che si vuole assegnare alle due variabili.

Successivamente, è stato aggiornato il codice relativo alla gestione dei bit aggiuntivi (modulo SW helloworld.c su SDK). In particolare è stata aggiunta la seguente inizializzazione relativa ai due banchi di registri da 32 bit:

```
void ClearPDMStruct(Pdm_struct *PDM){
...
// aggiunta inizializzazione
(*PDM).reg1_mask = (unsigned long) 0;
(*PDM).reg2_mask = (unsigned long) 0;
...}
```

Ora, i byte 37, 36, 35, 34, 33, 32, 31 e 30 del vettore `FpgaTable` della *command line interface* sono stati assegnati al registro delle maschere dei trigger, `reg_mask`, in modo tale che con i byte 37-34 è possibile mascherare i trigger dei canali 63-32, mentre con i byte 33-30 è possibile mascherare i trigger dei canali 31-0. A tale scopo, nel comando SEND TABLE FPGA (**`Cmd8`**) sono state aggiunte le seguenti due assegnazioni:

```
void Cmd8(Pdm_struct *PDM, Cli_struct *Cli){
...
// aggiunte 2 assegnazioni
PDM->reg1_mask = (((Cli->FpgaTable[37]<<24)&(0xFF000000))
                 +((Cli->FpgaTable[36]<<16)&(0xFF0000))
                 +((Cli->FpgaTable[35]<<8)&(0xFF00))
                  +(Cli->FpgaTable[34])); // fpga_table_byte_37_36_35_34
PDM->reg2_mask = (((Cli->FpgaTable[33]<<24)&(0xFF000000))
                 +((Cli->FpgaTable[32]<<16)&(0xFF0000))
                 +((Cli->FpgaTable[31]<<8)&(0xFF00))
                  +(Cli->FpgaTable[30])); // fpga_table_byte_37_36_35_34
...}
```

Inoltre, nel comando WRITE TABLE FPGA (**`Cmd11`**) sono state aggiunte le seguenti due istruzioni (modulo SW helloworld.c su SDK):

```
void Cmd11(Pdm_struct *PDM, Xgpio *Gpio){
...
// aggiunte 2 chiamate
Write_trigger_mask1((u32) (*PDM).reg1_mask);
Write_trigger_mask2((u32) (*PDM).reg2_mask);
...}
```

attraverso le quali è possibile scrivere sui due registri a 32 bit `reg11_p2f` e `reg12_p2f` i 64 valori dei byte 37-30 della tabella di configurazione dell'FPGA `FpgaTable` impostati dal software di gestione.

La scrittura dei valori assegnati in `PDM->reg1_mask` e `PDM->reg2_mask` avviene mediante le seguenti procedure (modulo SW inaf_fee_core_regs.c su SDK):

```
void Write_trigger_mask1(u32 Data)
{
reg32 tempReg = 0;
tempReg = Data;
   * ( volatile reg32 *) (INAF_FEE_CORE_REGS_BASEADDRESS + REG11_OFFSET)
     = tempReg;
}
void Write_trigger_mask2(u32 Data)
{
reg32 tempReg = 0;
tempReg = Data;
   * ( volatile reg32 *) (INAF_FEE_CORE_REGS_BASEADDRESS + REG12_OFFSET)
     = tempReg;
}
```

in cui i valori degli *offset* `REG11_OFFSET` e `REG12_OFFSET` individuano rispettivamente in HW i due registri `reg11_p2f` e `reg12_p2f`, sicché i byte 37-34 della tabella dell'FPGA (assegnati a `PDM->reg1_mask`) andranno a mascherare i trigger dei canali ASIC 63-32 (essendo scritti in `reg11_p2f`), mentre i byte 33-30 della tabella dell'FPGA (assegnati a `PDM->reg2_mask`) andranno a mascherare i trigger dei canali ASIC 31-0 (essendo scritti in `reg11_p2f`).

Le 2 funzioni aggiunte sono state opportunamente introdotte nella sezione dichiarativa del codice associato ai registri (modulo SW inaf_fee_core_regs.h su SDK):

```
void Write_trigger_mask1(u32 Data);
void Write_trigger_mask2(u32 Data);
```

Nel software di gestione dell'FPGA in modalità *stand alone*, la sezione relativa alla mascheratura dei trigger è rappresentata dal riquadro raffigurato in Fig. 39.

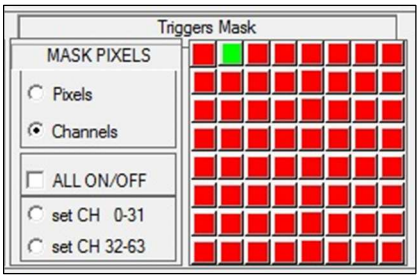


Fig. 39. Sezione del software di gestione dell'FPGA (in modalità stand alone) relativa ai settaggi per la mascheratura dei trigger.

4.7 Generazione di un segnale di OR sui trigger mascherati ()

Inizialmente, il modulo HW dedicato alla generazione del trigger di PDM, il *trig_gen*, riceveva in ingresso i 64 segnali di trigger provenienti dall'ASIC CITIROC, unitamente al segnale *nor32_t* legato alla catena non mascherabile dei segnali di trigger provenienti dal *fast-shaper* dell'ASIC. Nella Fig. 40 è rappresentata la porzione del diagramma a blocchi del CITIROC con i due segnali *open collector nor32* e *nor32_t* [R5].

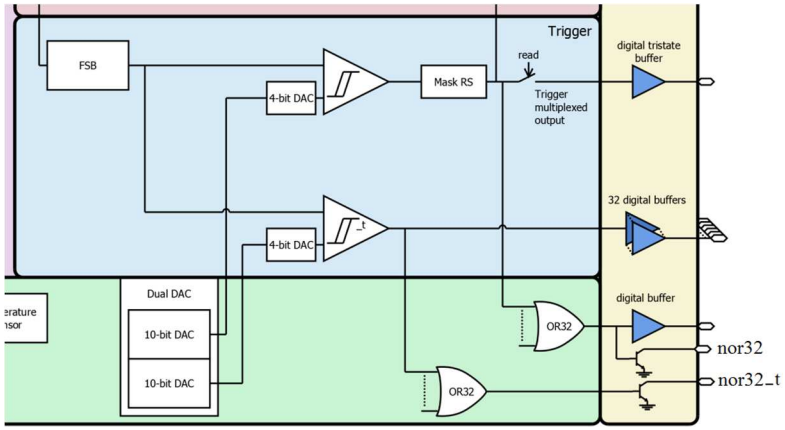


Fig. 40. Dettaglio del diagramma a blocchi del CITIROC con le uscite nor32 e nor32_t relative alle due catene di trigger.

Nella nuova versione del firmware, come illustrato in Fig. 41, il segnale *nor32_t* in ingresso al modulo *trig_gen* è stato scollegato, ed è stato posto in ingresso al blocco un segnale aggiuntivo generato dalla somma logica dei 64 segnali di trigger mascherati in uscita al modulo di mascheratura dei trigger, descritto in precedenza.

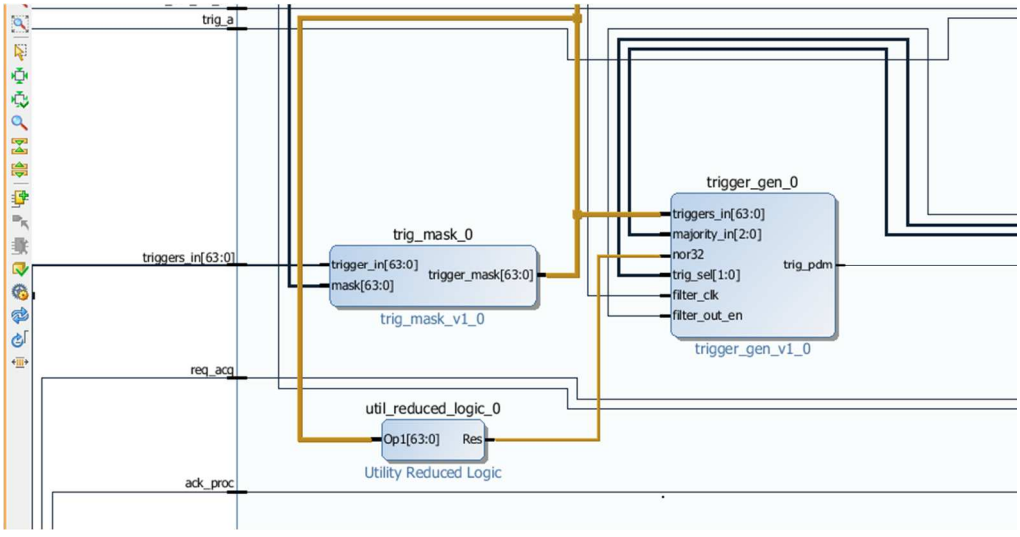


Fig. 41. Dettaglio del diagramma a blocchi dell'FPGA con il blocco logico per la generazione del segnale di OR sui trigger mascherati.

4.8 Disabilitazione della lettura di varianza in trasmissione dati ()

Dalle simulazioni iSIM sul modulo *acquire*, si è potuto constatare un altro problema legato alla trasmissione dei dati scientifici verso la BEE quando è abilitata l'acquisizione in *free-running* degli eventi di varianza, che impedisce il corretto trasferimento in uscita dei dati di evento ogni qualvolta si verifica un trigger di PDM.

Poiché le ragioni di tale anomalia risiedono nella modalità di gestione dei dati acquisiti dagli ADC in uscita ai due chip CITIROC nella scheda ASIC di *front-end*, viene qui dettagliatamente descritta la procedura di acquisizione dati da parte del modulo *acquire*.

L'abilitazione della modalità di acquisizione è subordinata all'attivazione dei seguenti segnali di configurazione da parte del microprocessore dell'FPGA:

```
acq_enable : in std_logic;
acq_var_enable : in std_logic;
```

Il primo parametro, se posto ad "1", consente l'acquisizione dei dati dagli ADC in presenza di un trigger di PDM. Il secondo parametro, se settato a "1", abilita l'acquisizione in *free-running* dei dati per la varianza.

In presenza di un trigger di PDM, il segnale "acq_start" in ingresso al modulo *acquire* commuta ad un valore logico alto. Dal fronte di salita di questo segnale, al successivo fronte di salita del clock di ingresso "acq_clk" viene posto basso il segnale di uscita "acq_nbusy", che serve ad attivare i *peak detector* di entrambi gli ASIC della scheda di *front-end*. A partire dal fronte di discesa di "acq_nbusy" e dopo un tempo proporzionale al parametro di configurazione **acq_valid_cycles** (misurato in cicli del clock "acq_clk" a 200MHz [R1]) viene posto alto il segnale "acq_hold", che attiva la lettura dei dati dagli ADC. Il parametro **acq_valid_cycles** rappresenta il tempo necessario al raggiungimento del picco dei segnali analogici delle due catene di *slow-shaper* degli ASIC [R3]. Nella Fig. 42 è raffigurata una simulazione su iSIM nella quale vengono visualizzate le evoluzioni dinamiche dei principali segnali coinvolti nella prima fase dell'acquisizione dati, in seguito alla ricezione di un trigger di PDM sul segnale "acq_start".

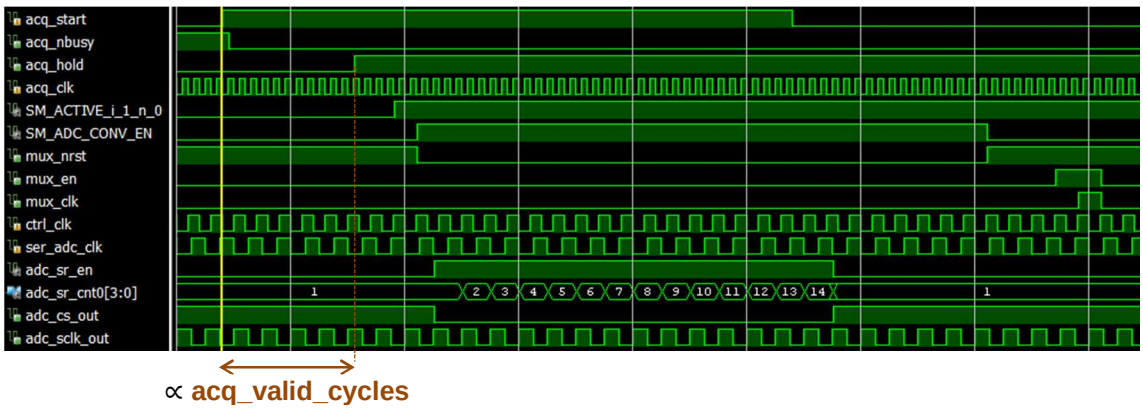


Fig. 42. Simulazione iSIM relativa all'inizio dell'acquisizione dati a seguito di un trigger di PDM.

Successivamente, dal fronte di salita del segnale “acq_hold”, dopo 3 periodi del clock di ingresso “ctrl_clk” scende il segnale “mux_nrst” (i multiplexer degli ASIC vengono posti in alta impedenza). Dal *falling edge* di “mux_nrst” scende il segnale “adc_cs_out” al successivo fronte di discesa del clock di ingresso “ser_adc_clk”. Al 14-esimo fronte di discesa di “ser_adc_clk” viene rialzato “adc_cs_out”, mentre “mux_nrst” resta a “0” per un tempo proporzionale al parametro di configurazione **acq_adc_conv_cycles** (misurato in cicli del clock “ctrl_clk” a 100MHz). Durante questa prima fase, nonostante alcuni segnali vengano dinamicamente mossi dal modulo *acquire* verso gli ADC, l’uscita di entrambi i convertitori viene mantenuta nello stato di reset, poiché i multiplexer degli ASIC non hanno ancora effettuato alcuna selezione dei canali.

Dal rising di “mux_nrst”, dopo altri 3 cicli di “ctrl_clk” viene attivato il segnale “mux_en” (lo *shifting “1”* per la selezione dei canali ADC), e dopo un altro periodo di “ctrl_clk” viene abilitata l’uscita “mux_clk, il cui periodo, d’ora in avanti, sarà legato alla frequenza di acquisizione di ciascun canale. Dal fronte di discesa di “mux_clk” e dopo un tempo proporzionale al parametro di configurazione **acq_mux_hold_cycles** (misurato in cicli del clock “ctrl_clk” a 100MHz), ovvero un tempo sufficiente a far assestare il dato da convertire, sale il segnale `SM_ADC_CONV_EN`, indicando l’inizio della fase di acquisizione nella macchina a stati del modulo. Durante il semiperiodo alto di questo segnale viene effettuata la conversione e la relativa acquisizione dei dati (14 bit) corrispondenti a ciascun singolo canale dei due CITIROC.

Dal *rising edge* di `SM_ADC_CONV_EN`, al successivo fronte di discesa di “ser_adc_clk” scende nuovamente “adc_cs_out” (segnale di *start conversion*), abilitando la conversione del primo canale (14 bit, di cui i primi 2 mantenuti a “0”, secondo quanto riportato nel datasheet dell’ADC [R4]). Sul 14-esimo fronte di discesa di **“**ser_adc_clk” risale “adc_cs_out”, indicando la fine della conversione e della relativa acquisizione del dato di un singolo canale. Il clock d’ingresso “ser_adc_clk” e il clock d’uscita “adc_sclk_out” sono sfasati di mezzo periodo tra loro; il fronte di salita di “adc_sclk_out” scandisce i singoli bit del dato da acquisire. Il segnale `SM_ADC_CONV_EN` resta alto per un tempo proporzionale al parametro di configurazione **acq_adc_conv_cycles**, un tempo sufficiente ad effettuare la conversione di 1 canale.

Dal *falling edge* di `SM_ADC_CONV_EN`, la variabile interna “v_cycle_num” comincia a contare il numero dei dati memorizzati, e sul secondo fronte di salita di “ctrl_clk” ritorna alto “mux_clk”, il cui periodo viene dunque determinato dalla combinazione dei parametri **acq_mux_hold_cycles** ed **acq_adc_conv_cycles**, come si evince chiaramente in Fig. 43. L’uscita “mux_clk” resta alta sempre per 1 periodo di “ctrl_clk”.

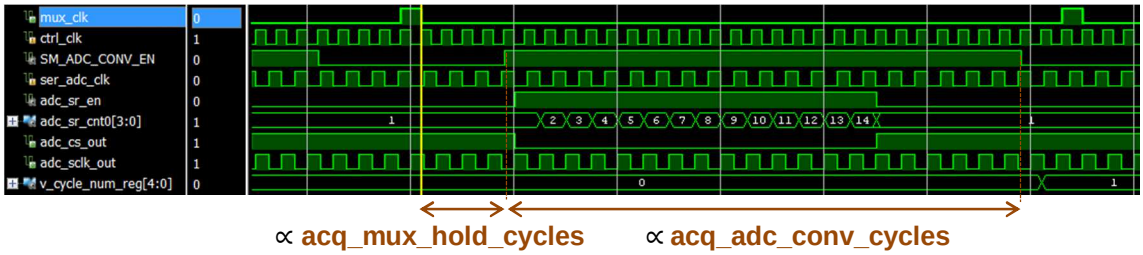


Fig. 43. Simulazione iSIM relativa al processo di acquisizione di un singolo canale (14 bit).

I tre parametri di configurazione legati al processo di acquisizione dei dati dagli ADC sono riassunti di seguito:

1) **acq_valid_cycles** (misurato in cicli del clock “acq_clk” a 300MHz, 0-255 *step*):

   rappresenta il tempo necessario al raggiungimento del picco dei segnali degli *slow-shaper* e che precede la lettura dei dati dagli ADC. Dalla ricezione di un trigger, il segnale analogico viene campionato dopo un numero di cicli di “acq_clk” pari al valore del parametro +1, in aggiunta ad un altro tempo (al massimo un altro periodo dello stesso clock), legato al campionamento del segnale “acq_start” (“acq_nbusy”). Ad esempio, assumendo un valore del parametro pari a 10 e un periodo di “acq_clk” di 3.3ns, tale ritardo complessivo risulta compreso tra 3.3ns×11 e 3.3ns×12.

2) **acq_mux_hold_cycles** (misurato in cicli del clock “ctrl_clk” a 100MHz, 0-255 *step*):

   rappresenta il tempo necessario a far assestare il dato da convertire, in seguito alla selezione del canale da parte dei *mux* degli ASIC. Dal fronte di salita del segnale “mux_clk”, che segna l’avvio del ciclo di acquisizione di un canale, fino al *rising edge* di `SM_ADC_CONV_EN`, in seguito al quale viene attivata la lettura dei dati, trascorre un tempo equivalente ad un numero di cicli di “ctrl_clk” pari al valore del parametro +3. Ad esempio, per un valore del parametro pari a 2 ed un periodo di “ctrl_clk” di 10ns, tale ritardo risulta 10ns×5.

3) **acq_adc_conv_cycles** (misurato in cicli del clock “ctrl_clk” a 100MHz, 0-255 *step*):

   rappresenta il tempo necessario ad effettuare la conversione completa di un canale (14 bit). La permanenza nello stato logico alto di `SM_ADC_CONV_EN`, che rappresenta la reale fase di acquisizione dei dati di ciascun canale degli ASIC, dura un numero di cicli di “ctrl_clk” pari al valore del parametro +5. Ad esempio, per un valore del parametro pari a 20 ed un periodo di “ctrl_clk” di 10ns, tale ritardo risulta 10ns×25.

Poiché la scansione dei singoli bit di un dato, legata al clock “ser_adc_clk” in ingresso al modulo *acquire*, può avvenire con una frequenza massima di 80MHz (in accordo alle specifiche del datasheet [R4]), mentre il tempo richiesto per la conversione completa del dato di un canale è legata al terzo parametro di configurazione attraverso il clock “ctrl_clk”, si evince che ai fini del corretto funzionamento della procedura di acquisizione è necessario che venga soddisfatta la seguente condizione:

(**acq_adc_conv_cycles** +5) × periodo “ctrl_clk” > 14 × periodo “ser_adc_clk”

Nella versione definitiva del firmware, essendo impostato il periodo di “ctrl_clk” a 10ns (100MHz) ed il periodo di “ser_adc_clk” a 16.7ns (60MHz), la precedente condizione equivale a soddisfare il seguente requisito:

acq_adc_conv_cycles ≥ 19

Dunque, mentre i primi due parametri di configurazione non presentano vincoli teorici sui relativi valori ammissibili, il terzo parametro non può assumere valori inferiori a 19.

Non appena la variabile “v_cycle_num” giunge al 32-esimo valore, completando il ciclo di acquisizioni legate ad un singolo evento, viene resettata in automatico e si alza il segnale SM_DATA_BUF_CNT_OUT_EN della macchina a stati del modulo *acquire*, dando avvio al processo di trasmissione dei dati convertiti e memorizzati nei buffer interni. Secondo quanto previsto dall’istruzione

```
data_buf_out_acq_reg <= ‘0’
   when (acq_enable=’0’ or data_buf_cnt_out_rst=’1’)
else ‘1’
   when ((clk_out’event and clk_out=’1’) and acq_en_reg=’1’ and
SM_DATA_BUF_CNT_OUT_EN=’1’);
```

l’ingresso della macchina a stati in SM_DATA_BUF_CNT_OUT_EN fa sì che il segnale data_buf_out_acq_reg venga alzato e permanga a “1” per tutta la durata della trasmissione dei dati seriali verso la BEE. I dati in uscita vengono serializzati sul segnale “sdata_out” a 12 bit sul *rising edge* di “sdata_out_clk”.

Anche in assenza di trigger di PDM, la procedura di acquisizione dei dati ADC per la varianza *free-running* rimane invariata rispetto a quanto descritto finora, abilitata sul fronte di salita di SM_ADC_CONV_EN (a cui corrisponde il *falling edge* di “mux_nrst”). In tal caso, il ritardo per il riavvio automatico dell’acquisizione dati tra un evento e il successivo viene impostato dal parametro di configurazione **acq_var_delay_cycles** (misurato in cicli del clock “ctrl_clk” a 100MHz).

Ora, ad ogni ciclo di scansione, i dati acquisiti dagli ADC secondo la procedura descritta vengono immagazzinati internamente in due buffer di memoria (uno per i dati HG ed uno per quelli LG) indipendentemente dal tipo di acquisizione effettuata (evento di trigger o evento di varianza). In altre parole, per come è stato concepito inizialmente il codice, un unico buffer di evento viene utilizzato per memorizzare ciascuna catena di dati (HG ed LG) provenienti dagli ADC. In assenza di trigger di PDM, ciò non crea problemi, in quanto per ogni acquisizione di varianza i buffer vengono sovrascritti di volta in volta con i nuovi valori acquisiti. Tuttavia, in presenza di trigger, durante la fase di trasmissione seriale che fa seguito all’acquisizione dei dati scientifici si verifica una successiva lettura dei dati provenienti dagli ADC. Pertanto, mentre i dati contenuti nei buffer interni (relativi al ciclo di lettura avviato in precedenza dal trigger) vengono trasferiti in uscita verso la BEE, i valori dei nuovi dati acquisiti durante la successiva sequenza di lettura vengono immagazzinati nei medesimi buffer, sovrascrivendo i valori precedenti ed alterando così i dati scientifici prodotti.

Tale inconveniente, legato alla presenza di un singolo buffer di evento per ciascuna tipologia di dati (HG ed LG), provocherebbe la corruzione dei dati inviati al modulo di varianza relativi all’acquisizione immediatamente successiva a quella innescata dal verificarsi di un trigger di PDM. A titolo illustrativo, in Fig. 44 è raffigurata una simulazione iSIM nella quale è possibile visualizzare il problema descritto. In particolare, un segnale di trigger di PDM attiva il processo di acquisizione, al termine del quale i dati acquisiti e memorizzati nei due buffer di evento vengono trasmessi serialmente verso la BEE ed alterati dal successivo ciclo di acquisizione automaticamente innescato, che determina una sovrascrittura simultanea dei medesimi buffer di dati.

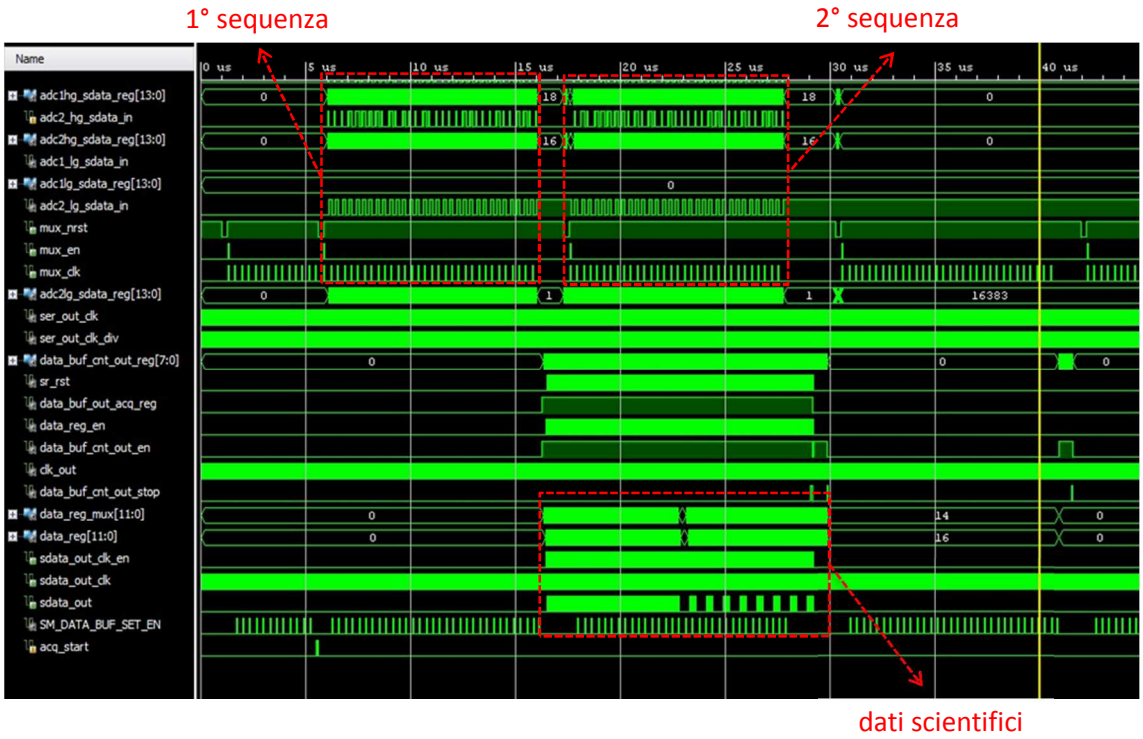


Fig. 44. Simulazione iSIM relativa all'acquisizione di un evento scientifico nella precedente versione del codice. Durante la fase di trasmissione dei dati scientifici verso la BEE viene effettuata una successiva sequenza di lettura dagli ADC, che determina una alterazione dei dati trasmessi alla BEE a causa della sovrascrittura simultanea sui medesimi buffer d'evento.

Per evitare l'inconveniente, è stata sviluppata ed apportata una modifica al codice sorgente del modulo *acquire*, in grado di inibire il processo di acquisizione dagli ADC durante tutto il periodo di tempo necessario alla trasmissione seriale dei 128 dati relativi ad un evento di trigger. A tal fine, è stato sfruttato il segnale `data_buf_out_acq_reg`, la cui assegnazione è stata riportata in precedenza nel presente paragrafo, per generare una condizione relativa all'abilitazione del processo di acquisizione. In particolare, poiché `data_buf_out_acq_reg`, legato allo stato `SM_DATA_BUF_CNT_OUT_EN`, rimane alto per tutta la durata della trasmissione seriale verso la BEE, si è pensato di aggiungere su tale segnale un controllo associato all'abilitazione dell'acquisizione automatica dei dati di varianza. Nello specifico, il segnale che segna l'avvio del ciclo di lettura dagli ADC è descritto dal seguente stralcio di codice:

```
auto_cyc_cnt_en <= '0' when (acq_enable='0' or acq_var_enable='0' or
acq_en_reg='1' or (SM_ACQ_VALID_RST='1' and acq_hold_enable='0'))
else '1';
```

Aggiungendo all'assegnazione a "0" del segnale `auto_cyc_cnt_en` il controllo su `data_buf_out_acq_reg`, si può bloccare l'avvio del processo di acquisizione per tutto il tempo necessario alla trasmissione seriale dei dati scientifici:

```
auto_cyc_cnt_en <= ‘0’ when (data_buf_out_acq_reg=’1’ or
acq_enable=’0’ or acq_var_enable=’0’ or acq_en_reg=’1’ or
(SM_ACQ_VALID_RST=’1’ and acq_hold_enable=’0’))
else ‘1’;
```

In Fig. 45 è possibile osservare l’effetto della modifica effettuata al codice. Durante la trasmissione seriale di una sequenza di dati scientifici, è inibito il processo di acquisizione dagli ADC, evitando pertanto l’alterazione dei dati scientifici trasmessi dovuta alla sovrascrittura dei buffer di evento.

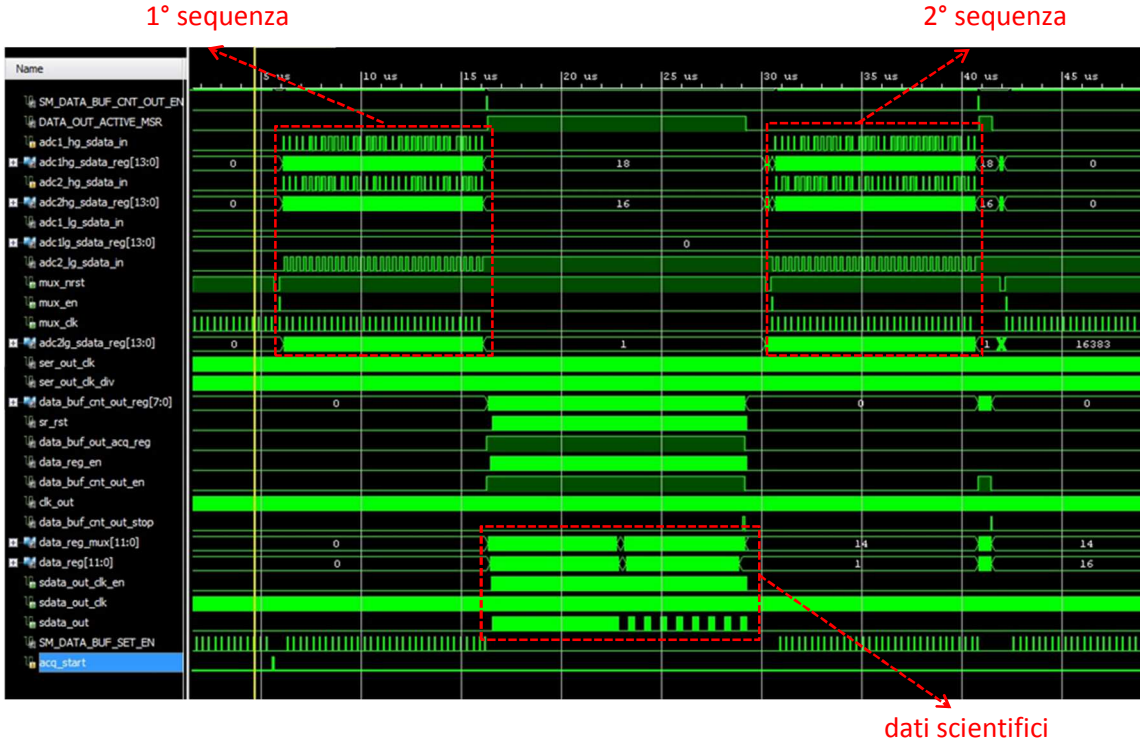


Fig. 45. Simulazione iSIM relativa all’acquisizione di un evento scientifico nella versione aggiornata del codice. Durante la trasmissione dei dati scientifici verso la BEE viene inibito il processo di acquisizione dagli ADC fino al completamento della trasmissione seriale, evitando in tal modo la sovrascrittura simultanea dei buffer d’evento.

La sopracitata modifica non inficia le altre funzionalità del modulo *acquire*, poiché comporta un semplice ritardo nell’acquisizione del successivo ciclo di dati di varianza che fa seguito alla trasmissione seriale di una sequenza di dati scientifici.

4.9 Disabilitazione della lettura dei trigger in trasmissione dati ()

Dalle simulazioni iSIM sul modulo *acquire*, si è rilevata una problematica simile a quella riscontrata nella sezione precedente, relativamente all'alterazione dei dati scientifici durante la trasmissione seriale verso la BEE. In particolare, in seguito all'arrivo di un PDM trigger ed alla conseguente attivazione del segnale "acq_start", l'eventuale occorrenza di un successivo segnale trigger nella fase di trasmissione dei dati relativi all'evento in corso, determina la corruzione dei dati trasmessi.

A titolo illustrativo, in Fig. 46 è raffigurata una simulazione iSIM nella quale è possibile visualizzare il problema descritto. In particolare, un segnale di trigger di PDM attiva il processo di acquisizione, al termine del quale i dati acquisiti e memorizzati nei due buffer di evento vengono trasmessi serialmente verso la BEE ed alterati dal verificarsi di un successivo trigger durante la fase di trasmissione, che determina una sovrascrittura simultanea dei medesimi buffer di dati.

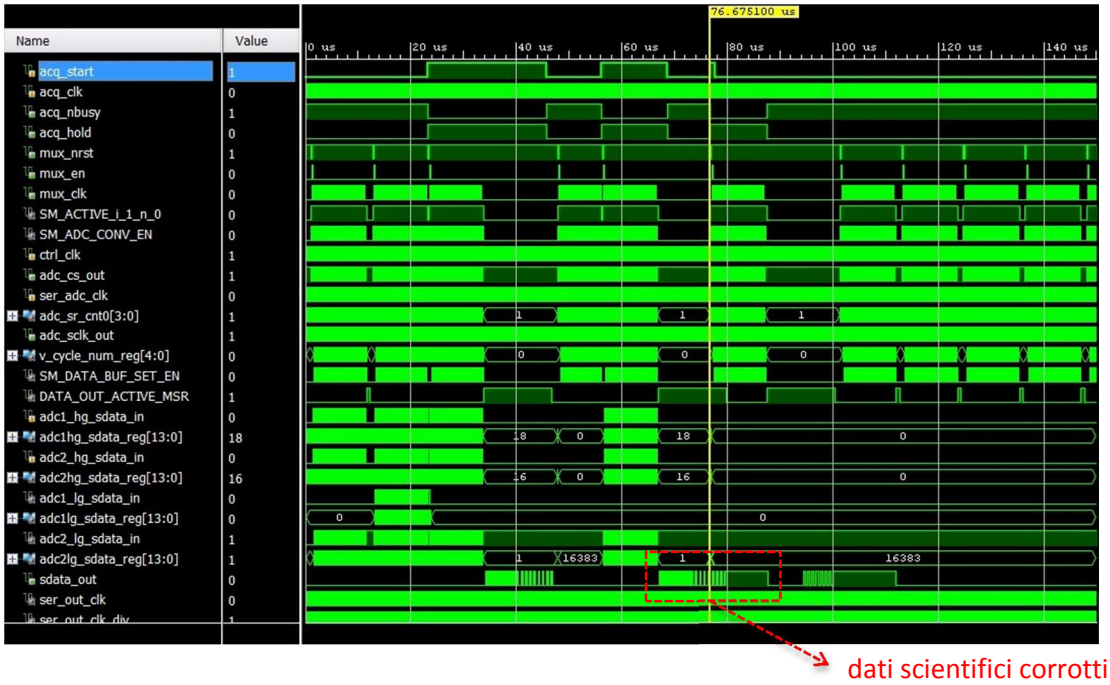


Fig. 46. Simulazione iSIM relativa all'acquisizione di un evento scientifico nella versione precedente del codice. Durante la fase di trasmissione dei dati scientifici verso la BEE viene simulata una successiva occorrenza del trigger di PDM, che determina una alterazione dei dati trasmessi alla BEE a causa della sovrascrittura simultanea sui medesimi buffer d'evento.

Per evitare l'inconveniente, è stata sviluppata ed apportata un'altra modifica al codice sorgente del modulo *acquire*, in grado di rendere trasparente il segnale di acquisizione "acq_start" durante tutto il periodo di tempo necessario alla trasmissione seriale dei 128 dati relativi ad ogni evento di trigger. A tale scopo, è stato sfruttato anche in questo caso il segnale `data_buf_out_acq_reg` per stabilire una condizione che inibisca la lettura del segnale di acquisizione durante la trasmissione seriale dei dati. Nello specifico,

i due segnali interni al modulo *acquire* che coinvolgono la lettura dei trigger sono descritti dal seguente stralcio di codice:

```
acq_en_reg <= ‘0’ when (acq_enable=’0’) else acq_start when
   ((acq_clk’event and acq_clk=’1’) and acq_en_comb=’1’);
acq_rst_reg <= ‘1’ when (acq_en_reg=’0’) else acq_start when
   ((acq_clk’event and acq_clk=’1’) and acq_inhibit_reg=’0’);
```

In particolare, `acq_rst_reg` ed `acq_en_reg` sono posti rispettivamente a “0” e ad “1” quando il segnale `acq_enable` (che determina l’attivazione del processo di acquisizione) è basso; in caso contrario, vengono entrambi ad assumere il valore del segnale `acq_start` sul fronte di salita del clock `acq_clk` e al simultaneo verificarsi delle condizioni `acq_en_comb=’1’` e `acq_inhibit_reg=’0’`, rispettivamente. Ora, l’occorrenza di un trigger di evento scientifico durante la fase di trasmissione verrebbe a modificare, secondo quanto descritto dalle due precedenti assegnazioni, lo stato logico dei due segnali `acq_rst_reg` ed `acq_en_reg`, che innescherebbero una immediata sovrascrittura dei due buffer di evento, alterando la trasmissione seriale. Si è reso necessario, pertanto, trovare una soluzione che impedisse anche la lettura di successivi trigger durante la trasmissione seriale.

Aggiungendo all’assegnazione ad `acq_start` dei due segnali interni `acq_rst_reg` ed `acq_en_reg` il controllo su `data_buf_out_acq_reg`, si può inibire la lettura di successivi segnali di trigger per tutto il tempo necessario alla trasmissione seriale del primo trigger di evento scientifico. In particolare, ponendo

```
acq_en_reg <= ‘0’ when (acq_enable=’0’) else acq_start when
   ((acq_clk’event and acq_clk=’1’) and acq_en_comb=’1’
                                    and data_buf_out_acq_reg=’0’);
acq_rst_reg <= ‘1’ when (acq_en_reg=’0’) else acq_start when
   ((acq_clk’event and acq_clk=’1’) and acq_inhibit_reg=’0’
                                    and data_buf_out_acq_reg=’0’);
```

è possibile rendere trasparente la lettura di un trigger durante la trasmissione seriale dei dati scientifici.

Da notare, a tal proposito, che se un fronte di salita del segnale “acq_start” si verifica invece durante il precedente processo di acquisizione, allora tale segnale viene automaticamente trascurato, e il problema di una successiva lettura di trigger non si pone. Tale assunto è stato verificato in simulazione, dove il segnale “acq_start” è stato posto ad “1” in corrispondenza dell’acquisizione dei dati relativi ad un evento scientifico, senza notare alterazioni durante la successiva fase di trasmissione.

In Fig. 47 si può osservare l’effetto della modifica effettuata al codice. La linea gialla in figura segna l’istante temporale in cui si verifica il trigger. Dunque, durante la trasmissione seriale di una sequenza di dati scientifici, è stato inibito il processo di lettura di successivi segnali di trigger, evitando pertanto l’alterazione dei dati scientifici trasmessi dovuta alla sovrascrittura dei buffer di evento.

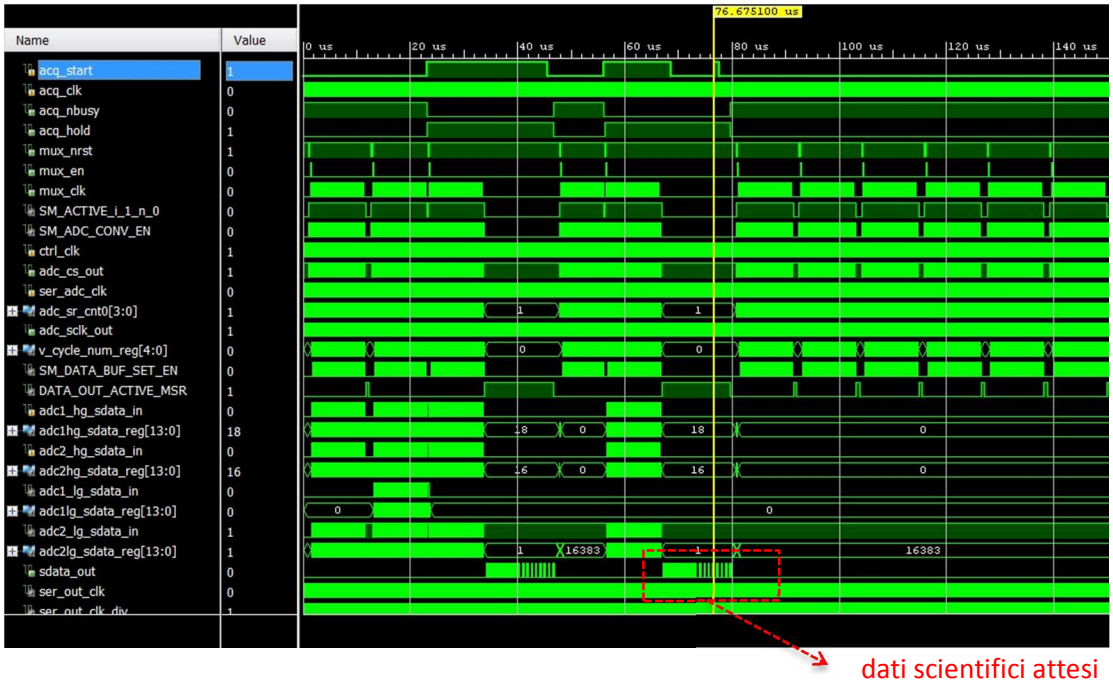


Fig. 47. Simulazione iSIM relativa all'acquisizione di un evento scientifico nella versione aggiornata del codice. Durante la trasmissione dei dati scientifici verso la BEE viene inibita la successiva lettura di eventuali trigger di PDM fino al completamento della trasmissione seriale, evitando in tal modo la sovrascrittura simultanea dei buffer d'evento.

La sopracitata modifica non inficia le altre funzionalità del modulo *acquire*, poiché coinvolge istruzioni legate esclusivamente alla lettura del segnale di trigger di PDM.

4.10 Generazione di un blocco HW di monostabili per i trigger ()

Per migliorare l'efficienza dell'algoritmo di trigger nell'individuazione delle adiacenze tra pixel, è stato realizzato e implementato un blocco HW costituito da 64 circuiti monostabili a larghezza d'impulso programmabile, per allungare e rendere costante la larghezza temporale dei 64 segnali di trigger provenienti dagli ASIC. Tale blocco è stato inserito a monte del processamento digitale dei trigger, in modo da inviare ai moduli *trig_gen* e *trig_count* dei segnali di trigger di larghezza costante. Il beneficio così ottenuto nel miglioramento dell'efficienza del trigger di PDM è illustrato graficamente in Fig. 48.

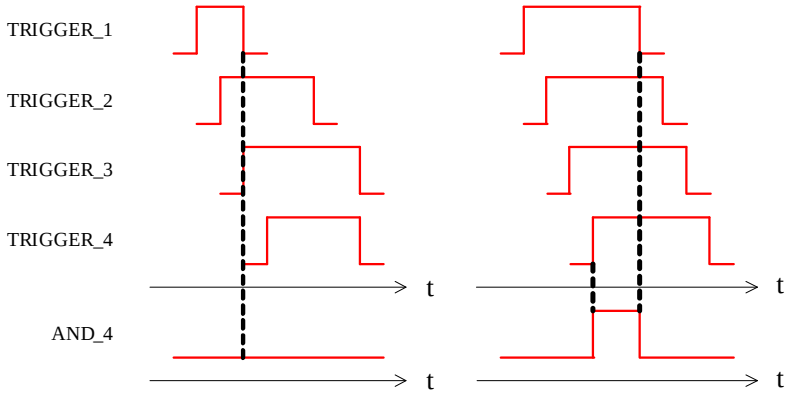


Fig. 48. Miglioramento dell'efficienza di trigger. Nel grafico di sinistra, le 4 adiacenze fra i trigger non vengono rilevate a causa della diversa larghezza degli impulsi; nel grafico di destra, i trigger hanno tutti la stessa durata prefissata, sufficiente a garantire l'individuazione dell'adiacenza.

In considerazione della modalità asincrona con cui giungono i trigger all'FPGA, si è reso necessario implementare 64 multivibratori monostabili, uno per ogni segnale di trigger. È stato definito per tutti i monostabili un identico ritardo programmabile a 4 bit in step da 3.3ns, utilizzando il clock veloce a 300MHz disponibile all'interno dell'FPGA. Di conseguenza, la massima larghezza ammissibile per ciascun segnale di trigger è fissata in circa 50ns. In Fig. 49 è illustrato il diagramma a blocchi del nuovo modulo.

Fig. 49. Schema a blocchi del modulo di generazione dei ritardi programmabili per i trigger.

In virtù del parametro di configurazione “window_mon” a 4 bit è possibile impostare la larghezza dei trigger in step del segnale di clock “clk” a 3.3ns. Come si evince in figura, è stata altresì prevista la possibilità di inibire la formazione di trigger di durata prefissata utilizzando un multiplexer. In particolare, se la sequenza dei 4 bit di configurazione è pari a “0000” il blocco HW di generazione dei ritardi viene praticamente bypassato, sicché i 64 trigger provenienti dagli ASIC giungeranno direttamente al blocco *trig_mask*.

L’intero codice sorgente VHDL del nuovo modulo *monostable* è il seguente:

```
entity monostable is
    Port ( trigger_in : in STD_LOGIC_VECTOR (63 downto 0);
           clk : in STD_LOGIC;
           window_mon : in STD_LOGIC_VECTOR (3 downto 0);
           trigger_mon : out STD_LOGIC_VECTOR (63 downto 0)
         );
end Monostable;

architecture Behavioral of Monostable is

type array64_type is array(integer range <>) of
                     std_logic_vector(3 downto 0);

signal count_window : array64_type(0 to 63) := (others=>”0000”);
signal enable_large : STD_LOGIC_VECTOR (63 downto 0):= (others=>’0’);
signal res_trig : STD_LOGIC_VECTOR (63 downto 0);
signal trigger_in_ff : STD_LOGIC_VECTOR (63 downto 0);

begin

trigger_mon <= trigger_in when window_mon = “0000” else enable_large;
trigger_in_ff <= trigger_in when (clk’event and clk = ‘1’);

process(clk)
begin
if (clk’event and clk = ‘1’) then
for I in 0 to 63 loop
   if trigger_in(i) = ‘1’ and trigger_in_ff(i) = ‘0’ then
      enable_large(i) <= ‘1’;
   end if;
   if enable_large(i) = ‘1’ then
      if (count_window(i) = window_mon – ‘1’) then
         enable_large(i) <= ‘0’;
         count_window(i) <= “0000”;
      else
         count_window(i) <= count_window(i) + ‘1’;
      end if;
   end if;
end loop;
end if;
end process;

end Behavioral;
```

La tabella sottostante riassume i valori programmabili della larghezza degli impulsi di trigger in corrispondenza dei valori dei 4 bit del parametro di configurazione.

Command	*window_mon*	*trigger width*
1	0000	actual trigger
2	0001	1 × 3.3ns = 3.3ns
3	0010	2 × 3.3ns = 6.6ns
4	0011	3 × 3.3ns = 9.9ns
5	0100	4 × 3.3ns = 13.2ns
6	0101	5 × 3.3ns = 16.5ns
7	0110	6 × 3.3ns = 19.8ns
8	0111	7 × 3.3ns = 23.1ns
9	1000	8 × 3.3ns = 26.4ns
10	1001	9 × 3.3ns = 29.7ns
11	1010	10 × 3.3ns = 33.0ns
12	1011	11 × 3.3ns = 36.3ns
13	1100	12 × 3.3ns = 39.6ns
14	1101	13 × 3.3ns = 42.9ns
15	1110	14 × 3.3ns = 46.2ns
16	1111	15 × 3.3ns = 49.5ns

In considerazione della durata media stimata dei segnali digitali in uscita ai circuiti discriminatori dei CITIROC, la selezione dei primi valori del segnale `window_mon` (ad eccezione del valore “0000) potrebbero comportare un restringimento, piuttosto che un allargamento, della larghezza dei segnali di trigger effettivi e pertanto potrebbero non venire adoperati. Difatti, è stato verificato che segnali di trigger di ampiezza inferiore ai 10ns non vengono correttamente discriminati dai comparatori degli ASIC.

Il blocco HW dei multivibratori monostabili è stato inserito come componente all’interno di una entità strutturale più generale contenente anche il modulo *trig_mask*, come illustrato in Fig. 49. Nella Fig. 50 è rappresentata la porzione del diagramma a blocchi dell’FPGA con l’entità che include il modulo *monostable* per la regolazione della larghezza degli impulsi ed il modulo *trig_mask* per la mascheratura dei trigger.

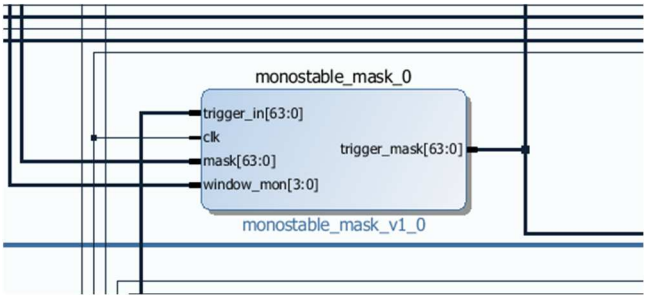


Fig. 50. Diagramma a blocchi del modulo monostable_mask.

Per la generazione del parametro di configurazione `window_mon` a 4 bit sono stati utilizzati in HW 4 bit liberi di un banco di registri a 32 bit già presente all'interno del blocco *inaf_fee_core_registers*, come illustrato in Fig. 51.

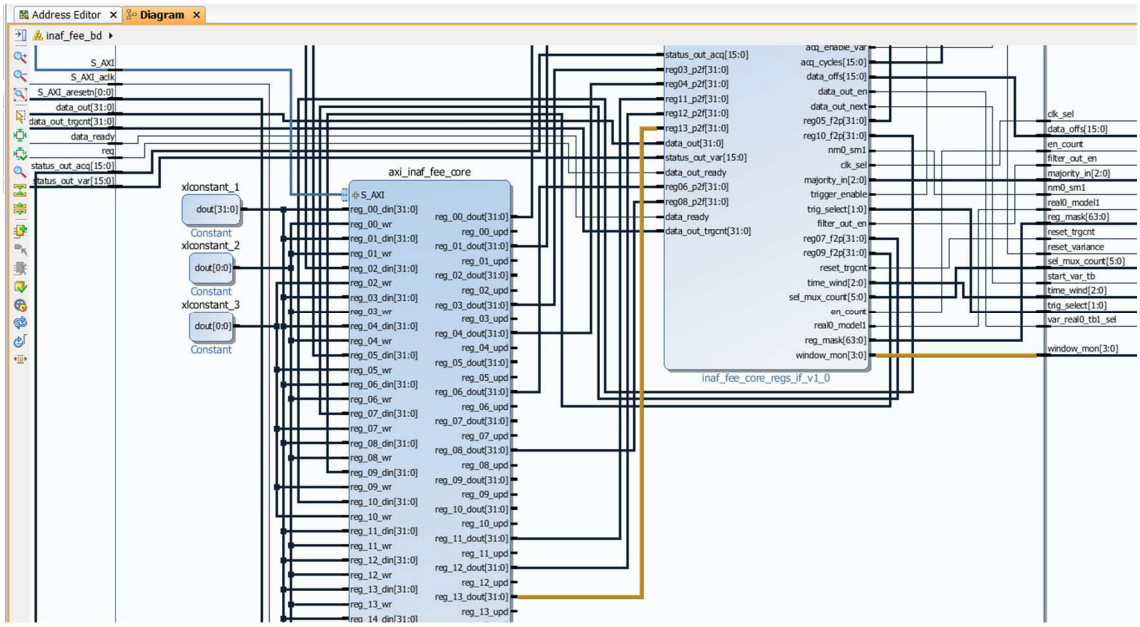


Fig. 51. Diagramma a blocchi dei registri comandati dal MicroBlaze, in cui è evidenziato il nuovo segnale di configurazione che determina la larghezza degli impulsi di trigger.

In particolare, nell'entità del blocco che definisce i segnali di configurazione dei registri gestiti dal processore è stata aggiunta la definizione seguente:

```
window_mon : out std_logic_vector(3 downto 0);
```

ed è stata altresì inserita nel corpo della struttura la seguente assegnazione:

```
-- monostable
   window_mon(3 downto 0) <= reg13_p2f(3 downto 0);
```

relativamente all'assegnazione di 4 bit del banco libero di registri a 32 bit `reg13_p2f` al segnale di uscita `window_mon` a 4 bit, che andrà dunque a configurare il modulo *monostable*.

Per quanto concerne il comparto SW gestito dal MicroBlaze, sono state inserite alcune istruzioni e modificate alcune *routine* in vari moduli in linguaggio C.

I 4 bit aggiuntivi del registro `window_mon` sono stati aggiunti alla tabella degli SWITCH sfruttando 4 bit liberi nel primo byte della tabella, e pertanto non si è resa necessaria nessuna modifica alla *command line interface* (modulo SW cli.c su SDK) sul controllo della lunghezza dei bit della tabella degli SWITCH.

È stata introdotta una variabile di riferimento per i due banchi di registri da 32 bit nella struttura dichiarativa del codice (modulo SW global.h su SDK), utilizzata successivamente per l'assegnazione dei bit della tabella degli SWITCH:

```
typedef struct
{
...
unsigned char wind_mon; // switch_table_byte_1_bit_2/5
...
} Pdm_struct;
```

dove il commento in verde accanto alla definizione specifica la posizione del byte (e relativi bit) della tabella degli SWITCH che si vuole assegnare alla variabile.

Il codice relativo alla gestione dei bit aggiuntivi (modulo SW fee_cmds.c su SDK) è stato aggiornato. In particolare, i bit 2, 3, 4 e 5 del primo byte del vettore `SwitchTable` della *command line interface* sono stati assegnati rispettivamente ai bit 0, 1, 2 e 3 del registro `window_mon`. A tale scopo, nel comando SET SWITCH (**`Cmd11_SetSwitch`**) è stata aggiunta la seguente assegnazione:

```
void Cmd11_SetSwitch(Pdm_struct *PDM, Cli_struct *Cli){
...
// aggiunta assegnazione
PDM->wind_mon = (Cli->SwitchTable[1])&(0x3C);
             // switch_table_byte_1_bit_2/5
             // "0x3C"="00111100"
...}
```

Inoltre, nello stesso comando SET SWITCH (**`Cmd11`**) è stata aggiunta la seguente istruzione (modulo SW fee_cmds.c su SDK):

```
void Cmd11_SetSwitch(Pdm_struct *PDM, Cli_struct *Cli){
...
// aggiunta chiamata
Write_wind_mon((u8) (*PDM).wind_mon);
...}
```

attraverso cui è possibile scrivere sui primi 4 bit del registro `reg13_p2f` i 4 valori dei bit 2-5 del primo byte della tabella di configurazione degli SWITCH `SwitchTable` impostati dal software di gestione.

La scrittura dei 4 valori assegnati in `PDM->wind_mon` avviene mediante la seguente procedura (modulo SW inaf_fee_core_regs.c su SDK):

```
void Write_wind_mon(u8 Data)
{
reg32 tempReg = 0;
tempReg = * ( volatile reg32 *) (INAF_FEE_CORE_REGS_BASEADDRESS
                                 + REG13_OFFSET);
// scrive il registro reg13_p13f in tempReg
u8 wind_mon[7];
int i = 0;
      for (i = 0; i < sizeof(wind_mon); i++){
            wind_mon[i] = 0;
      }
      for (i = 0; i < sizeof(wind_mon); i++){
            if (Data%2 == 1) { wind_mon[i] = 1; }
            Data = Data >> 1;
      // scrive Data=(*PDM).wind_mon nella variabile wind_mon
      }
            if (wind_mon[2] == 1){
                tempReg |= wind_mon_0_MASK; //set tempReg
            }
            else{
                tempReg &= ~wind_mon_0_MASK; //clear tempReg
            }
            if (wind_mon[3] == 1){
                tempReg |= wind_mon_1_MASK; //set tempReg
            }
            else{
                tempReg &= ~wind_mon_1_MASK; //clear tempReg
            }
            if (wind_mon[4] == 1){
                tempReg |= wind_mon_2_MASK; //set tempReg
            }
            else{
                tempReg &= ~wind_mon_2_MASK; //clear tempReg
            }
            if (wind_mon[5] == 1){
                tempReg |= wind_mon_3_MASK; //set tempReg
            }
            else{
                tempReg &= ~wind_mon_3_MASK; //clear tempReg
            }
      * ( volatile reg32 *) (INAF_FEE_CORE_REGS_BASEADDRESS + REG13_OFFSET)
        = tempReg;
      // scrive il registro reg13_p13f con i valori di tempReg
}
```

in cui il valore dell'*offset* `REG13_OFFSET` individua in HW il registro `reg13_p2f`.

I valori delle maschere per l'identificazione dei 4 bit del registro `reg13_p2f` attraverso la variabile `reg32 tempReg` (da cui attinge i 32 valori) sono stati introdotti tramite le seguenti assegnazioni (modulo SW inaf_fee_core_regs.h su SDK):

```
#define wind_mon_0_MASK   0x00000001   //reg13_p13f(0)
#define wind_mon_1_MASK   0x00000002   //reg13_p13f(1)
#define wind_mon_2_MASK   0x00000004   //reg13_p13f(2)
#define wind_mon_3_MASK   0x00000008   //reg13_p13f(3)
```

Pertanto, i bit 2-5 del primo byte della tabella degli SWITCH (assegnati ai corrispondenti bit del byte `PDM->wind_mon`) verranno riportati nei bit 0(LSB)-3(MSB) del segnale di configurazione `window_mon` del modulo *monostable* (essendo scritti nei rispettivi bit 0-3 di `reg13_p2f`). Gli altri 4 bit fuori maschera ("0x3C") di `PDM->wind_mon`, invece, restano settati a 0 e non vengono utilizzati dalla funzione **`Write_wind_mon`**.

La funzione aggiunta è stata opportunamente introdotta nella sezione dichiarativa del codice associato ai registri (modulo SW inaf_fee_core_regs.h su SDK):

```
void Write_wind_mon(u8 Data);
```

Nel software di gestione dell'FPGA in modalità *stand alone*, la sezione relativa alla impostazione della durata dei trigger è rappresentata dalla casella di controllo TRIGGER WIDTH presente tra i settaggi, illustrati in Fig. 39, relativi alla configurazione della tabella degli SWITCH.

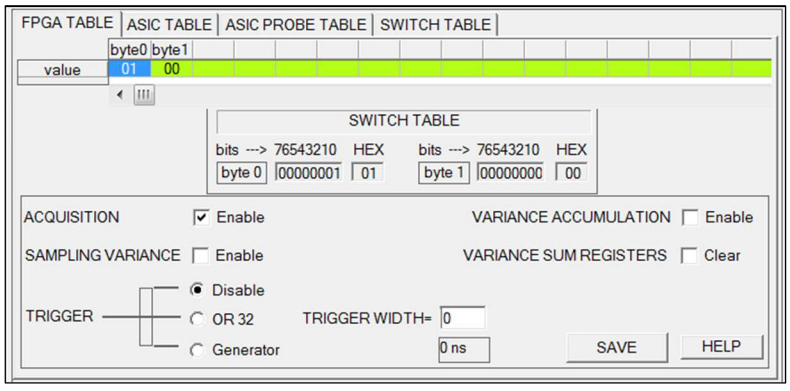


Fig. 52. Sezione del software di gestione dell'FPGA (in modalità stand alone) relativa ai settaggi per l'impostazione della larghezza dei trigger.

4.11 Generazione di una finestra temporale per i segnali di trigger ()

Un'altra modifica al firmware dell'FPGA di *front-end* ha riguardato la realizzazione di un blocco HW che genera una finestra temporale fissa per i segnali di trigger in modalità BEE, ed inibisce al contempo la lettura dei trigger provenienti dagli ASIC dopo un certo tempo prefissato dalla ricezione del camera trigger proveniente dalla BEE, per evitare conteggi spuri durante i test di illuminazione con i LED.

In particolare, è stato progettato un ulteriore blocco HW, *camera_trig_mask*, costituito da un multivibratore monostabile a ritardo fisso (100ns), agganciato al segnale di trigger proveniente dalla BEE e comandato dallo stesso segnale di clock `clk` utilizzato per la programmazione dei ritardi dei trigger nel modulo *monostable*. In aggiunta, è stato previsto un segnale di controllo supplementare, `val_evt`, per disabilitare la generazione di tale ritardo fisso sul segnale di uscita, come illustrato nella Fig. 53.

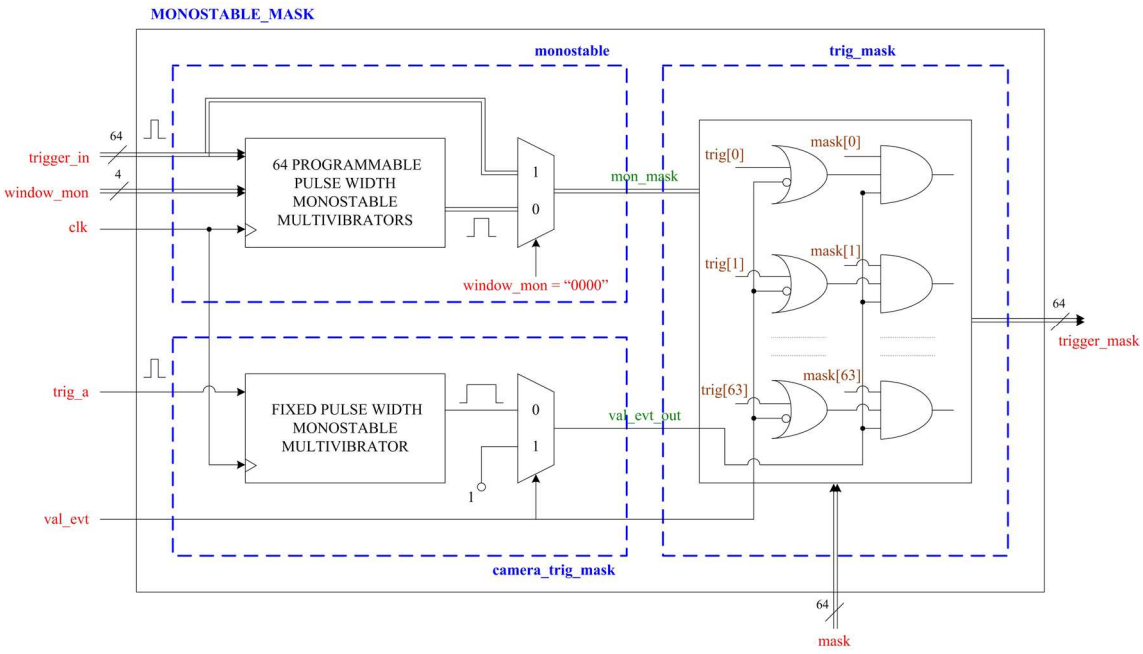


Fig. 53. Schema a blocchi del modulo monostable_mask.

L'uscita di tale modulo, come rappresentato in figura, deve pilotare il blocco logico per la mascheratura dei trigger, in modo tale da non alterare la possibilità di inibire i segnali di trigger tramite il parametro `mask`, e allo stesso tempo in grado di fornire in uscita 64 segnali di ampiezza fissa al verificarsi di un fronte di salita del segnali di camera trigger proveniente dalla BEE.

A tale scopo, è stato aggiunto il blocco *camera_trig_mask* per la generazione del ritardo fisso, ed è stato modificato il modulo di mascheratura *trig_mask* per inibire la lettura di segnali di trigger degli ASIC durante questo lasso di tempo. Infine, tali blocchi, unitamente al modulo *monostable* per la programmazione dei ritardi dei trigger degli ASIC, sono stati inseriti all'interno dell'entità strutturale *monostable_mask*, il cui codice VHDL è riportato nelle righe seguenti:

```
entity monostable_mask is
   port (
      trigger_in : in STD_LOGIC_VECTOR (63 downto 0);
      trig_a : in STD_LOGIC;
      val_evt : in STD_LOGIC;
      clk : in STD_LOGIC;
      mask : in STD_LOGIC_VECTOR (63 downto 0);
      trigger_mask : out STD_LOGIC_VECTOR (63 downto 0);
      window_mon : in STD_LOGIC_VECTOR (3 downto 0)
   );
end monostable_mask;

architecture Behavioral of monostable_mask is

component trig_mask is // componente trig_mask
   port (
      trig_in : in STD_LOGIC_VECTOR (63 downto 0);
      mask : in STD_LOGIC_VECTOR (63 downto 0);
      cam_trig_mask : in STD_LOGIC;
      val_evt : in STD_LOGIC;
      trigger_mask : out STD_LOGIC_VECTOR (63 downto 0)
 );
 end component;

component monostable is // componente monostable
   port (
      trigger_in : in STD_LOGIC_VECTOR (63 downto 0);
      clk : in STD_LOGIC;
      window_mon : in STD_LOGIC_VECTOR (3 downto 0);
      trigger_mon : out STD_LOGIC_VECTOR (63 downto 0)
   );
end component;

component camera_trig_mask is // componente camera_trig_mask
   port (
      trig_a : in STD_LOGIC;
      clk : in STD_LOGIC;
      val_evt : in STD_LOGIC;
      val_evt_out : out STD_LOGIC
   );
end component;

signal mon_mask : std_logic_vector(63 downto 0);
signal val_evt_out : std_logic;

begin

monostable_1 : monostable
port map(
   trigger_in => trigger_in,
   clk => clk,
   window_mon => window_mon,
   trigger_mon => mon_mask
);
```

```
trig_mask_1 : trig_mask
port map(
   trig_in => mon_mask,
   mask => mask,
   val_evt => val_evt,
   cam_trig_mask => val_evt_out,
   trigger_mask => trigger_mask
);

camera_trig_mask_1 : camera_trig_mask
port map(
   trig_a => trig_a,
   clk => clk,
   val_evt => val_evt,
   val_evt_out => val_evt_out
);

end Behavioral;
```

ed è costituito fondamentalmente dalle definizioni dell'entità e dei suoi componenti interni, nonché dall'assegnazione dei segnali di interconnessione tra i vari blocchi nella specifica sezione della mappatura delle porte (*port map*).

Il componente *monostable* con i 64 multivibratori monostabili a ritardo programmabile rimane inalterato rispetto a quello progettato e descritto in precedenza. Il nuovo modulo `camera_trig_mask` per la generazione del ritardo fisso di 100ns sul fronte di salita del camera trigger riceve in ingresso il segnale `trig_a` ed il bit di configurazione `val_evt`, oltre al clock `clk` da 3.3ns, e fornisce in uscita il segnale `val_evt_out`, che va a pilotare le porte logiche del blocco di mascheratura. Il codice sorgente del nuovo modulo è riportato di seguito:

```
entity camera_trig_mask is
   port (
      trig_a : in STD_LOGIC;
      clk : in STD_LOGIC;
      val_evt : in STD_LOGIC;
      val_evt_out : out STD_LOGIC
   );
end camera_trig_mask;

architecture Behavioral of camera_trig_mask is

signal trig_a_ff : STD_LOGIC;
signal count_window : integer range 0 to 40; -- x 3.3ns
signal val_evt_large : STD_LOGIC;

begin

trig_a_ff <= trig_a when (clk'event and clk = '1');
val_evt_out <= val_evt_large when val_evt ='0' else '1';
```

```
process(clk)
begin
if (clk'event and clk = '1') then
   if trig_a = '1' and trig_a_ff = '0' then
      val_evt_large <= '1';
   end if;
   if val_evt_large = '1' then
      if (count_window = 30 ) then – 31 x 3.3ns
         val_evt_large <= '0';
         count_window <= 0;
      else
         count_window <= count_window + 1;
      end if;
   end if;
end if;
end process;

end Behavioral;
```

Allo scopo di inibire la lettura dei trigger provenienti dagli ASIC al verificarsi del camera trigger (quando il parametro `val_evt` è settato a "0"), si è resa necessaria una modifica al blocco *trig_mask*. In particolare, come illustrato in Fig. 53, sono state aggiunge 64 porte OR, pilotate dai 64 segnali di trigger e dal parametro di configurazione in comune, fra i 64 trigger in uscita al modulo *monostable* e le 64 porte AND adibite alla mascheratura dei trigger. È stato inoltre aumentato il *fan-out* delle porte AND per consentirne un pilotaggio comune da parte del segnale di uscita del blocco *camera_trig_mask*. Il nuovo codice sorgente del blocco *trig_mask* è il seguente:

```
entity trig_mask is
   port (
      trig_in : in STD_LOGIC_VECTOR (63 downto 0);
      mask : in STD_LOGIC_VECTOR (63 downto 0);
      val_evt : in STD_LOGIC;
      cam_trig_mask : in STD_LOGIC;
      trigger_mask : out STD_LOGIC_VECTOR (63 downto 0)
   );
end trig_mask;

architecture Behavioral of trig_mask is

begin

process(trig_in, val_evt, mask, cam_trig_mask)
begin
   for i in 0 to 63 loop
      trigger_mask(i) <= (trig_in(i) or (not val_evt)) and mask(i)
                         and cam_trig_mask;
   end loop;
end process;

end Behavioral;
```

La tabella sottostante riassume le modalità di formazione dei trigger alla luce delle modifiche effettuate al blocco *monostable_mask*.

val_evt	***mask***	***trigger_mask***	***meanings***
0	0	0	no trigger
1	0	0	no trigger
0	1	*trig_a* formed	100 ns pulse: *trig_a* external trigger
1	1	*trigger_in*	*window_mon* = 0 -> actual pulse *window_mon* = 3 ÷ 15 * 3.3ns

Il nuovo blocco HW per la generazione della finestra temporale fissa è stato dunque inserito all'interno del diagramma a blocchi dell'FPGA come ulteriore componente di una entità strutturale (*monostable_mask*), che comprende anche il modulo *monostable* per la regolazione della larghezza degli impulsi, ed il modulo *trig_mask* per la mascheratura dei trigger, come illustrato in Fig. 53. Nella Fig. 54 è rappresentata la porzione del diagramma a blocchi dell'FPGA con l'entità *monostable_mask* aggiornata.

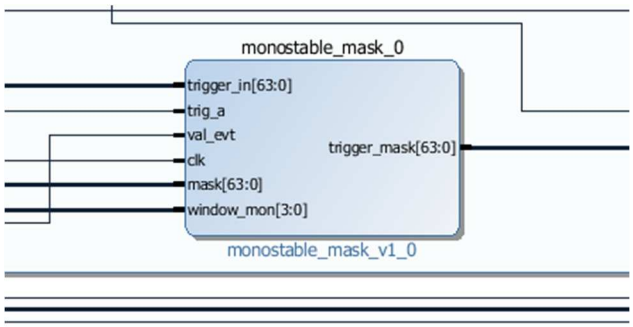


Fig. 54. Diagramma a blocchi aggiornato del modulo monostable_mask, che contiene al suo interno i componenti monostable, trig_mask e camera_trig_mask.

Per la gestione del segnale di configurazione `val_evt` è stato utilizzato l'omonimo parametro della tabella di configurazione dell'FPGA, utilizzato dal MicroBlaze per comandare il relativo ingresso di entrambi gli ASIC tramite GPIO.

In Fig. 55 è mostrato il diagramma a blocchi dell'FPGA con il modulo *gp_ios_block*, le cui prime due uscite ("o00" e "o01") vanno a pilotare i segnali degli ASIC "val_evt_p_1" e "val_evt_p_2" (entrambi ingressi digitali dei due CITIROC). Come si nota in figura, la prima uscita del blocco GPIO è stata collegata direttamente al segnale di configurazione `val_evt` del blocco *monostable_mask*. Più specificamente, il valore del bit relativo al segnale di comando proveniente dal processore viene condotto contemporaneamente alle due porte di uscita del blocco GPIO, che verranno applicate ai due pin di ingresso degli ASIC, mentre la prima di tali porte verrà utilizzata allo stesso tempo anche per il segnale di configurazione del blocco *monostable_mask*.

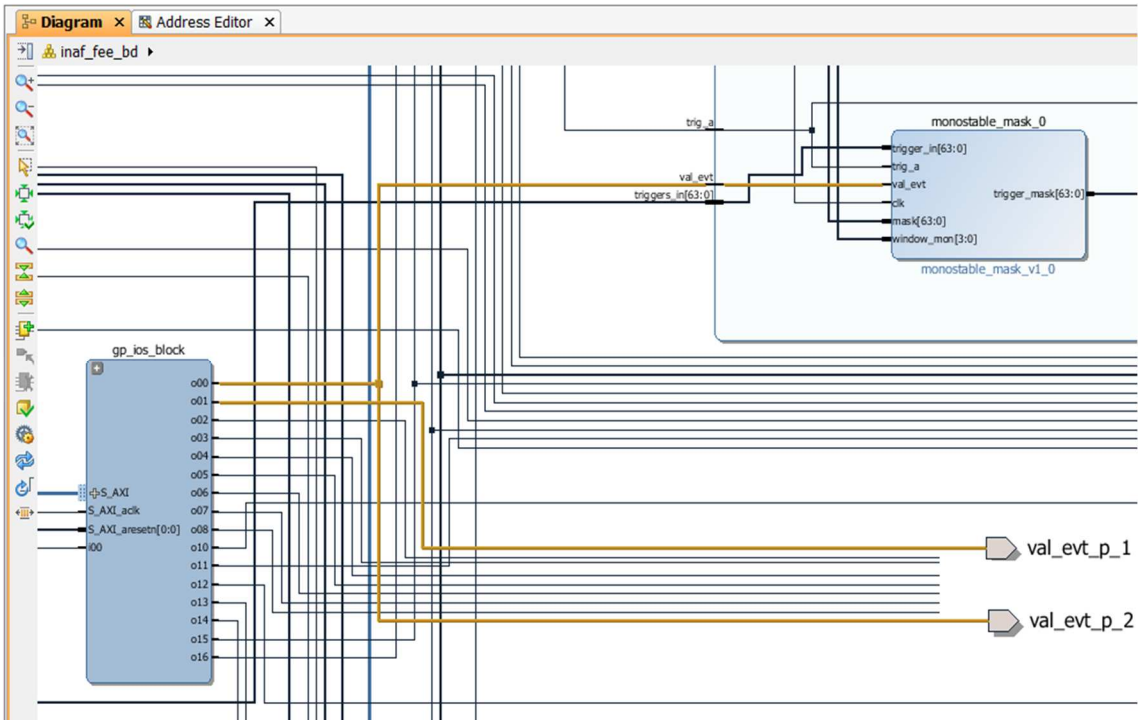


Fig. 55. Dettaglio del diagramma a blocchi dell'FPGA relativo alla gestione dei due segnali statici val_evt_p_1 e val_evt_p_2 attraverso il modulo GPIO. La prima delle due linee di uscita è collegata all'ingresso del modulo monostable_mask.

La variabile di riferimento utilizzata nella struttura dichiarativa del codice (modulo SW global.h su SDK) per la gestione dei due segnali d'ingresso ai due ASIC, "val_evt_p_1" e "val_evt_p_2", è rappresentata nel seguente stralcio di codice:

```
typedef struct
{
...
unsigned char val_evt_p_1_2; // fpga_table byte_7_bit_3
...
} Pdm_struct;
```

dove il commento in verde accanto alla definizione specifica la posizione del byte (e relativo bit) della tabella dell'FPGA che si assegna alla variabile.

L'inizializzazione della variabile (modulo SW helloworld.c su SDK) è rappresentata dalla seguente riga di codice all'interno della funzione **`ClearPDMStruct`**:

```
void ClearPDMStruct(Pdm_struct *PDM){
...
// inizializzazione
(*PDM).val_evt_p_1_2 = (unsigned char) 0;
...}
```

Il valore del terzo bit del settimo byte relativo al vettore `FpgaTable` della *command line interface* viene assegnato alla variabile `val_evt_p_1_2` all'interno del comando SEND TABLE FPGA (**`Cmd8`**) tramite la seguente procedura:

```
void Cmd8(Pdm_struct *PDM, Cli_struct *Cli){
...
if ( (Cli->FpgaTable[7] & MASKBYTE_BIT3) == MASKBYTE_BIT3){
                                          // fpga_table byte_7_bit_3
            PDM->val_evt_p_1_2 = 1;
      } else {
            PDM->val_evt_p_1_2 = 0;
      }
...}
```

dove il valore della maschera per l'identificazione del bit 3 del byte 7 della tabella di configurazione dell'FPGA viene introdotto mediante la seguente assegnazione (modulo SW global.h su SDK):

```
#define MASKBYTE_BIT3 0x08 // fpga_table byte_7_bit_3
```

L'assegnazione dei bit impostati dalla *command line interface* ai segnali "val_evt_p_1" e "val_evt_p_2" avviene all'interno del comando WRITE TABLE FPGA (**`Cmd11`**) tramite le seguenti istruzioni (modulo SW helloworld.c su SDK):

```
void Cmd11(Pdm_struct *PDM, Xgpio *Gpio){
...
      if ((*PDM).val_evt_p_1_2 == (unsigned char) 1){
         WrGpO(GP_IOS_BASEADDR, 0, 1);
         WrGpO(GP_IOS_BASEADDR, 1, 1);
      } else {
          WrGpO(GP_IOS_BASEADDR, 0, 0);
          WrGpO(GP_IOS_BASEADDR, 1, 0);
      }
...}
```

4.12 Risoluzione del problema *shift* dati nel modulo *acquire* ()

Da un'analisi dei vari segnali in uscita al modulo *acquire* visualizzati su ILA Chipscope è stato riscontrato che l'attivazione della varianza durante l'acquisizione dei dati scientifici determina, dopo un numero random di acquisizioni, uno *shift* di 7 posizioni relativamente ai dati seriali trasmessi verso la BEE.

In Fig. 56 ed in Fig. 57 sono riportati gli istogrammi di carica relativi al piedistallo LG del 15-esimo pixel, rispettivamente attivando e disattivando il campionamento automatico della varianza. Come si evince da un confronto dei grafici, il piedistallo relativo al pixel 15, nelle acquisizioni successive allo *shift*, modifica il suo valore medio assumendo il valore del piedistallo di un altro pixel, esattamente quello del pixel 8. Si osserva, inoltre, dallo *screenshot* dei segnali su ILA Chipscope, come l'indirizzo di memoria in cui viene scritto il primo dato scientifico dopo il verificarsi dello *shift* risulta 7 anziché 0.

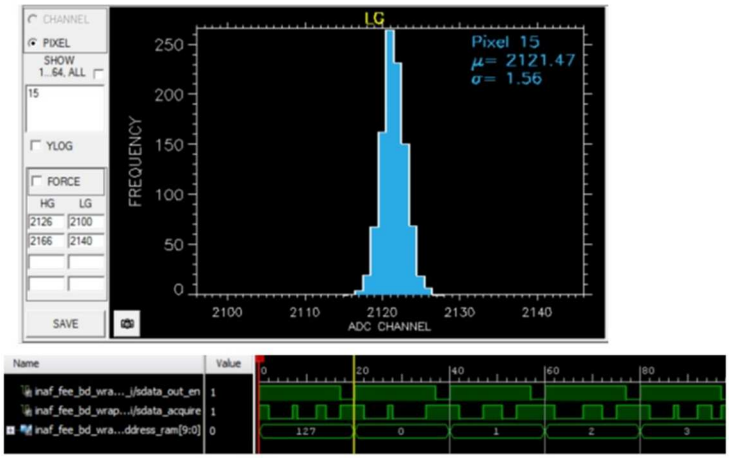


Fig. 56. Istogramma del 15-esimo pixel LG con varianza disattivata (1000 acquisizioni).

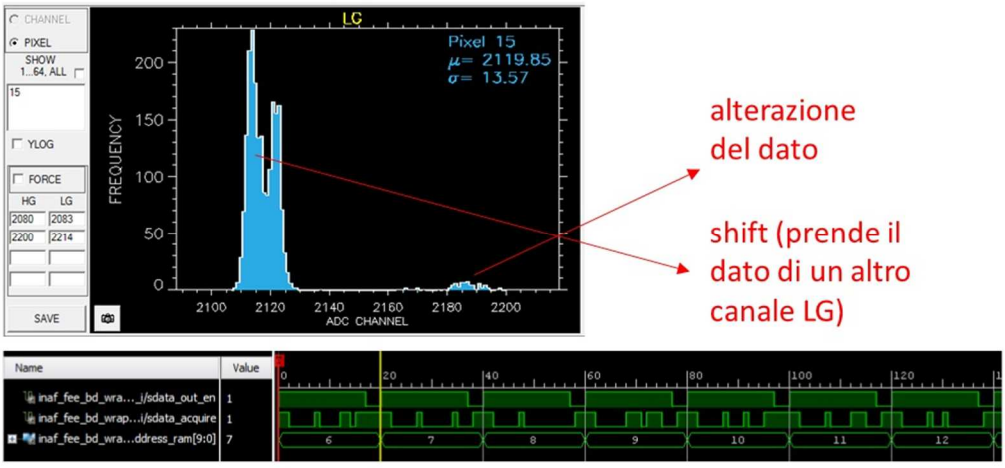


Fig. 57. Istogramma del 15-esimo pixel LG con varianza attivata (1000 acquisizioni).

In Fig. 58 è rappresentato il momento esatto, catturato tramite comandi di *smart trigger* su ILA Chipscope, in cui si verifica tale *shift* dei dati di uscita, in modo tale da facilitare la comprensione del problema e la determinazione della sua soluzione.

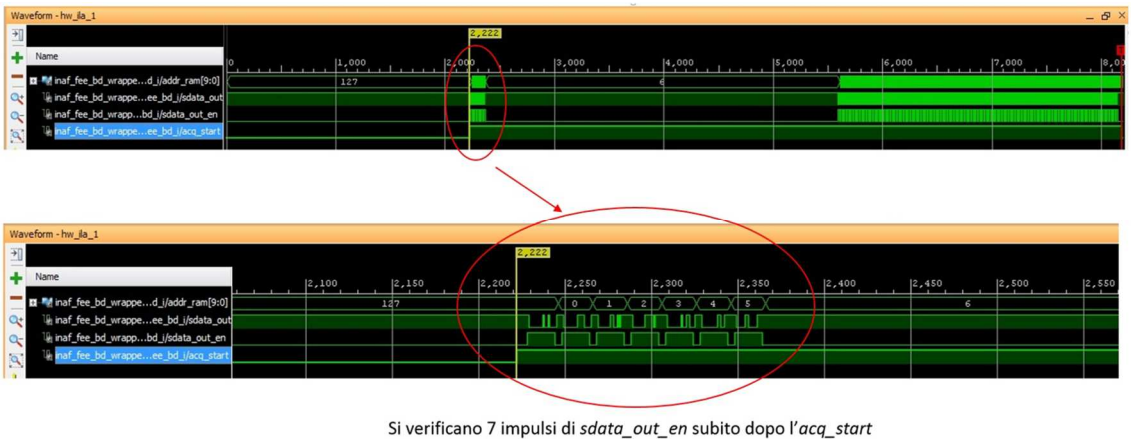


Fig. 58. Screenshot dei segnali forniti da ILA Chipscope in corrispondenza del momento in cui si verifica il problema dello shift dei dati.

La spiegazione del problema e la conseguente soluzione, descritte nelle righe seguenti, sono state suggerite dall'ideatore del codice (Francesco Russo, IASF Palermo).

Il problema si manifesta solo e soltanto se in prossimità della fine dell'acquisizione della varianza (per la precisione, al termine dell'acquisizione ma un piccolo istante prima dell'invio dei dati verso il relativo modulo di accumulo) si asserta il segnale "acq_start" (questo istante è una finestra temporale formata da 2 cicli di "ctrl_clk", quindi a 100MHz sono 20ns).

Quando questa condizione si realizza, allora accade che contemporaneamente all'invio dei dati verso il modulo della varianza parte l'invio degli stessi dati (ma corrotti) verso l'uscita seriale. Questa trasmissione però dura solo per il periodo di trasmissione dei dati di varianza (circa 700ns), ed ecco quindi il motivo dei soli 7 invii seriali ben precisi.

Esaminando il codice si evince che l'anomalia consiste in una errata valutazione della condizione che produce l'attivazione o meno dell'uscita dei dati seriali.

Ora, il modulo *acquire*, al completamento di una qualunque acquisizione, deve verificare come inviare i dati (se al modulo di varianza oppure per via seriale verso la BEE) facendo riferimento allo stato attivo di un segnale che inequivocabilmente indichi la tipologia della acquisizione appena effettuata.

Il segnale più idoneo a svolgere questo compito è il segnale "acq_valid_reg", che si asserta esclusivamente quando il segnale "acq_start" è rimasto attivo per un tempo prefissato, ed indica appunto, tramite l'uscita "acq_hold", che l'acquisizione scientifica è in corso.

Allora, invece di questo segnale, si è fatto riferimento per questa valutazione al segnale "acq_en_reg", che invece si asserta immediatamente all'arrivo del segnale "acq_start" e che comanda tramite l'uscita "acq_nbusy" l'armarsi del peak-detector.

La soluzione è semplice. Basta sostituire nel codice il segnale di riferimento come sopra indicato. In particolare nei righi:

```
data_buf_out_acq_reg <= '0' when
      (acq_enable = '0' or data_buf_cnt_out_rst = '1')
   else '1' when
      ((clk_out'event and clk_out = '1') and
      acq_en_reg = '1' and SM_DATA_BUF_CNT_OUT_EN = '1');

data_buf_out_auto_reg <= '0' when
      (acq_enable = '0' or data_buf_cnt_out_rst = '1')
   else '1' when
      ((clk_out'event and clk_out = '1') and
      acq_en_reg = '0' and SM_DATA_BUF_CNT_OUT_EN = '1');
```

è stato sostituito il segnale `acq_en_reg` con il segnale `acq_valid_reg`. Così, la suddetta sezione del codice diviene:

```
data_buf_out_acq_reg <= '0' when
      (acq_enable = '0' or data_buf_cnt_out_rst = '1')
   else '1' when
      ((clk_out'event and clk_out = '1') and
      acq_valid_reg = '1' and SM_DATA_BUF_CNT_OUT_EN = '1');

data_buf_out_auto_reg <= '0' when
      (acq_enable = '0' or data_buf_cnt_out_rst = '1')
   else '1' when
      ((clk_out'event and clk_out = '1') and
      acq_valid_reg = '0' and SM_DATA_BUF_CNT_OUT_EN = '1');
```

Dopo avere effettuato le suddette modifiche al codice VHDL del modulo *acquire*, il problema dello *shift* dei dati scientifici è stato risolto, come si può osservare in Fig. 59.

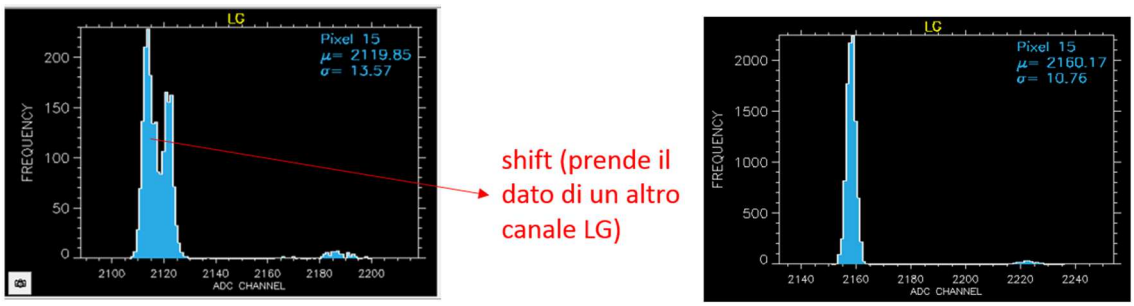


Fig. 59. Istogramma del 15-esimo pixel LG prima (a sinistra) e dopo (a destra) la modifica al codice attivando il campionamento di varianza.

4.13 Soluzione al problema iniezione di carica nel modulo *acquire* ()

Dai test effettuati sulle distribuzioni di carica degli impulsi dei SiPM è emersa una anomalia legata all'attivazione delle acquisizioni di varianza durante la lettura dei dati degli eventi scientifici. In particolare, le distribuzioni effettuate abilitando l'acquisizione di varianza nel modulo *acquire* presentano dei dati anomali.

Con l'acquisizione di varianza disabilitata, le distribuzioni della catena LG si presentano come degli istogrammi di frequenza con valori in ascissa compresi in un *range* di canali ADC compreso tra 2100 e 2170, come mostrato in Fig. 60.

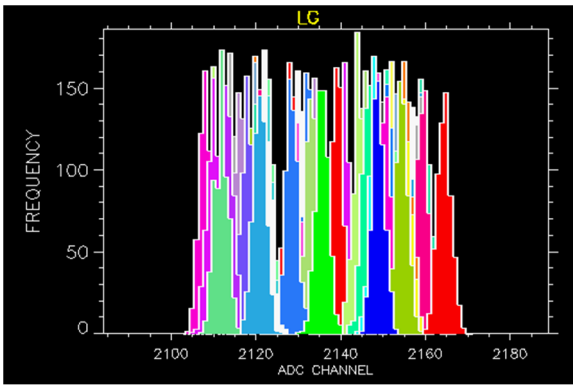


Fig. 60. Distribuzioni di carica relative alla catena LG disattivando l'acquisizione della varianza.

Tuttavia, attivando l'acquisizione della varianza nel modulo *acquire* si è notata una discrepanza rispetto ai valori attesi, evidenziata negli istogrammi di Fig. 61.

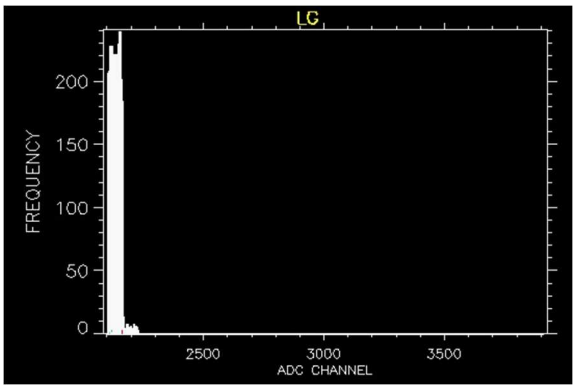


Fig. 61. Distribuzioni di carica relative alla catena LG attivando l'acquisizione della varianza.

La spiegazione del problema e la conseguente soluzione, descritte nelle righe seguenti, sono state suggerite dall'ideatore del codice (Francesco Russo, IASF Palermo).

È stato recentemente scoperto che l'ASIC CITIROC non "gradisce" l'arrivo del segnale PS_MODEB_EXT, che arma i *peak detector*, se il *mux* di uscita dei segnali analogici è rimasto selezionato su uno dei suoi 32 canali interni.

Questa intolleranza del CITIROC si manifesta con una strana anomalia sul valore del segnale di uscita del canale rimasto selezionato nel *mux*, conseguenza di una possibile iniezione di carica all'arrivo del PS_MODEB_EXT.

Questa "brutta" scoperta poteva in qualche modo compromettere pesantemente la modalità di acquisizione *mixed*, che prevede, in contemporanea, sia l'acquisizione dei dati scientifici che quelli per il calcolo della varianza, in quanto una interruzione asincrona di quest'ultima dovuta all'arrivo improvviso dello stop, creava per l'appunto quelle condizioni che il CITIROC non "digeriva".

Dunque, per correre ai ripari, è stata pensata e realizzata una modifica al modulo *acquire* che contemplasse la caratteristica appena scoperta, e che praticamente consiste nel resettare preventivamente detto *mux*, portando così la sua uscita in *tri-state*, immediatamente prima dell'arrivo del segnale PS_MODEB_EXT.

Il "costo" di tale operazione è un leggero *delay* aggiuntivo che si va a sommare a tutto il tempo di ritardo, per propagazione, tra il trigger in uscita dagli ASIC e il segnale di stop proveniente dalla BEE.

Essenzialmente, le modifiche effettuate al codice riguardano i seguenti punti:

- aggiunta di un *bus* a 3 bit in ingresso al modulo, per la programmazione del ritardo tra il reset del *mux* e l'attivazione del *peak detector*;
- aggiunta di un contatore per questo ritardo;
- integrazione di questo contatore con la logica già presente;
- aggiunta di un registro per il reset immediato del *mux* all'arrivo dell'*acq_start*;
- integrazione di questo registro con la logica già presente.

Di seguito le modifiche in dettaglio:

1) Aggiunto il seguente rigo nella sezione "port" della descrizione dell'*entity*:

```
acq_delay_cycles  : in  std_logic_vector (2 downto 0);
```

2) Aggiunti i seguenti righi nella sezione dichiarativa dell'architettura:

```
signal acq_delay_cnt_val     : integer range 0 to 7;
signal acq_delay_cnt         : integer range 0 to 7;
signal acq_delay_comb        : std_logic;
signal acq_delay_reg         : std_logic;
signal acq_mux_rst_reg       : std_logic;
signal SM_MUX_NRST_EN        : std_logic;
```

3) subito dopo il "begin" trovato e sostituito il rigo (commentato):

```
acq_nbusy <= not(acq_en_reg);
```

4) con il rigo:

```
acq_nbusy <= not(acq_delay_reg);
```

5) subito a seguire aggiunto il rigo:

```
mux_nrst <= not(acq_mux_rst_reg) and SM_MUX_NRST_EN;
```

6) adesso, sono stati trovati e sostituiti i righi (commentati):

```
acq_valid_cnt <= 0 when (acq_en_reg='0') else acq_valid_cnt+1 when
   ((clk_div_out'event and clk_div_out='1') and acq_valid_reg='0');
acq_valid_comb <= '1' when (acq_valid_cnt=acq_valid_cnt_val) else '0';
acq_valid_reg <= '0' when (acq_en_reg='0') else '1' when
   ((clk_div_out'event and clk_div_out='1') and acq_valid_comb='1');
acq_inhibit_reg <= '0' when (acq_en_reg='0'
   or (SM_ACQ_VALID_RST='1' and acq_hold_enable='0')) else '1' when
   ((clk_div_out'event and clk_div_out='1') and acq_valid_comb='1');
```

7) con questi righi:

```
acq_valid_cnt <= 0 when (acq_delay_reg='0') else acq_valid_cnt+1 when
   ((clk_div_out'event and clk_div_out='1') and acq_valid_reg='0');
acq_valid_comb <= '1' when (acq_valid_cnt=acq_valid_cnt_val) else '0';
acq_valid_reg <= '0' when (acq_delay_reg='0') else '1' when
   ((clk_div_out'event and clk_div_out='1') and acq_valid_comb='1');
acq_inhibit_reg <= '0' when (acq_delay_reg='0' or
   (SM_ACQ_VALID_RST='1' and acq_hold_enable='0')) else '1' when
   ((clk_div_out'event and clk_div_out='1') and acq_valid_comb='1');
```

8) e subito a seguire aggiunti:

```
acq_delay_cnt_val <= conv_integer(acq_delay_cycles);
acq_delay_cnt <= 0 when (acq_en_reg='0') else acq_delay_cnt+1 when
   ((clk_div_out'event and clk_div_out='1') and acq_valid_reg='0');
acq_delay_comb <= '1' when (acq_delay_cnt=acq_delay_cnt_val) else '0';
acq_delay_reg <= '0' when (acq_en_reg='0') else '1' when
   ((clk_div_out'event and clk_div_out='1') and acq_delay_comb='1');
acq_mux_rst_reg <= '0' when (acq_en_reg='0' or SM_ADC_CONV_EN='1')
   else '1' when ((clk_div_out'event and clk_div_out='1')
   and acq_valid_reg='0');
```

A questo punto, si è reso necessario modificare anche alcuni righi di codice relativi alla sezione della macchina a stati. In particolare:

9) è stato trovato e sostituito (commentato), in tutta la macchina a stati, il segnale di reset del *mux*:

```
mux_nrst
```

10) con il segnale:

```
SM_MUX_NRST_EN
```

In seguito alle precedenti variazioni sul codice, successivi test hanno evidenziato un altro inconveniente: ci si è accorti che continuavano a verificarsi valori anomali esclusivamente sulle distribuzioni dei canali 0 e 32. A tal riguardo, si è scoperto che l'iniezione di carica su tali canali avviene quando il segnale `acq_start` sale nel momento in cui sta iniziando un nuovo ciclo di acquisizione di varianza, e quindi il `mux_en` si attiva due volte consecutive, come illustrato in Fig. 62.

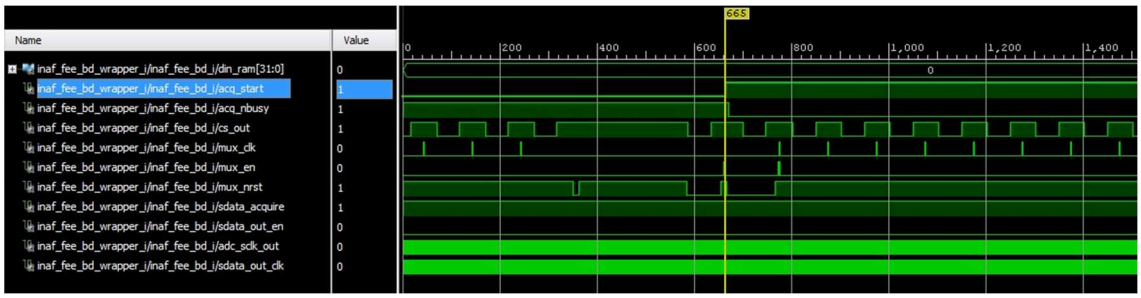


Fig. 62. Screenshot da ILA Chipscope relativi all'attivazione del segnale "mux_en".

La spiegazione del problema e la conseguente soluzione, descritte nelle righe seguenti, sono state suggerite dall'ideatore del codice (Francesco Russo, IASF Palermo).

Si rileva che quando il *mux*, a seguito dell'avvio della sequenza di acquisizione, passa dallo stato disabilitato (*tri-state*) allo stato attivo dell'uscita (selezionando, tramite il segnale di clock ad esso predisposto, il canale 0), questo *mux* risulta momentaneamente sordo ad un'eventuale ed improvviso segnale di reset che vorrebbe riportarlo immediatamente in *tri-state*.

È come se ci fosse una sorta di precedenza dell'attivazione rispetto alla disattivazione. In ogni caso sembra qualcosa legata a questi eventi.

In pratica, quando periodicamente parte l'acquisizione della varianza con il relativo avvio della scansione dei canali attraverso il *mux*, se in coincidenza di questo evento arriva anche un segnale di stop che attua contemporaneamente il reset dello stesso *mux*, allora quest'ultimo non si resetta affatto e di conseguenza il canale 0 viene colpito dalla famosa iniezione di carica. In Fig. 63 è mostrato l'effetto di questo problema:

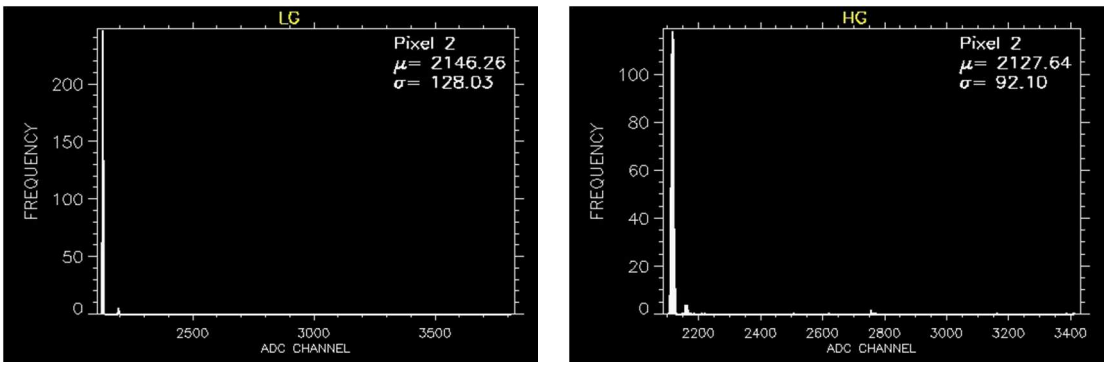


Fig. 63. Istogrammi di carica LG (a sinistra) ed HG (a destra) relativi al canale 0 (pixel 2).

La soluzione pensata per aggirare l'ostacolo è basata su una nuova modalità di gestione che prevede quindi che durante l'acquisizione della varianza avvenga soltanto la semplice e progressiva scansione dei 32 canali del CITIROC senza l'invio di alcun reset del *mux*. All'arrivo dello stop, prodotto dal trigger, si attua la procedura già sviluppata recentemente che prevede l'invio preventivo del segnale di reset e successivamente l'attivazione del *peak detector*. Questa volta, il reset non intercetterà mai l'attivazione del *mux*, che altrimenti produrrebbe l'anomalia, perché questa attivazione non avviene mai durante l'acquisizione della varianza.

Di seguito le modifiche da apportate al codice per risolvere il problema di iniezione di carica sul canale 0 di entrambi i CITIROC:

1) trovato e commentato il rigo nella sezione dichiarativa dell'architettura:

```
signal SM_MUX_NRST_EN   : std_logic;
```

2) subito a seguire, aggiunto il rigo:

```
signal SM_ACQ_MUX_RST   : std_logic;
```

3) trovato e commentato il rigo:

```
mux_nrst <= not(acq_mux_rst_reg) and SM_MUX_NRST_EN;
```

4) subito a seguire, aggiunto il rigo:

```
mux_nrst <= not(acq_mux_rst_reg);
```

5) trovato e commentato il rigo:

```
acq_mux_rst_reg <= '0' when (acq_en_reg='0' or SM_ADC_CONV_EN='1')
   else '1' when ((clk_div_out'event and clk_div_out='1')
   and acq_valid_reg='0');
```

6) subito a seguire, aggiunto il rigo:

```
acq_mux_rst_reg <= '0' when (acq_en_reg='0' or SM_ACQ_MUX_RST='1')
   else '1' when ((clk_div_out'event and clk_div_out='1')
   and acq_valid_reg='0');
```

7) trovati e commentati, in tutta la macchina a stati, i righi che contengono il segnale:

```
SM_MUX_NRST_EN
```

8) aggiunto nello stato `SM_CYC_INIT_1` il rigo:

```
SM_ACQ_MUX_RST<= acq_valid_reg;
```

9) aggiunto nello stato `SM_CYC_MUX_STEP_0` il rigo:

```
SM_ACQ_MUX_RST<= '0';
```

Vale la pena di rimarcare che, in effetti, il ritardo tra il reset del *mux* e l'attivazione del *peak detector* non è stato reso programmabile (per evitare di ritardare l'inizio dei test), ma sono stati invece provati diversi valori di ritardo così da ottenere il minimo valore utile a risolvere il problema in questione. Pertanto, al segnale `acq_delay_cnt_val` è stato assegnato un valore costante per le diverse verifiche.

In particolare per `acq_delay_cnt_val=5` il problema dell'iniezione di carica (attivando la varianza) risulta ancora presente anche se fortemente diminuito come è possibile osservare negli istogrammi di carica in Fig. 64, ottenuti triggerando su un pixel attivo. Nell'istogramma in blu mostrato dello *screenshot* inferiore della figura sono visibili i pic-chetti delle distribuzioni del primo e del secondo fotoelettrone.

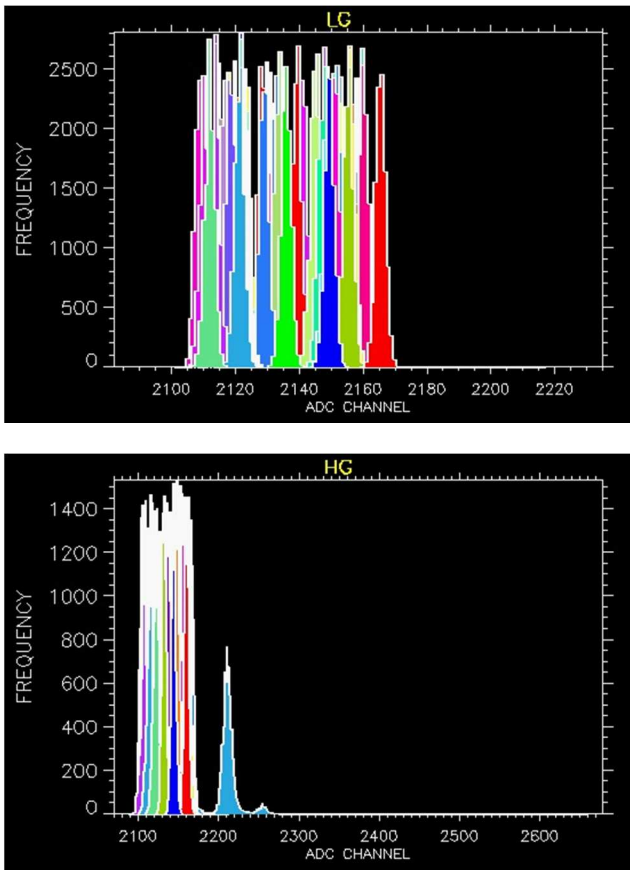


Fig. 64. Istogrammi di carica LG (sopra) ed HG (sotto) ottenuti per acq_delay_cnt_val=5.

Invece, aumentando leggermente il ritardo legato al quel segnale (misurato a step di 3.3ns), cioè ponendo `acq_delay_cnt_val=6`, il problema risulta praticamente risolto su tutti i canali, come è possibile riscontrare in Fig. 65.

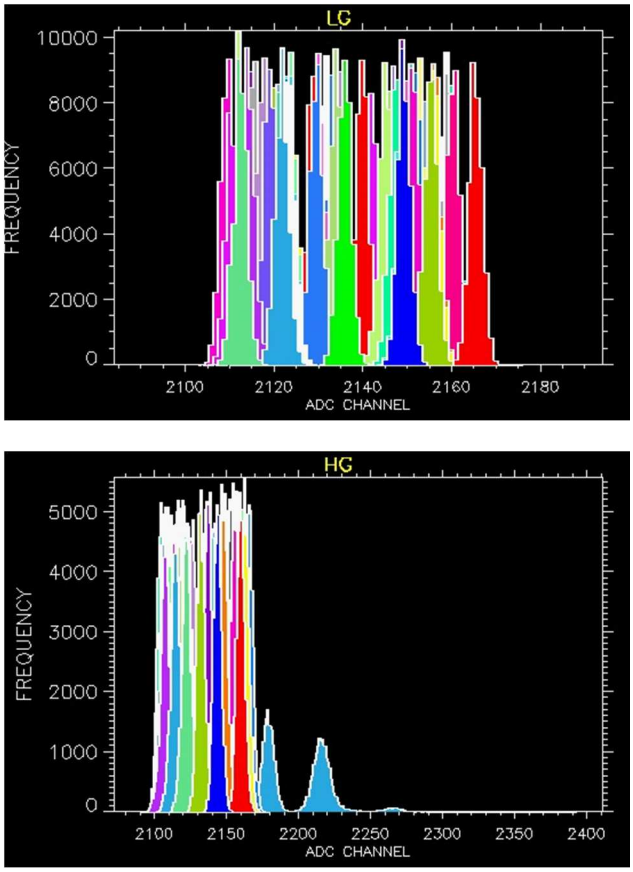


Fig. 65. Istogrammi di carica LG (sopra) ed HG (sotto) ottenuti per acq_delay_cnt_val=6.

Dunque, alla luce dei risultati ottenuti nei test, nella versione definitiva del codice VHDL del modulo *acquire*, il ritardo tra il reset del multiplexer degli ASIC e l'attivazione del rivelatore di picco è stato fissato al valore 6 (corrispondente a circa 20ns).

4.14 Aggiunta del protocollo UART nel SW per collegamento BEE ()

In questo aggiornamento è stato inserito, in ambiente SDK, del codice aggiuntivo per la gestione del protocollo UART per il collegamento alla BEE.

La risposta ai comandi mandati dal software di controllo nelle precedenti versioni del SW era costituita, fra le altre cose, anche da stringhe fornite mediante dei “printf”, che, per ridurre i tempi dei test e per rendere il codice pronto per la connessione con la BEE, sono state eliminate in questa nuova versione del codice. Infatti, nella versione aggiornata del SW, vengono spediti solamente i byte necessari per il protocollo di comunicazione BEE-FEE.

In pratica, mediante la procedura *send_bee_replay* vengono spediti 7 byte di comando (SOC, PDM_ID, LENGTH1, LENGTH2, CODE, ARG, ACK), successivamente vengono spediti i valori dei dati richiesti dal comando, ed infine si spediscono altri 3 byte di comando (CRC1, CRC2, EOC).

Il byte ACK (0x59=”Y”) indica alla BEE che il comando è stato accettato, mentre il byte NACK (0x4E=”N”) indica alla BEE che il comando non è stato riconosciuto.

Di seguito sono riportate le righe di codice relative al protocollo UART nel caso del comando REQUEST DATA (evento scientifico):

```
temp_send_bee_reply_2[0] = 0x0A;        // SOC
temp_send_bee_reply_2[1] = PDM->PdmId;  // PDM_ID
temp_send_bee_reply_2[2] = 0x01;        // LENGTH
temp_send_bee_reply_2[3] = 0x00;        // LENGTH
temp_send_bee_reply_2[4] = 10;          // CODE
temp_send_bee_reply_2[5] = 1;           // ARG
temp_send_bee_reply_2[6] = 0x59;        // ACK

send_bee_reply(&UartUsb_device, temp_send_bee_reply_2, 8);

BufCrc_r[0] = PDM->PdmId;
BufCrc_r[1] = 0x01;
BufCrc_r[2] = 0x00;
BufCrc_r[3] = 0x0A;
BufCrc_r[4] = 0x01;
BufCrc_r[5] = 0x59; //"Y"

for (idx = 0; idx < 64; idx++) { // Byte di dati
temp_send_bee_reply_2[0] = (PDM->acquire_buffer[idx] & 0xFF000000) >> 24;
temp_send_bee_reply_2[1] = (PDM->acquire_buffer[idx] & 0x00FF0000) >> 16;
temp_send_bee_reply_2[2] = (PDM->acquire_buffer[idx] & 0x0000FF00) >>  8;
temp_send_bee_reply_2[3] = (PDM->acquire_buffer[idx] & 0x000000FF);

BufCrc_r[5 + idx*4 + 1]     = (PDM->acquire_buffer[idx] & 0xFF000000) >> 24;
BufCrc_r[5 + idx*4 + 1 + 1] = (PDM->acquire_buffer[idx] & 0x00FF0000) >> 16;
BufCrc_r[5 + idx*4 + 1 + 2] = (PDM->acquire_buffer[idx] & 0x0000FF00) >>  8;
BufCrc_r[5 + idx*4 + 1 + 3] = (PDM->acquire_buffer[idx] & 0x000000FF);

send_bee_reply(&UartUsb_device, temp_send_bee_reply_2, 5);
}
```

```
BufCrc_r_l = 262;
crc_r  = (unsigned short) gen_crc16(BufCrc_r, BufCrc_r_l);
crc0_r = (unsigned char) ((crc_r & 0xFF00) >> 8);
crc1_r = (unsigned char) (crc_r & 0x00FF);

temp_send_bee_reply_2[0] = crc0_r;      // CRC
temp_send_bee_reply_2[1] = crc1_r;      // CRC

send_bee_reply(&UartUsb_device, temp_send_bee_reply_2, 3);

temp_send_bee_reply_2[0] = 0x0D;        // EOC

send_bee_reply(&UartUsb_device, temp_send_bee_reply_2, 2);
```

Nel caso in cui il comando non viene riconosciuto dalla FEE, viene inviato come risposta alla BEE il comando INIT con il byte NACK (0x4E):

```
temp_send_bee_reply_2[0] = 0x0A;              // SOC
temp_send_bee_reply_2[1] = PDM->PdmId;        // PDM_ID
temp_send_bee_reply_2[2] = 0;                 // LENGTH
temp_send_bee_reply_2[3] = 0;                 // LENGTH
temp_send_bee_reply_2[4] = 1;                 // CODE
temp_send_bee_reply_2[5] = PDM->PdmId;        // ARG
temp_send_bee_reply_2[6] = 0x4E;              //NACK "N"

BufCrc_r[0] = PDM->PdmId;
BufCrc_r[1] = 0x00;
BufCrc_r[2] = 0x00;
BufCrc_r[3] = 0x01;
BufCrc_r[4] = PDM->PdmId;
BufCrc_r[5] = 0x4E; //"N"
BufCrc_r_l = 6;

crc_r = (unsigned short) gen_crc16(BufCrc_r, BufCrc_r_l);
crc0_r = (unsigned char) ((crc_r & 0xFF00) >> 8);
crc1_r = (unsigned char) (crc_r & 0x00FF);
temp_send_bee_reply_2[7] = crc0_r;
temp_send_bee_reply_2[8] = crc1_r;
temp_send_bee_reply_2[9] = 0x0D;

send_bee_reply(&UartUsb_device, temp_send_bee_reply_2, 11);
```

4.15 Gestione del time-out su richiesta dati scientifici e di varianza ()

In base al funzionamento del modulo *mode_sel* precedentemente discusso, il comando di richiesta di dati scientifici abilita il *flip-flop* in Fig. 7 alla ricezione di un segnale di trigger di PDM proveniente dal modulo di generazione del trigger. Il campionamento di tale segnale produce in uscita il trigger “acq_start”, che dà l’avvio all’acquisizione dei dati scientifici da parte del modulo *acquire* e al successivo trasferimento verso la memoria. Qualora, invece, non dovesse verificarsi alcun evento di trigger che abiliti l’acquisizione dagli ADC, i dati scientifici non verranno prodotti e, di conseguenza, la memoria dedicata non verrà caricata. Questo determina una condizione di stallo nel software di gestione dovuta all’attesa dei dati sulla porta USB, che si traduce nella necessità di riavviare l’applicazione e la stessa scheda FPGA.

Per risolvere tale problema è stato inserito un timeout HW di 20s, superato il quale viene automaticamente sbloccata la macchina a stati che governa il trasferimento in memoria anche in assenza di dati prodotti. Nello stralcio di codice riportato di seguito, relativo al modulo *mode_sel*, è stato aggiunto un contatore a 20s comandato dal clock di sistema a 160MHz per forzare l’uscita dallo stato di attesa dei dati:

```
if sys_rstn = ‘0’ then
   ....
elsif rising_edge(sys_clk) then
   ....
   if int_state = x”0” then
   ....
   elsif int_state = x"1" then -- wait for the end of transmission
      if ((ram_we = x"F") and (ram_addr = "0001111111"))
      or (count = "10110010110100000101111000000000") then
         count <= (others =>'0');
         int_state <= x"2";
         ready_acq <= '1';
      else
         int_state <= x"1";
         count <= count + '1';
      end if;
   ....
   end if;
end if;
```

Analogamente, per quanto concerne il modulo *variance*, qualora la richiesta dei dati di varianza non dovesse produrre dei dati di uscita (in base alla modalità di funzionamento selezionata) si determina una condizione di blocco simile alla situazione precedente. In questo caso, il modulo coinvolto è quello relativo alla memorizzazione dei dati di varianza (*variance_output*). Pertanto, anche nel codice VHDL di tale blocco è stato introdotto un timeout di 20sec comandato dal clock di ingresso a 20MHz, così da sbloccare la macchina a stati in assenza di dati di varianza:

```
if rst = '1' then
   ....
elsif rising_edge(clk) then
```

```
    if int_state = x"0" then -- wait for data_out_ready pulse
       if (data_out_ready = '1')
       or (count = "10111110101111000010000000000") then
          int_state <= x"3";
          data_out_en <= '0';
          data_out_next <= '0';
          we <= x"0";
          din <= data_out;
       else
          count <= count + '1';
          we <= x"0";
          addr_int <= (others => '0');
          din <= (others => '0');
          data_out_en <= '0';
          data_out_next <= '0';
          int_state <= x"0";
       end if;
    elsif int_state = x"3" then
       data_out_en <= '1';
       we <= x"F";
       din <= data_out;
       int_state <= x"4";
    elsif int_state = x"4" then
       int_state <= x"1";
    elsif int_state = x"1"  then -- continue storing data
       if (cnt_out_word = x"80")
       or (count = "10111110101111000010000000000") then
          count <= (others =>'0');
          data_out_next <= '0';
          cnt_out_word <= x"00";
          data_out_en <= '0';
          req <= '1';
          we <= x"0";
          int_state <= x"2";
       else
          data_out_en <= '1';
          data_out_next <= '1';
          cnt_out_word <= cnt_out_word + x"01";
          int_state <= x"1";
       end if;
    ...
    end if;
end if;
```

In seguito alla modifica HW dei suddetti blocchi, il problema descritto è stato risolto in entrambe le situazioni di stallo.

5. COMANDI E TABELLE DI CONFIGURAZIONE AGGIORNATI

La filosofia adottata per l'invio e la ricezione delle informazioni da parte della PDM tende sostanzialmente a minimizzare il numero complessivo di comandi richiesti. Questo viene realizzato utilizzando delle tabelle di configurazione, sia per la programmazione degli ASIC che per la configurazione del sistema dell'FPGA e dei suoi moduli firmware. Ciò comporta che la variazione anche di uno soltanto dei parametri dell'FPGA o degli ASIC comporta l'invio di tutta la tabella di configurazione corrispondente.

Esistono in totale tre diverse tipologie di tabelle di configurazione: due tabelle per i due ASIC CITIROC (*configuration table* e *probe table*) e una tabella per l'FPGA (*FPGA table*), in modo tale che tutti i comandi previsti possano operare sulle diverse tipologie di tabelle, individuate come argomenti del comando. La lista dei possibili comandi è:

1) PDM INIT (*PDM ID*)
2) RESET MODULE (*acquire, variance, trig_cnt*)
3) CLOCK SELECT (*internal, external*)
4) START COUNT
5) SEND TABLE (*FPGA, configuration, probe*)
6) WRITE TABLE (*FPGA, configuration, probe*)
7) LOAD TABLE (*configuration, probe*)
8) EXEC TABLE (*FPGA, configuration, probe*)
9) REQUEST TABLE (*FPGA, configuration, probe*)
10) REQUEST DATA (*acquire, variance, trig_cnt, HKs*)
11) SET SWITCH

Ogni commando è composto da un certo numero di byte in funzione della tipologia di commando e di argomento, nell'ordine:

- 1 byte di sincronismo
- 1 byte di indirizzamento (identificativo PDM = da 1 a 37, 0=broadcast)
- 2 byte di length (lunghezza dei dati in byte)
- 1 byte di codice comando
- 1 byte di argomento
- 2**n* byte di dati (solo per i comandi che includono dati di una tabella)
- 2 byte di CRC

Nel codice SDK, la lettura dei campi `CliCmdCode` e `CliCmdArg`, relativi rispettivamente ai codici del comando e del relativo argomento impostati dal software di controllo, avviene nella *command line interface* (modulo cli.c), e tale coppia di parametri viene successivamente tradotta in un singolo comando tramite la funzione **`encodeCmdCodeArg`** (modulo global.c):

```
unsigned short encodeCmdCodeArg(Cli_struct *Cli){
unsigned char CmdCode = Cli->CliCmdCode;
unsigned char CmdArg  = Cli->CliCmdArg;

if (CmdCode == 1) {return 1;}                    // PDM INIT
```

```
if ((CmdCode == 2) & (CmdArg == 1)) {return 2;}   // RESET MODULE ACQUIRE
if ((CmdCode == 2) & (CmdArg == 2)) {return 3;}   // RESET MODULE VARIANCE
if ((CmdCode == 2) & (CmdArg == 3)) {return 4;}   // RESET MODULE TRIG_CNT
if ((CmdCode == 3) & (CmdArg == 0)) {return 5;}   // CLOCK SELECT INTERNAL
if ((CmdCode == 3) & (CmdArg == 1)) {return 6;}   // CLOCK SELECT EXTERNAL
if ((CmdCode == 4) & (CmdArg == 0)) {return 7;}   // START COUNT
if ((CmdCode == 5) & (CmdArg == 1)) {return 8;}   // SEND TABLE FPGA
if ((CmdCode == 5) & (CmdArg == 2)) {return 9;}   // SEND TABLE ASIC
if ((CmdCode == 5) & (CmdArg == 3)) {return 10;}  // SEND TABLE PROBE ASIC
if ((CmdCode == 6) & (CmdArg == 1)) {return 11;}  // WRITE TABLE FPGA
if ((CmdCode == 6) & (CmdArg == 2)) {return 12;}  // WRITE TABLE ASIC
if ((CmdCode == 6) & (CmdArg == 3)) {return 13;}  // WRITE TABLE PROBE ASIC
if ((CmdCode == 7) & (CmdArg == 2)) {return 14;}  // LOAD TABLE FPGA
if ((CmdCode == 7) & (CmdArg == 3)) {return 15;}  // LOAD TABLE ASIC
if ((CmdCode == 8) & (CmdArg == 1)) {return 16;}  // LOAD TABLE PROBE ASIC
if ((CmdCode == 8) & (CmdArg == 2)) {return 17;}  // EXECUTE TABLE ASIC
if ((CmdCode == 8) & (CmdArg == 3)) {return 18;}  // EXECUTE TABLE PROBE ASIC
if ((CmdCode == 9) & (CmdArg == 1)) {return 19;}  // REQUEST TABLE FPGA
if ((CmdCode == 9) & (CmdArg == 2)) {return 20;}  // REQUEST TABLE ASIC
if ((CmdCode == 9) & (CmdArg == 3)) {return 21;}  // REQUEST TABLE PROBE ASIC
if ((CmdCode == 10) & (CmdArg == 1)) {return 22;} // REQUEST DATA ACQUIRE
if ((CmdCode == 10) & (CmdArg == 2)) {return 23;} // REQUEST DATA VARIANCE
if ((CmdCode == 10) & (CmdArg == 3)) {return 24;} // REQUEST DATA TRIG_CNT
if ((CmdCode == 10) & (CmdArg == 4)) {return 25;} // REQUEST DATA HK
if ((CmdCode == 11) & (CmdArg == 0)) {return 26;} // SET SWITCH
}
```

L'esecuzione dei comandi avviene tramite la procedura (modulo helloworld.c):

```
int main()
{
...
valCmd = 0;
/*
 * Execute Command
 */
      cmd = encodeCmdCodeArg(Cli);
      switch(cmd){
      case 1:{...}  // PDM INIT
      case 2:{...}  // RESET MODULE ACQUIRE
      ...
      case 26:{...} // SET SWITCH
      }
...
}
```

Il comando REQUEST DATA richiede all'FPGA di leggere i dati richiesti (*acquire, variance, trig_cnt, HK*) dallo spazio di memoria corrispondente e di spedirli alla BEE tramite UART o SPI). I blocchi dati relativi ad un evento scientifico, all'accumulazione della varianza, ai contatori di trigger e agli *housekeeping* sono riassunti di seguito:

Modulo	*Blocco dati*	*Byte totali*
acquire	64 HG x 16 bit 64 LG x 16 bit	256 byte
	128 x 16 bit 128 x 2 byte	
variance	64 HG somme x 32 bit 64 HG quadrati x 32 bit	512 byte
	128 x 32 bit 128 x 4 byte	
trig_cnt	64 pixel x 32 bit	256 byte
	64 x 32 bit 64 x 4 byte	
HK	10 temp. sensori (PDM board) x 16 bit 3 corr. + 1 temp. (ASIC board) x 16 bit 1 corr. + 1 temp. (FPGA board) x 16 bit 3 spare x 16 bit	38 byte
	(16 + 3) x 16 bit 19 x 2 byte	

Gli spazi di memoria allocati per questi blocchi dati sono definiti all'interno della struttura seguente (modulo SW global.h su SDK):

```
typedef struct
{
...
      u32 trgcnt_buffer[64];    // 64*32 word32  = 128 word16 = 256 byte
      u32 acquire_buffer[64];   // 64*32 word32  = 128 word16 = 256 byte
      u32 variance_buffer[128]; // 128*32 word32 = 256 word16 = 512 byte
      u16 hk_buffer[19];        //               = 19 word16  = 38 byte
...
} Pdm_struct;
```

Nelle sezioni seguenti viene riportato l'elenco delle tabelle di configurazione aggiornate in base alle varie modifiche effettuate nel firmware e descritte nei paragrafi precedenti. L'elenco comprende le seguenti quattro tabelle:

- Tabella di configurazione dell'FPGA
- Tabella di configurazione degli SWITCH
- Tabella di configurazione degli ASIC
- Tabella di configurazione delle PROBE degli ASIC

I parametri aggiuntivi rispetto all'architettura della versione iniziale sono stati evidenziati all'interno di ciascuna tabella.

5.1 Tabella di configurazione dell'FPGA

La tavola sottostante elenca i parametri relativi alla tabella di configurazione dell'FPGA.

Le righe della tabella evidenziate in grigio rappresentano i parametri aggiuntivi rispetto all'architettura della versione iniziale.

Ampiezza della tabella = 40 byte (320 bit)

parameter	*module*	*start byte*	*start bit*	*n. bit*	*description*
	uBlaze	0			
PDM ID		0	0	8	1-37 identificativo PDM
HKs_en		1	0	8	0=disable,1=enable HK data
ASIC_table_verify		2	0	8	0=disable 1=verify table
stand alone		3	0	8	0=stand-alone, 1=BEE
		4+			3*spare bytes
rstb_PSC_1-2		7	0	1	signal set ("0" / "1")
PS_global_trig_1-2		7	1	1	signal set ("0" / "1")
Raz_p_1-2		7	2	1	signal set ("0" / "1")
Val_Evt_p_1-2		7	3	1	signal set, 0=trig_BE, 1=trigger ASIC
RESET_PA		7	4	1	signal set ("0" / "1")
	trig_gen	8			
majority_in	write	8	0	3	**n ≥ n adjacent pixels**
filter_out_en	write	9	0	1	0=disable,1=enable
		10+			2*spare bytes
	trig_count	12			
time_window	write	12	0	3	**time window = [2^n]*0.125s**
		13+			3*spare bytes
	acquire_ctrl	16			
acq_valid_cycles	write	16	0	8	0=0ns,255=850ns@300MHz
acq_mux_cycles	write	17	0	8	0=0ns,255=2550ns@100MHz
acq_conv_cycles	write	18	0	8	0=0ns,255=2550ns@100MHz
var_delay_cycles	write	19	0	8	0=0ns,255=2550ns@100MHz

parameter	*module*	*start byte*	*start bit*	*n. bit*	*description*
acq_hold_en	write	20	0	1	**0=default**,1=enable (single shot)
		21+			3*spare bytes
	variance	24			
acq_cycles	write	24	0	16	n samples to accumulate
data_offs	write	26	0	16	offset to be subtract to raw data
		28+			2*spare bytes
	trigger_mask	30			
reg2_mask	write	30	0	32	mask channel CITIROC 0-31
reg1_mask	write	34	0	32	mask channel CITIROC 32-63
total		38		304	
totali + 8byte inclusi nel comando		46			46*(8+1+1)bit=460→2.3**ms**@ 200Kbps USB baud (115200)→**3.3 ms**

La definizione del vettore contenente i parametri della suddetta tabella è realizzato nel codice dalla struttura seguente (modulo SW global.h su SDK):

```
typedef struct
{
...
      unsigned char FpgaTable[40];
...
} Cli_struct;
```

5.2 Tabella di configurazione degli Switch

La tavola sottostante elenca i parametri della tabella di configurazione degli SWITCH.

Le righe della tabella evidenziate in grigio rappresentano i parametri aggiuntivi rispetto all'architettura della versione iniziale.

Ampiezza della tabella = 2 byte (16 bit)

signal	*module*	*start byte*	*start bit*	*n. bit*	*description*
	acquire_ctrl	0			
acq_en	write	0	0	1	0=disable,1=enable acquisition
acq_var_en	write		1	1	0=disable,1=enable sampling
			2	2	spare
	trig_gen	0			
trig_sel	write		4	2	1=nor32, 2=trig_gen, 0/3=disable
			6	2	spare
	variance	1			
acq_enable	write	1	0	1	0=dis, 1=enable var data accum.
			1	1	1*spare bit
	monostable	1			
wind_mon	write	1	2	4	**wind_mon = n(bit)*3.3ns**
total		2		13	

La definizione del vettore contenente i parametri della suddetta tabella è realizzato nel codice dalla struttura seguente (modulo SW global.h su SDK):

```
typedef struct
{
...
      unsigned char SwitchTable[2];
...
} Cli_struct;
```

5.3 Tabella di configurazione degli ASIC

La tavola sottostante elenca i parametri della tabella di configurazione degli ASIC.

Ampiezza della tabella = 286 byte (2288 bit)

module	*start byte*	*start bit*	*n. bit*		*default*
channel 0 to 31 4-bit_t	0	0	128	4bit trigger threshold_t	32x"0000"
channel 0 to 31 4-bit		128	128	4bit trigger threshold	32x"0000"
discriminator parameters		256	9	9x1-bit parameters	
discriminator mask		265	32	allow to mask discri_t	32x"0" mask active
LG/HG en/dis parameters		297	34	28x1-bit + 2x3-bit	
input 8-bit DAC		331	288	(8+1)bit x 32 = 288bit	
Pre-amplifier configuration (ch0-ch31)		619	480	(6+6+1+1+1) = 15 bit/ch gain-h gain-l s s s	
enable/disable parameters		1099	8	8x1-bit parameters	
DAC1 threshold	138 +3	1107	10	trigger threshold_t (?)	"01 0000 0000"
DAC2 threshold	139 +5	1117	10	trigger threshold	"01 0000 0000"
enable/disable parameters	140 +7	1127	17	17x1-bit parameters	
		1144		bit totali x un ASIC	
totali x 2 ASIC	286		2288	per 2 ASIC	
totali + 8byte inclusi nel comando	294			2940 bit → **14.7 ms** portaUSB → **25.5 ms**	@200000 baud @115200 baud

La definizione del vettore contenente i parametri della suddetta tabella è realizzato nel codice dalla struttura seguente (modulo SW global.h su SDK):

```
typedef struct
{
...
      unsigned char AsicTable[286];
...
} Cli_struct;
```

5.4 Tabella di configurazione delle PROBE degli ASIC

La tavola sottostante riassume, infine, i parametri relativi alla tabella di configurazione delle PROBE dei due ASIC.

Ampiezza della tabella = 56 byte (448 bit)

module	*start byte*	*start bit*	*n. bit*		*signal output*
Out_fs	0	0	32	from channel 0 to 31	analog
Out_ssh_LG		32	32	from channel 0 to 31	analog
PeakSensing_modeb_LG		64	32	from channel 0 to 31	digital
Out_ssh_HG		96	32	from channel 0 to 31	analog
PeakSensing_modeb_HG		128	32	from channel 0 to 31	digital
Out_PA_HG/LG		160	64	from channel 0 to 31	analog
	28	224		bit totali x un ASIC	
totali	56		448	per 2 ASIC	
totali + 8byte inclusi nel comando	64			640 bit → **3.2 ms** porta-USB → **5.5 ms**	@200000 baud @115200 baud

La definizione del vettore contenente i parametri della suddetta tabella è realizzato nel codice dalla struttura seguente (modulo SW global.h su SDK):

```
typedef struct
{
...
      unsigned char ProbeAsicTable[56];
...
} Cli_struct;
```

6. CONTACTS

Le risorse umane INAF che operano sul design della camera ASTRI sono composte dal team COLD dell'Osservatorio Astrofisico di Catania e dal team dell'Istituto di Astrofisica Spaziale e Fisica Cosmica di Palermo. L'unione delle suddette risorse viene comunemente indicato con il termine "Electronics Camera Team".

Di seguito vengono riportati i rispettivi recapiti.

Giovanni Bonanno	gbo@oact.inaf.it	OACT Catania
Salvatore Garozzo	salvatore.garozzo@oact.inaf.it	OACT Catania
Davide Marano	davide.marano@oact.inaf.it	OACT Catania
Alessandro Grillo	agrillo@oact.inaf.it	OACT Catania
Giuseppe Romeo	giuseppe.romeo@oact.inaf.it	OACT Catania
Maria Cristina Timpanaro	mctimpanaro@oact.inaf.it	OACT Catania
Osvaldo Catalano	catalano@ifc.inaf.it	IFC Palermo
Carmelo Gargano	gargano@ifc.inaf.it	IFC Palermo
Salvatore Giarrusso	jerry@ifc.inaf.it	IFC Palermo
Domenico Impiombato	domenico.impiombato@ifc.inaf.it	IFC Palermo
Giovanni La Rosa	larosa@ifc.inaf.it	IFC Palermo
Francesco Russo	russo@ifc.inaf.it	IFC Palermo
Pierluca Sangiorgi	sangiorgi@ifc.inaf.it	IFC Palermo
Giuseppe Sottile	sottile@ifc.inaf.it	IFC Palermo