UNIVERSITA' DEGLI STUDI DI FIRENZE

Facoltà di Scienze Matematiche Fisiche e Naturali

Corso di Laurea in Informatica

Tesi di Laurea

Realizzazione di un sistema di trigger per esperimenti di fisica nucleare mediante dispositivi logici programmabili

Relatore: Prof. Gabriele Pasquali

Correlatore: Prof. Pierluigi Crescenzi

Candidato: Giordano Bruno Marzocco

Anno Accademico 2008-2009

INTRODUZIONE

Il lavoro descritto in questa tesi è consistito nella realizzazione di un *sistema di trigger* da impiegare in un esperimento di fisica nucleare. In tali esperimenti si studiano eventi di collisione fra nuclei atomici: il sistema di trigger ha il compito di segnalare al *sistema di acquisizione dati* che si sono verificate le condizioni per l'acquisizione di un "evento" – definito da un particolare insieme di segnali logici in coincidenza temporale – e che si può procedere alla lettura delle informazioni fornite dai rivelatori di particelle. In passato, il sistema di trigger era realizzato sfruttando moduli elettronici discreti, ciascuno dedicato a una o più operazioni elementari (porte logiche, contatori, registri). Il notevole sviluppo della tecnologia FPGA (*Field Programmable Gate Array*) permette oggi di trasferire l'intero sistema di trigger su un unico circuito integrato programmabile.
La programmazione avviene tramite un HDL (*Hardware Description Language*) simile a un normale linguaggio di programmazione. Il programma infatti gestisce tutti gli aspetti legati alla generazione del segnale di trigger, come la possibilità di interrompere il flusso di segnali in entrata (tramite una logica di veto), la possibilità del downscaling, cioè l'applicazione alla frequenza del segnale proveniente dal rivelatore di un preciso fattore di riduzione, etc.
Durante lo sviluppo di un progetto, programmato in un HDL, è anche possibile simulare, con appositi software, il comportamento del sistema. I vantaggi di una realizzazione su FPGA, in termini di tempi di sviluppo, complessità programmabilità e versatilità del sistema sono, quindi, notevoli rispetto a una realizzazione a componenti discreti.
Il sistema di trigger realizzato in questo lavoro è pensato per essere installato su una scheda a standard VME (un bus industriale sviluppato dalla Motorola [1]) realizzata dalla CAEN di Viareggio [2]. E' possibile gestire fino a 32 segnali logici di ingresso, provenienti dai rivelatori. Tali segnali vengono utilizzati per assegnare un vettore di variabili logiche (le operazioni effettuate sono programmabili dall'utente) che costituiscono una "istantanea" della situazione del sistema di rivelazione e che vengono poi impiegate per decidere se generare il segnale di trigger.
L'ambiente di sviluppo utilizzato in questo lavoro è il software QUARTUS II prodotto dall'ALTERA, la stessa ditta che produce le FPGA installate sulla nostra scheda. Nell'ambito di Quartus, le funzioni logiche che la FPGA dovrà realizzare possono essere descritte in

termini di porte logiche, registri e altri elementi logici fra loro interconnessi (come in un foglio schematico di un circuito) oppure possono essere descritte in un HDL. In questo lavoro è sfruttata la seconda possibilità, descrivendo il sistema in linguaggio VHDL (VHSIC Hardware Description Language). QUARTUS II permette all'utente di compilare i propri programmi, eseguire analisi temporizzate, simulare la reazione della descrizione logica rispetto a certi segnali detti "stimoli", ed infine, trasferire la programmazione alla FPGA.
Il capitolo 1 descrive brevemente un tipico apparato per esperimenti di fisica nucleare, e spiega la funzione del sistema di trigger di un esperimento di questo tipo.
Il capitolo 2 presenta la famiglia dei dispositivi logici programmabili (FPGA) e cioè i dispositivi usati in questo lavoro, e la loro programmazione tramite linguaggi descrittivi come il VHDL. Vengono inoltre illustrate le principali caratteristiche del VHDL.
Il capitolo 3 analizza nel dettaglio la trigger box realizzata in questo lavoro. Modulo per modulo, viene descritta la funzione svolta e riportato il risultato delle simulazioni del comportamento del modulo stesso.

Capitolo 1

Apparati per esperimenti di fisica nucleare

1.1 I rivelatori, gli “occhi” dell’esperimento

Un possibile esperimento di fisica nucleare consiste nell’inviare un fascio di ioni accelerato contro un bersaglio. Il bersaglio è in genere costituito da un foglio molto sottile (circa 1 μm) del materiale oggetto di studio. Nello scontro i nuclei del fascio e quelli del bersaglio si spezzano dando luogo a frammenti più leggeri, la cui massa varia da quella degli isotopi dell’idrogeno o dell’elio a quella di nuclei più pesanti ad esempio carbonio, azoto, neon. I prodotti della collisione, radiazione e particelle dello stesso tipo o diverse da quelle incidenti, attraversano opportuni dispositivi, i rivelatori, producendo un segnale elettrico. Tale segnale contiene informazioni sul tipo di particella, la sua energia, l’istante di arrivo etc. Queste informazioni devono essere estratte dal segnale ai fini di una ricostruzione complessiva dell’evento di collisione. La grande quantità di dati da trattare, la loro elaborazione e raccolta richiedono lo sviluppo di tecnologie sempre più avanzate con successive ricadute nei campi più disparati: dall’elettronica alle tecniche di alto vuoto, dalla diagnostica e terapia medica, all’informatica.
In figura è riprodotto schematicamente un tipico apparato per esperimenti di fisica nucleare: intorno al bersaglio, in cui avviene la collisione, sono disposti i rivelatori di particelle

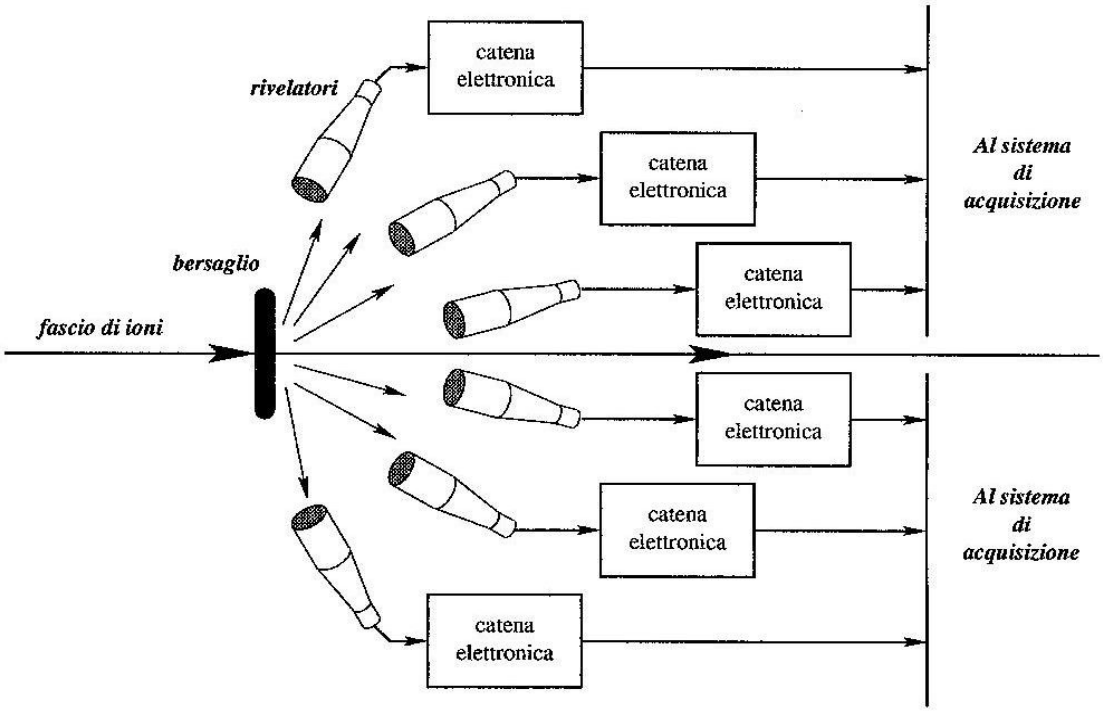


Fig 1.1 Sistema di rivelazione di un esperimento

Come già accennato, i frammenti che colpiscono i rivelatori producono in esso una risposta che, a seconda del tipo di rivelatore considerato, può essere un segnale elettrico o un segnale luminoso (*luce di scintillazione*) [3]. Molti tipi di rivelatori sfruttano la creazione di coppie dovuta al passaggio della radiazione, per esempio coppie elettrone-ione in un *rivelatore a gas* o coppie elettrone-lacuna in un *rivelatore a semiconduttore*.
La caratteristica fondamentale di un rivelatore di particelle è la sua sensibilità, cioè la capacità di produrre in uscita un segnale utilizzabile per una dato tipo di particella e per un dato intervallo di energia.
Nel caso della luce di scintillazione il segnale luminoso è convertito in un segnale elettrico grazie a dispositivi come *fotodiodi, o fotomoltiplicatori* [3].
La carica elettrica prodotta dal passaggio della particella è legata, spesso con una semplice legge di proporzionalità, all'energia depositata dalla particella nel rivelatore, mentre il fronte di salita del segnale è impiegato per determinare l'istante d'arrivo della particella e quindi (nota la distanza dal bersaglio e l'istante in cui è stata prodotta) la sua velocità [3].

La catena elettronica associata a ciascun rivelatore ha il compito di estrarre dal segnale le informazioni fisiche interessanti, ottimizzando anche il rapporto segnale/rumore.

1.2 Elettronica di Trigger

Un rivelatore colpito da una particella comunica di solito il proprio stato al sistema di trigger dell'esperimento, mediante un segnale logico (un bit) che diventa "vero".
Lo stato dei segnali logici dei rivelatori, ad un certo istante, costituisce una "fotografia" della situazione dell'apparato sperimentale. Allo scopo di ridurre l'informazione da trattare, e sulla base del fenomeno fisico che si vuole studiare, tali segnali vengono combinati di solito mediante semplici operazioni logiche, per ottenere una serie di *subtrigger*. Ad esempio, i segnali logici di tutti i rivelatori di un certo tipo potranno essere raccolti in un unico "OR" se non si è interessati a quale di essi esattamente sia stato colpito. L'OR dei segnali sarà poi impiegato per la logica di trigger dell'esperimento
Dal punto di vista del sistema di acquisizione, un insieme opportuno di subtrigger (o di variabili logiche ottenute combinando i subtrigger) che diventano veri entro un dato intervallo di tempo (detto *tempo risolutivo*) costituisce un *evento,* ovvero una situazione dell'apparato candidata per una possibile acquisizione. In corrispondenza di un evento il sistema di trigger dell'esperimento produce un segnale chiamato *trigger*. A seconda degli scopi prefissati per l'esperimento, si costruiranno opportuni trigger che indicheranno quanti e quali eventi acquisire. In sistemi come quello trattato in questa tesi, i segnali di trigger sono poi inviati a una porta OR che produce il cosiddetto *Main-trigger* dell'esperimento. Al verificarsi di un evento, il main-trigger abiliterà i circuiti che convertono le informazioni analogiche in valori digitali (i cosiddetti ADC, *Analog to Digital Converter*) nonchè i circuiti (CPU, etc.) predisposti all' acquisizione dati [4]. I dati raccolti sono memorizzati (su disco o nastro magnetico) per le successive analisi offline.
I segnali logici prodotti dai rivelatori costituiscono gli ingressi della trigger box realizzata in questa tesi.

CAPITOLO 2

Dispositivi logici programmabili e linguaggio VHDL

2.1 Dispositivi logici programmabili

I componenti *FPGA* (*Field Programmable Gate Array*) sono dispositivi logici programmabili la cui diffusione ha avuto inizio nella seconda metà degli anni ’70 come terza generazione di *PLD* (*Programmable Logic Device*).
Un PLD è un circuito ad elevata scala di integrazione (*VLSI, Very Large Scale Integration*) che può essere opportunamente programmato o personalizzato dall'utente finale, in modo da realizzare una specifica funzione [5].

I vantaggi legati all'utilizzo di PLD sono molteplici:

- Un singolo PLD può sostituire numerosi circuiti a bassa o media scala di integrazione, con significativi miglioramenti in termini di area occupata sul circuito stampato, di affidabilità e di costi.

- Le interconnessioni a livello di circuito stampato vengono sostituite da collegamenti all'interno di un singolo circuito integrato. La riduzione delle capacità parassite consente di migliorare sia i tempi di propagazione che la potenza dissipata.

- Un sistema basato su PLD è molto più flessibile rispetto ad uno realizzato con componenti logici discreti. Molti PLD sono infatti riprogrammabili elettricamente anche dopo essere stati montati su di un circuito stampato (*in-system programmability*). Ciò consente di modificare la funzionalità di un sistema digitale senza dover aggiungere o rimuovere componenti.

- Dopo aver programmato un PLD è possibile impedire che la configurazione interna del dispositivo sia leggibile dall'esterno, proteggendo in questo modo la riservatezza del progetto.

L'utilizzo dei componenti *PLD* risale alla seconda metà degli anni '60, con l'introduzione di una "*matrice a diodi configurabile*". Il dispositivo fu sviluppato e realizzato presso i laboratori della *Harris Semiconductor*, conosciuta allora come *Radiation Inc* [5]. Il dispositivo consentiva di realizzare semplici funzioni logiche bruciando le connessioni tra i diodi all'interno della matrice, tramite il passaggio di una corrente elevata attraverso le connessioni stesse. La programmazione della matrice era eseguita dalla *Harris*, in quanto non erano disponibili in commercio dispositivi per la programmazione.
Storicamente i primi circuiti veramente programmabili dall'utente apparsi sul mercato furono le *PROM* (*Programmable Read Only Memory)*, ossia le memorie programmabili a sola lettura. Lo scopo di tali circuiti é semplice: permettere al progettista/costruttore di sistemi digitali di configurarsi rapidamente e a basso costo le ROM di cui può avere bisogno. Prima dell'avvento delle PROM l'unica possibilità era quella di rivolgersi ad un fabbricante di circuiti integrati, specificando il contenuto della ROM e chiedere la preparazione del circuito integrato con le specifiche richieste.
Le PROM sono invece memorie a sola lettura in cui una semplice procedura permette di alterare stabilmente le connessioni interne al fine di memorizzare i dati voluti.

In fig. 2.1 è riportata una memoria PROM. Essa é costituita da 2 parti: una matrice di NAND che ha lo scopo di decodificare l'indirizzo ed una matrice di OR realizzata a diodi che provvede a trasferire in output i bit corrispondenti dell'indirizzo selezionato. E' questa seconda matrice di OR che deve essere modificata per alterare i dati memorizzati.

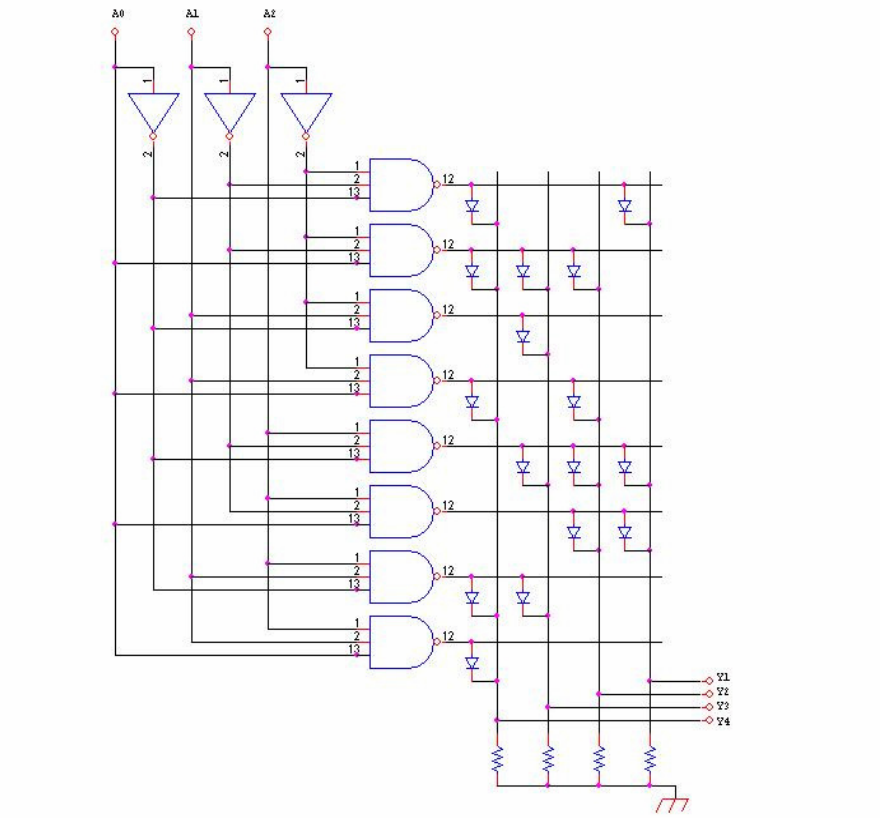


Fig 2.1 Schema di una memoria PROM

I diodi, posti fra le linee di uscita, rappresentano dei "fusibili" ossia delle connessioni molto fragili che possono essere distrutte selettivamente dal passaggio di una corrente sufficientemente elevata. Solo le linee in cui il diodo è lasciato integro (cioè i diodi mostrati in figura) andranno a un livello di tensione più alto (di solito associato a un bit ’1’) quando il corrispondente NAND é alto.

Una nuova famiglia di componenti programmabili che vanno sotto vari nomi commerciali (PAL, GAL, PLA, LCA, FPGA, ecc) era caratterizzata dall’avere anche la matrice di AND programmabile. Nonostante la maggior versatilità, anche con questo tipo di architettura era possibile costruire solo funzioni combinatorie degli ingressi e dello stato interno. Questo limite fu superato dalle PAL "versatili", in cui furono introdotti dei dispositivi *flip-flop*, rendendo il circuito sequenziale: lo stato dell’uscita non dipende solo dallo stato dell’ingresso a un dato istante ma anche dalla “storia” precedente [6].

I dispositivi PAL e PLA e i loro derivati vengono comunemente raccolti nella categoria dei SPLD (*Simple Programmable Logic Device*). Grazie

allo sviluppo della tecnologia *CMOS* la loro complessità è aumentata nel corso degli anni. Come nelle PROM, questa crescita è stata limitata, sebbene in misura minore, dall'aumento esponenziale della dimensione della matrice di porte OR in seguito all'inserimento di nuovi ingressi. I costruttori hanno perciò deciso di fornire chip di maggiore capacità, combinando insieme più dispositivi SPLD e connettendoli con unità di collegamento programmabili. Sono nati così i CPLD (*Complex Programmable Logic Device*). L'idea è quella di aumentare la capacità di un dispositivo logico programmabile integrando più blocchi logici, ognuno simile ad una PAL, su di uno stesso chip. I vari blocchi logici sono collegati fra loro mediante opportune interconnessioni programmabili, come mostra lo schema semplificato di Figura 2.3:

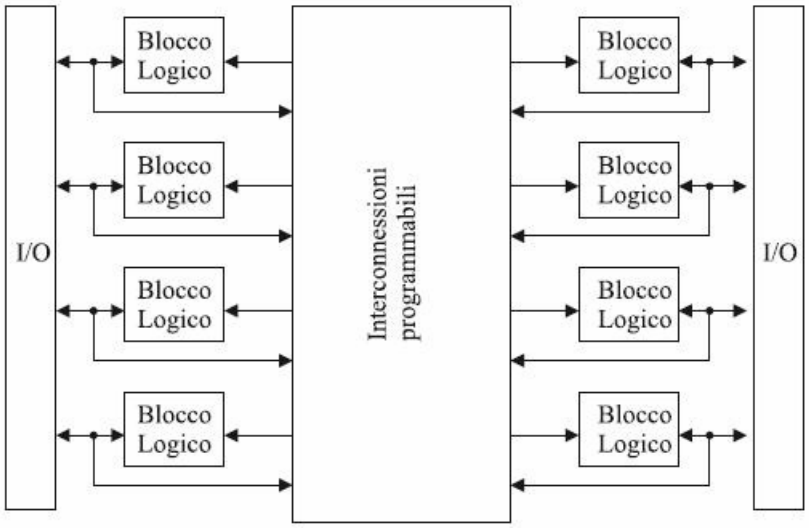


Fig 2.3 Schema a blocchi di una CPLD

Pur presentando con essi numerose analogie, i dispositivi FPGA differiscono dai CPLD in quanto i singoli blocchi logici non sono di tipo SPLD. Si tratta, come vedremo in seguito, di celle di memoria contenenti vettori di bit, dette LUT (*Look Up Table*). Sia i blocchi logici che le loro interconnessioni sono programmabili dall'utente.
Una FPGA è costituita da una matrice di blocchi logici (*L-Block*), [5] in cui vengono implementate le funzioni logiche, interconnessi tra di loro mediante risorse programmabili. In Figura 2.4 è mostrato un esempio, di architettura di FPGA: i blocchi funzionali (indicati con L-Block) sono connessi tra loro mediante "canali" verticali e orizzontali, all'interno dei quali corrono piste parallele di interconnessione definite *wire segment*;

nei punti di intersezione tra un canale verticale e uno orizzontale sono posizionati gli *switch module*, i quali contengono la logica programmabile che realizza i collegamenti necessari.
Ogni L-Block, a sua volta, contiene una Look Up Table, con il suo vettore di bit. Nella figura 2.4 sono anche evidenziate la matrice di commutazione (S-Block) e la matrice delle connessioni dei pin di una LUT alle linee di collegamento (C-Block):

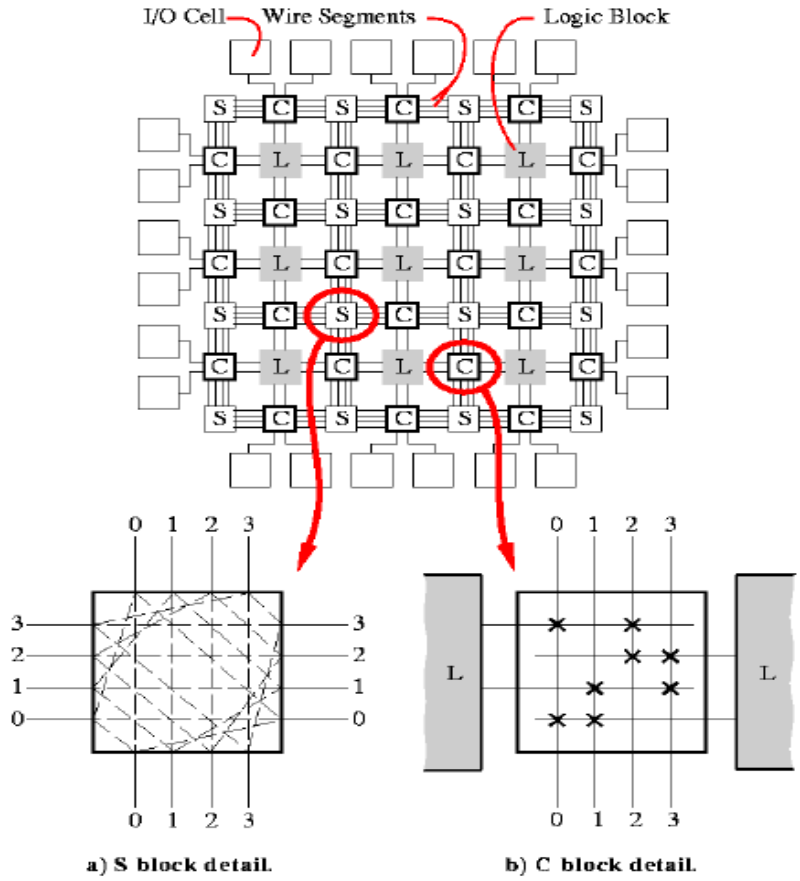


Fig 2.4 Zoom su una generica matrice di connessione (S-block) e sulle connessioni potenziali tra due LUT (C-Block)

fig 2.5 Scheda CAEN V1495 (a sinistra) e FPGA EP1C20F400C6 della famiglia Cyclone ALTERA (a destra).

Il chip FPGA montato sulla scheda CAEN V1495, (mostrata in fig 2.5) utilizzata in questo lavoro, è l' ALTERA Cyclon EP1C20F400C6. Esso è composto da 20060 elementi logici (LE, Logic Elements) che insieme consentono di arrivare a una memoria RAM interna di 294,912 bits. Ognuno dei LE contiene delle LUT di dimensione 16 x 1, dotate di quattro linee di indirizzo e di una per dati. Una LUT può così realizzare una qualsiasi funzione logica di quattro variabili da un bit. Oltre alle Look Up Table, un elemento logico contiene un flip-flop, configurabile per operazioni di tipo D, T, JK, o SR [7].

2.2 Ambiente di sviluppo

Lo sviluppo di un progetto basato su FPGA avviene di solito all'interno di un "ambiente di sviluppo" software fornito dalla ditta costruttrice o da terze parti. Tramite tale software è possibile programmare la FPGA, anche se parlare di programmazione del dispositivo è ormai diventato un po' restrittivo; se per una PAL era sufficiente un semplice software che implementasse le funzioni nella rete circuitale hardware, la complessità architetturale di un FPGA rende praticamente indispensabili un certo numero di strumenti di analisi e simulazione del funzionamento del dispositivo. Si tratta quindi di vere e proprie applicazioni CAD (*Computer Aided Design*) con le quali l'utente può sviluppare il proprio progetto. [8]

Con l'avvento delle logiche programmabili, si è sentita la necessità di introdurre opportuni linguaggi, denominati HDL (Hardware Description Language) attraverso i quali fosse possibile passare dalla descrizione funzionale, basata su equazioni logiche e disegni schematici, ad una descrizione interpretabile e compilabile da un apposito software. Un HDL è un tipo di linguaggio di programmazione volto alla descrizione di circuiti elettronici e, più specificamente, alla descrizione della logica digitale. Può descrivere il funzionamento del circuito, le sue caratteristiche e infine verificare le sue funzionalità grazie a opportune simulazioni. Appartenendo alla classe dei linguaggi di programmazione *concorrenti*, la sua sintassi e la sua semantica presentano la possibilità di definire istruzioni non sequenziali ma che vengono eseguite contemporaneamente. Quello che infatti differenzia un HDL dagli altri linguaggi di programmazione è la presenza del concetto di tempo, concetto fondamentale quando si ha a che fare con la descrizione di sistemi hardware. L'avvento degli HDL ha notevolmente modificato il lavoro dei progettisti di logiche programmabili, permettendo una verifica funzionale prima della costruzione dell'hardware vero e proprio. In figura 2.6 possiamo vedere il normale flusso di implementazione tramite HDL (a sinistra in

figura), confrontato con l'approccio usato prima dell'avvento degli HDL (destra). Si noti come una volta fosse necessario ottenere le equazioni logiche che regolavano il funzionamento del sistema e da esse ricavare lo schema delle interconnessioni degli elementi della logica programmabile che realizzavano tali funzioni.

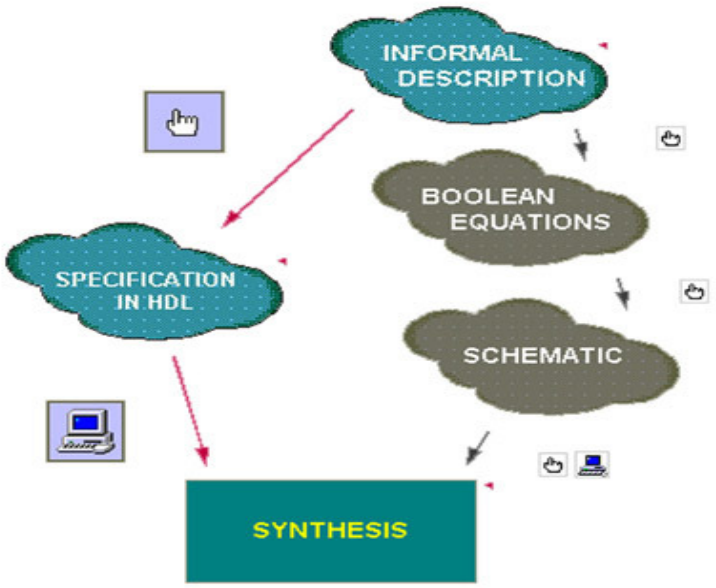


Fig 2.6 Fasi della programmazione di un disposiivo logico.

E' sicuramente possibile rappresentare una descrizione dell' hardware basandosi sui tradizionali linguaggi di programmazione come il C++, ma tali linguaggi non includono la possibilità di specificare esplicitamente la durata o l'istante di un determinato evento (ad esempio la transizione vero → falso di un segnale logico) [9]. I Linguaggi di programmazione di alto livello sono più semplici da usare ma danno un minore controllo sulle prestazioni del progetto; d' altro canto, descrivere un hardware generalmente è più complesso che scrivere un programma che svolge il medesimo compito, e spesso le applicazioni che vogliamo realizzare per l'hardware sono già state implementate nel mondo dei linguaggi software. Di conseguenza, si sta sviluppando un settore di ricerca molto attivo sulla conversione di programi scritti in linguaggi convenzionali (per esempio il C++) nell'equivalente HDL [9].

I linguaggi HDL attualmente più utilizzati sono il VHDL (*VHSIC, Very High Speed Integrated Circuits HDL*) diffuso soprattutto in Europa, la

cui sintassi richiama quella del Pascal e soprattutto dell'ADA, ed il Verilog, più simile al C. Esistono poi molti linguaggi proprietari, come l' AHDL (*Altera HDL*), CUPL della Logical Devices Inc., JHDL basato su Java, MyHDl basato sul linguaggio di programmazione Python etc.

In questo lavoro si è optato per il linguaggio VHDL, già impiegato dal gruppo di fisica nucleare del Dipartimento di Fisica e della sezione di Firenze dell' INFN (Istituto Nazionale di Fisica Nucleare) e ormai adottato da parecchie case produttrici di sistemi hardware. Il VHDL si sta inoltre affermando come standard, dato che le specifiche (sintesi, semantica) sono state fissate dalla IEEE *(Institute of Electrical and Electronics Engineers)* col documento IEEE Std 1076-1987.

Affinchè le istruzioni del linguaggio scelto per la descrizione dell' hardware vengano effettivamente eseguite, un programma software chiamato *synthesizer* può derivare le operazioni logiche dalle istruzioni definite nel linguaggio ad alto livello e produrre di conseguenza una *netlist*, cioè una rappresentazione strutturale del circuito che descrive le connessioni delle risorse logiche disponibili atte a implementare i comportamenti voluti.

Una volta sintetizzato, il progetto può essere trasferito sul dispositivo**.** Allo scopo di verificare la correttezza della programmazione dal punto di vista puramente logico, i sistemi di sviluppo includono simulazioni dette "funzionali". Una simulazione funzionale prescinde dal tipo di architettura scelta per l'implementazione fisica del dispositivo. Così facendo possiamo verificare tutte le operazioni logiche del circuito non curandoci dei ritardi fisici introdotti dalla propagazione dei segnali attraverso i vari moduli che possono comporre il progetto

A questa fase segue il *place and route*, detto anche *device fitting*, che si incarica di assegnare le risorse disponibili ai vari blocchi circuitali del dispositivo. L' obiettivo dello strumento di place and route è di assicurare un uso più efficiente possibile delle connessioni e della logica, nel rispetto dei vincoli temporali ed elettrici delle specifiche iniziali. Il place and route genera un file binario che viene usato per programmare il chip. Il progettista deve scegliere in questa fase il dispositivo FPGA target, cioè il preciso modello su cui deve essere trasferita la programmazione

Dopo il place and route, la programmazione è pronta per essere trasferita sul dispositivo programmabile, ma prima conviene verificare se i vincoli imposti dal particolare dispositivo permettono il corretto funzionamento del progetto. A questo punto abbiamo a disposizione un ulteriore strumento: la simulazione *temporale* (*timing simulation*). Questo tipo di simulazione, a differenza di quella funzionale, fornisce informazioni dettagliate sui tempi che intercorrono per la propagazione di un segnale all'interno del dispositivo. Naturalmente il ritardo totale complessivo di un circuito dipende dal numero di porte di cui sono composti i vari componenti del progetto. La simulazione temporale di tutto il progetto, detta simulazione full-timing, costituisce la verifica finale prima della programmazione del dispositivo.

La figura 2.7 riassume le diverse fasi dell'implementazione di un progetto così come sono realizzate dall'ambiente di lavoro QUARTUS II (vedi par. 2.3).

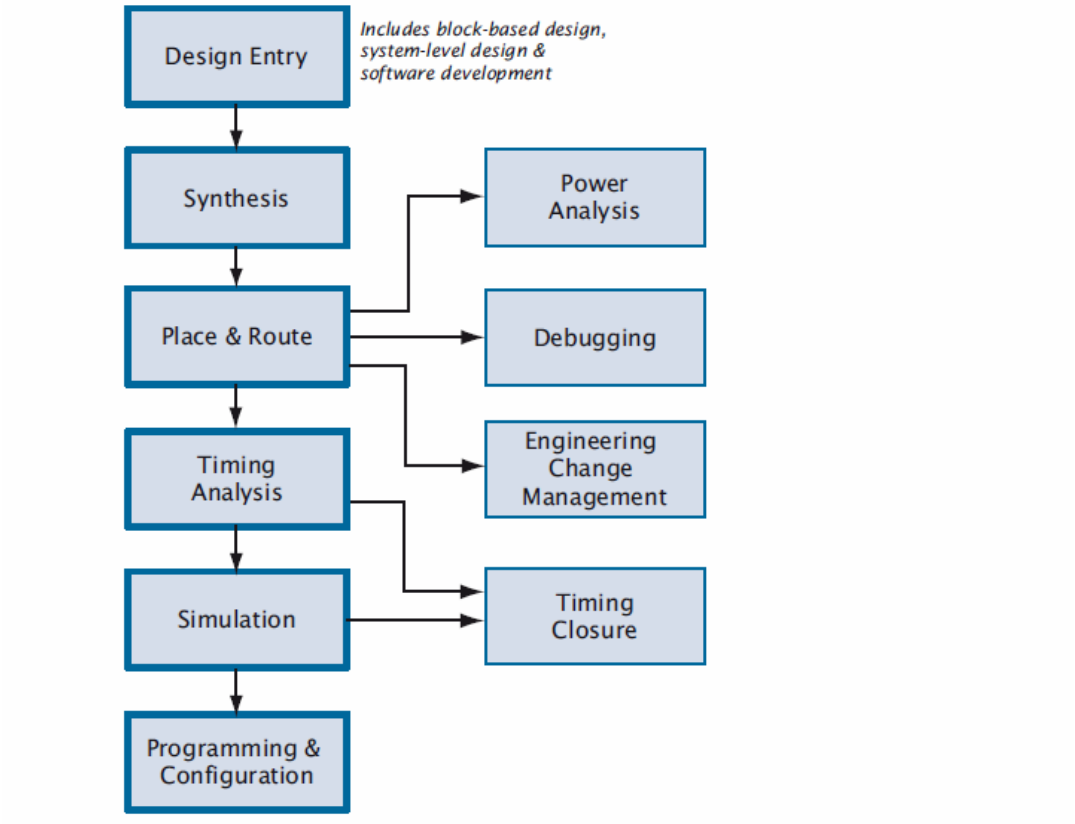


Fig 2.7 Design Flow del software QUARTUS II

2.2 VHDL ossia un linguaggio per la descrizione dell'hardware

Il VHDL, come abbiamo detto, è un linguaggio per la rappresentazione di sistemi elettronici digitali. E' uno dei linguaggi di rappresentazione di sistemi più utilizzato al giorno d'oggi, standardizzato e supportato dall' IEEE (*Institute of Electrical and Electronics Engineers*) e da varie società in tutto il mondo.
Dotato di una sintassi simile al linguaggio ADA, il VHDL fu sviluppato dal Department of Defense (DOD) americano, per permettere la documentazione dei comportamenti degli ASIC (*Application-specific integrated circuit*s, circuiti integrati specifici per applicazioni) che le compagnie fornitrici includevano nelle apparecchiature militari. Il passo successivo fu lo sviluppo di strumenti per la *sintesi logica* che potevano leggere files scritti in vhdl e di conseguenza produrre una descrizione del circuito in esame [10]. Gli strumenti per la sintesi più moderni possono ricavare dal listato VHDL le informazioni relativi, ad esempio, ai blocchi aritmetici o alle memoria necessaria, tenendo anche conto delle richieste dell' utente. Uno stesso codice VHDL, quindi, potrebbe essere sintetizzato diversamente a seconda delle specifiche sul clock, sulla potenza consumata o altro e, soprattutto, a seconda del particolare chip FPGA impiegato.
Fin dalla prima versione (1076-1987), il VHDL includeva una vasta gamma di tipi di dato, inclusi quelli strettamente numerici (*Integer* e *Real*), logici (i tipi *Bit* e *Boolean),* i tipi *Character* e *Time*, ed inoltre array di bit chiamati bit_vector e array di character chiamati *string*. In seguito, furono introdotti altri tipi di dato, per includere altri possibili livelli logici del segnale, fra cui anche il valore "unknown". Lo standard IEEE 1164 definisce nuovi tipi logici a 9 valori: lo scalare std_ulogic e la versione in forma vettoriale std_ulogic_vector.
Altri cambiamenti marginali nello standard (anni 2000 e 2002) hanno aggiunto caratteristiche tipiche dei linguaggi orientati ad oggetti, come i tipi protetti, e hanno rimosso restrizioni applicate in precedenza al mapping delle porte.
Nel giugno 2006, il Vhdl Technical Committee of Accellera (team delegato dall' IEEE per l'aggiornamento dello standard) ha approvato il cosiddetto Draft 3.0 per il *VHDL-2006*. Pur mantenendo una piena

compatibilità con le vecchie versioni, questo nuovo standard si propone di introdurre numerose estensioni al linguaggio che permettono una stesura e una manutenibilità del codice più semplice: incorporazioni di standard peculiari (1164, 1076.2, 1076.3) in quello generale (1076), un più esteso insieme di operatori, una più flessibile sintassi delle espressioni "case" (vedi pag. 27) e "generate".
Nel febbraio 2008, Accellera ha approvato VHDL 4.0,conosciuto anche come VHDL 2008,che include nuove modifiche alla Draft 3.0 e una gestione dei tipi migliorata [10].

2.2.1 Descrivere l'hardware con il VHDL

Illustriamo ora, in breve, alcuni elementi del VHDL frequentemente impiegati nel presente lavoro. Tale descrizione del linguaggio non ha nessuna pretesa di completezza, ma è volta solo a far meglio comprendere gli estratti del programma che compaiono nel capitolo 3. Come vedremo, un progetto VHDL, chiamato *entity,* ha uno o più ingressi ed una o più uscite che sono connesse a sistemi circostanti. Una entità è essa stessa composta da altri elementi, interconnessi ed operanti concorrentemente. Questi elementi possono essere altre entità oppure process o components (vedi pag. 22 e 24). In VHDL si impiegano i process sia per rappresentare circuiti sequenziali (cioè sincroni rispetto a un segnale di clock) dotati di flip flop e latch, sia per rappresentare circuiti combinatori (cioè non sincroni con un clock) dotati solo di porte logiche.[1]

2.2.2 Tipi di Dato

I processi comunicano uno con l'altro tramite dati di tipo ***signal***. Proprio questa caratteristica di essere "fili" di connessione fra processi concorrenti distingue i signal dagli altri tipi di dato: ***variable*** e ***constant.*** Un signal ha una sorgente che lo pilota (agisce cioè come "driver") ed una o più destinazioni, oltre a un tipo (bit, integer, boolean, etc) definito dal programmatore. Il collegamento fra due segnali si effettua con l'operatore "<=". Le dichiarazioni di segnali sono

[1] In questo paragrafo, il termine "sequenziale" è inteso come opposto a "combinatorio" e non, come nel seguito, come opposto a "concorrente"

ammesse in tutte le aree dichiarative tranne quelle di funzioni e processi.
Il tipo di dato *variable* può essere utilizzato solo all'interno di un processo e non è visibile all'esterno di esso. Esso svolge funzione analoga alle variabili di un comune linguaggio di programmazione. Esiste infine il tipo di dato *constant* che serve a mantenere valori che non hanno bisogno di essere modificati durante il funzionamento del progetto.

SINTASSI:

```
signal nome {, nome} : tipo_dato [range] [:= espressione] ;
```

ESEMPIO:

```
signal A, B, Y: std_logic;
        begin
                  Y <= A or B;
        End;
```

Nell'esempio riportato, al segnale y è collegato il risultato dell' OR tra i segnali A e B.

2.2.3 Entity

Un Entity è la più basilare delle strutture in un progetto VHDL. L'entity fa parte della minima descrizione di un sistema in vhdl e rappresenta in termini di input-output la sua interfaccia verso il mondo esterno. L'istruzione "entity" , di cui enunciamo la sintassi, dichiara una nuova *design entity* e ne definisce il nome. Al suo interno, l'istruzione *port* definisce le porte di ingresso e uscita. Le porte sono esse stesse segnali disponibili per il funzionamento interno dell'architettura (vedi 2.2.3) della

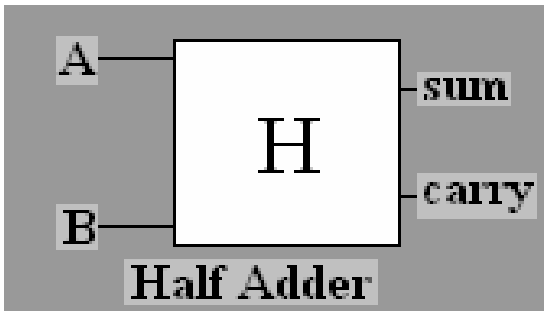


fig 2.8 Entity che rappresenta un half adder

entità, e, all'esterno, per il collegamento con altre entità.

SINTASSI:

```
entity nome_entità is
  [port ( nome_porta {, nome_porta} : modo tipo {;
          nome_porta {, nome_porta} : modo tipo } ) ;]
  {dichiarazione}
end [entity] [nome_entità] ;
```

Nella figura 2.8 possiamo notare la tipica rappresentazione grafica di un entity che descrive un Half Adder [11]: abbiamo due ingressi A e B che rappresentano gli operandi (ciascuno costituito da un singolo bit) e due uscite che sono rispettivamente la somma e il riporto dell'addizione dei due bit.
Per ogni porta si devono definire: *il nome e il tipo del segnale*.
Il segnale può essere di tipo:

- **In:** un segnale che entra nel sistema (è un segnale a sola lettura per l'entità stessa, mentre un'altra entità esterna vi può solo scrivere).
- **Out:** un segnale che esce dal sistema (l'architettura interna può scriverci, mentre le entità esterne vi possono solo leggere).
- **Inout:** segnale che può essere ingresso e uscita.

ESEMPIO:

Nell'esempio che segue è riportata la definizione della entity dell' *half adder* di fig. 2.8.

```
entity ha is
  port (A, B: in bit;
        Sum, Carry: out bit);
end;
```

2.2.4 Architecture

L'effettivo funzionamento di un *entity* è implementato dalla cosiddetta "architettura", introdotta dalla parola chiave **architecture.** Tutte le Entity effettivamente impiegate nel progetto devono avere una descrizione "architetturale". La descrizione determina il comportamento dell'entità. Una singola entità può essere associata a più descrizioni architetturali. Inoltre, come vedremo in seguito, un'architettura può essere di tipo *comportamentale*, oppure *strutturale* Come mostrato nel box "sintassi", il corpo della descrizione vera e propria inizia con la parola chiave **begin,** ed include tutte le istruzioni relative alla descrizione comportamentale. E' un'area concorrente, vale a dire che le istruzioni che contiene vengono eseguite contemporaneamente; il "body", che ha termine con la parola chiave **end**, può essere preceduto da una parte dichiarativa.

SINTASSI:

```
architecture nome_architettura of nome_entità is
  {dichiarazione}
begin
  {istruzione_concorrente}
end [architecture] [nome_architettura] ;
```

ESEMPIO:

Come esempio, diamo una possibile architettura per il funzionamento della design *entity* HA, la cui interfaccia è stata già descritta precedentemente. Il body contiene due istruzioni concorrenti, trattandosi di assegnazioni di segnali, che operano sui segnali A, B, Somma, Riporto già definiti come porte nell'entity.

```
architecture arc1 of ha is
begin
  Somma <= A xor B;
  Riporto <= A and B;
end;
```

Questa architettura è *comportamentale* perchè le istruzioni rappresentano equazioni logiche in ognuna delle quali i segnali A e B

rappresentano gli ingressi di un flusso di dati, i segnali Somma e Riporto rappresentano invece l'uscita del "flusso". Il termine "comportamentale" distingue questo tipo di architettura dal tipo detto "strutturale", basato sull'interconnessione di componenti predefiniti, come vedremo fra poco.

2.2.5 I Componenti e le Istanziazioni

Quando in un sorgente VHDL si vuole impiegare una entità definita altrove, si usa l'istruzione *component*. È bene, quindi, chiarire la differenza tra *entity* e *component*.
La entity descrive un'interfaccia di un modulo logico, mentre il component descrive l'interfaccia di una entity, già definita, che sarà usata grazie alla sua *istanziazione*. L'istanza rappresenta una copia distinta del component che viene collegata ai segnali del modello. In analogia con un linguaggio di programmazione *Object-Oriented*, possiamo immaginare la entity come una *classe* che descrive l'interfaccia e come le parti interagiscono tra di loro. Il component invece è un *oggetto* dichiarato di quella classe, e l'istanziazione del componente è infine l'istanziazione dell'oggetto, cioè la generazione di un oggetto che opera indipendentemente dagli altri.
Per istanziare il componente, si deve definire il *nome da assegnare all'istanza,* indicare il *nome dell'elemento istanziato* specificare l'entità associata al componente. Nel cosiddetto *port mappin,* infatti, si associano le porte del componente istanziato a segnali dell'architettura corrente, cioè dell'architettura della quale il componente (o meglio la sua istanza) entra a far parte. Esistono due modi per effettuare il port mapping

- **Associazione posizionale:** si inserisce la lista dei segnali dell'architettura nello stesso ordine in cui sono definite le porte del componente istanziato.

- **Associazione per nome:** si inserisce la lista delle coppie porta-segnale*,* separati dal simbolo "=>" in un ordine qualsiasi.

La dichiarazione del component va effettuata nell'area dichiarativa di un'architettura

SINTASSI:

Dichiarazione di un component e relativa istanziazione:

```
component nome_componente [is]
  [port ( nome_porta {, nome_porta} : modo tipo {;
          nome_porta {, nome_porta} : modo tipo } ) ;]
end component
```

```
nome_istanza :
  ([component] nome_componente)
  | (entity nome_entity [ ( nome_architettura ) ])
  [port map ( [formale =>] attuale {,
              [formale =>] attuale } )] ;
```

ESEMPIO:

Il listato mostra un esempio di “Full-Adder”, un sommatore che somma due bit e tiene conto di un eventuale bit di riporto prodotto da somme precedenti. Tale oggetto può essere realizzato con due component già precedentemente definiti, gli Half-Adder, connessi come mostrato in fig 2.9.

```
entity fa is
  port(X, Y, Cin: in bit;
       Z, Co: out bit);
end entity fa;

component ha is
 port (A, B: in bit;
        S, Riporto: out bit);
End component;

architecture afa of fa is
  signal S1, C1, C2:  bit;
begin

  ha1: entity ha
    port map (A=>X, B=>Y, S=>S1, Riporto=>C1);

  ha2: entity ha
    port map (A=>S1, B=>Cin, S=>Z, Riporto=>C2);

  Co <= C1 or C2;

end architecture fa
```

La design entity di *HA* con ingressi A,B, e uscite Somma e Riporto è stata già definita (pag 20). Ha1 e Ha2 sono istanze dell’entity *HA*. La porta A della prima istanza è mappata sul segnale X del Full-Adder, la porta B al segnale Y, etc.

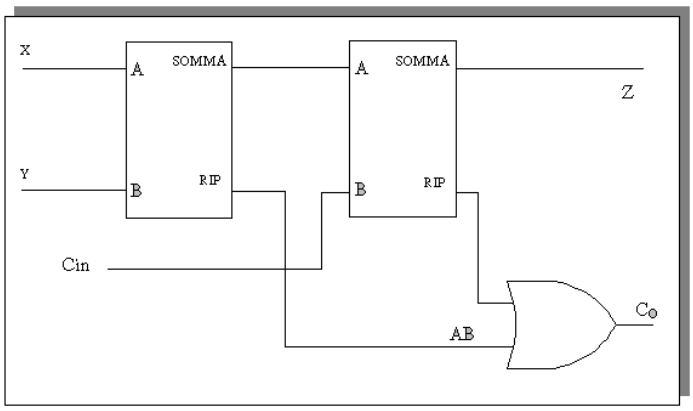


Figura 2.9 Esempio schematico di un FULL ADDER

L'architettura di questo Full Adder è quindi un esempio di architettura "strutturale".

2.2.6 Process

Nell'ambito del *body* di una architettura, un process è un'istruzione concorrente che definisce un'area "sequenziale"[2]. Le istruzioni sequenziali che può contenere si inseriscono nel cosiddetto *body del processo*, che inizia con la parola chiave *begin* e può essere preceduto da un'area dichiarativa. Un processo viene eseguito parallelamente alle altre istruzioni concorrenti e quindi, se in un'architettura ci sono più processi, questi sono attivati in modo concorrente, e **non** in sequenza. In fase di simulazione del funzionamento del progetto, l'esecuzione delle istruzioni contenute nel processo può essere condizionata da una ***sensitivity list,*** una lista che rappresenta i segnali su cui il processo deve essere attivato, nel senso che tutte le istruzioni contenute nel *body* verranno eseguite solo in caso di un cambiamento di stato dei segnali su tale lista. Tale cambiamento di stato costituisce un "evento" per il processo.

[2] Usiamo qui il termine "sequenziale" come opposto di "concorrente".

SINTASSI:

```
[nome_processo :] process [(sensitivity_list)]
  {dichiarazione}
begin
  {istruzione_sequenziale}
end process [nome_processo] ;
```

ESEMPIO:

In questo esempio il processo p1 segue due assegnazioni, che avvengono contemporaneamente al verificarsi di un evento sui segnali B o C.

```
p1: process (B, C)
begin
  A <= B and C;
  C <= '0';
end;
```

E' interessante notare che il valore di C impiegato per calcolare A è preso all'istante in cui il processo è stato "svegliato" dall'evento, mentre l'assegnazione di '0' al segnale C avviene successivamente.
In quest'altro esempio invece il processo *p2* scambia i valori dei segnali A e B servendosi della **variabile** TMP dichiarata come array di std_logic_vector di 4 bit (assumendo che i segnali sulla sensitivity list siano array di 4 bit)

```
p2: process (A, B)
  variable TMP: std_logic_vector(3 downto 0);
begin
  TMP := A;
  A <= B;
  B <= TMP;
end;
```

Nel contesto di un processo, si nota una delle differenze di comportamento fra "signal" e "variable": mentre l'assegnazione di un nuovo valore ad una variabile ha effetto immediato, così che il nuovo valore è disponibile già all'istruzione successiva, l'assegnazione di un nuovo valore ad un segnale ha luogo solo al termine dell'evento. Se il valore di un segnale viene modificato più volte nel corpo del processo, solo l'ultima assegnazione avrà veramente effetto.
E' bene ricordare infine che la *sensitivity list* è solamente un'indicazione per lo strumento di simulazione sia funzionale che temporale affinchè tale strumento sappia quando un processo deve

essere eseguito, non dovendo più eseguire le istruzioni ogni volta che un qualsiasi segnale cambia stato, sgravando il processore di un carico notevole. La sintesi delle istruzioni VHDL, invece, crea solamente la logica del progetto dal codice scritto, seguendo delle linee guida già create, ignorando completamente la sensitivity list.
Il software QUARTUS II produce un avvertimento qualora una sensitivity list dichiarata sia incompleta.

2.2.7 Istruzioni Sequenziali

Le istruzioni sequenziali, trovano la loro collocazione tipicamente nel corpo di un processo. In un assegnazione sequenziale un segnale o una variabile, indicati alla sinistra di una equazione logica, ricevono un valore risultante da un'espressione indicata a destra che elabora eventualmente altri segnali o variabili.
Mentre un segnale è un tipo di dato il cui valore può essere cambiato concorrentemente, una variabile invece serve a conservare il valore di un dato in una sequenza di istruzioni; quindi, per loro natura, gli assegnamenti di variabili sono ammessi solo in aree sequenziali.
Diversamente dalle assegnazioni concorrenti, quelle sequenziali sono eseguiti appunto in sequenza.

SINTASSI:

Assegnazione sequenziale di un segnale:

```
[etichetta :]
 nome_segnale <= espressione [after ritardo] {, espressione [after ritardo]} ;
```

Assegnazione sequenziale di variabile:

```
[etichetta :] nome_variabile := espressione ;
```

Istruzioni *IF* e *CASE.*

Un If statement permette a un flusso di istruzioni sequenziali di decidere quale direzione prendere tra più alternative. In base alla selezione, viene eseguita una tra più sequenze di istruzioni. Nel caso del *case* statement un'espressione viene valutata e, in base al suo risultato, viene selezionata una tra più alternative.

SINTASSI:

Selezione If:

```
 [etichetta :]
if condizione then
  {istruzione_sequenziale}
{elsif condizione then
  {istruzione_sequenziale}}
[else
  {istruzione_sequenziale}]
end if [etichetta] ;
```

Selezione Case

```
[etichetta :]
case espressione is
  when valore =>
    {istruzione_sequenziale}
 {when valore =>
    {istruzione_sequenziale}}
 [when others =>
    {istruzione_sequenziale}]
end case [etichetta] ;
```

ESEMPIO:

Un primo esempio di if statement è la seguente assegnazione condizionata (ricordiamo che questo tipo di istruzione deve essere contenuta in un processo):

```
if A = B then
  C <= D;
else
  C <= E;
end if;
```

Nel caso in cui il valore del segnale A sia uguale al valore del segnale B, verrà applicato il primo ramo dell'if e conseguentemente al Segnale C verrà collegato il segnale D, in caso contrario, al segnale C verrà collegato il segnale E.

Come esempio di utilizzo del “case” riportiamo un semplice multiplexer

```
case S is
  when "00" => Y <= A;
  when "01" => Y <= B;
  when "10" => Y <= C;
  when "11" => Y <= D;
end case;
```

La valutazione di S (che in questo caso deve contenere uno dei valori presenti nel listato) produrrà la “scelta” di quale segnale collegare a Y, proprio come in un multiplexer.

Iterazioni

Per eseguire ripetutamente una serie di istruzioni sequenziali si usa l’istruzione **loop.**
Ci sono tre tipi di loop:

- Loop con schema di iterazione *while:* l’esecuzione viene controllata dal verificarsi di una condizione specificata;
- Loop con schema di iterazione *for:* l’esecuzione viene controllata dalla variazione del valore di un parametro specificato;
- Loop senza schema di iterazione: l’esecuzione è controllata solo dalle stesse istruzioni interne

Il tipo di loop che abbiamo scelto per l’implementazione del codice è il loop con schema di iterazione “for” di cui sotto proponiamo la sintassi e un esempio.

SINTASSI:

Loop:

```
        [etichetta :]
[while condizione] | [for nome in range] loop
  {istruzione_sequenziale}
end loop [etichetta] ;
```

ESEMPIO:

un ciclo for per sommare i 100 elementi di un vettore x, escluso il decimo:

```
for I in 0 to 99 loop
  next when I := 9;
  S := S + X(I);
end loop;
```

2.2.6 Packages e Librerie

Un modo per ottenere un progetto modulare è quello di isolare fuori dalle altre unità di progetto (design unit) le porzioni riutilizzabili di codice, cosa che permette poi di usare porzioni di codice definite in moduli esterni al progetto. Un **package** è una design unit che può contenere, isolatamente dal resto del sorgente, dichiarazioni di oggetti, sottoprogrammi e componenti. Una **library** è un contenitore di design units analizzate (compilate). Le librerie contengono elementi esterni riusabili precompilati.

Anche tutto ciò che l'utente programma nel progetto corrente viene inserito in una libreria. Tutte le design units create nel sorgente (non solo i packages), sono infatti automaticamente inserite dal compilatore nella libreria corrente il cui nome di default è **work**.

Un *modello* VHDL è un insieme di librerie contenenti design units che a loro volta contengono altri elementi con nome. Questa struttura gerarchica implica la creazione di **name spaces** (spazi dei nomi) e di regole di **visibilità** degli elementi. Un elemento dichiarato in un processo, sarà visibile solo in quel processo. Un elemento dichiarato in un'architettura o in un'entità sarà visibile rispettivamente solo sotto quell'architettura o sotto tutte le architetture dell'entità, compresi i processi interni. Un elemento dichiarato in un package sarà visibile sotto tutte le design units che usano quel package. Questo significa anche che ci possono essere elementi con lo stesso nome, purchè appartenenti a spazi di nomi diversi.

Se dal sorgente di una design unit si vuole fare riferimento ad un'altra design unit o ad un elemento contenuto in un'altra design unit, si deve utilizzare il **nome selettivo** dell'elemento. Nel nome selettivo il nome dell'elemento è preceduto dal nome della sua design unit e dal nome della sua libreria, separati da un punto. Inoltre la design unit che fa uso di oggetti di librerie esterne deve ricorrere all'istruzione **library** che specifica la libreria esterna a cui si fa riferimento. La libreria corrente *work* è sempre visibile automaticamente.

Per evitare l'utilizzo del nome selettivo completo, la design unit che contiene riferimenti all'esterno deve essere preceduta da istruzioni **use** che specificano i nomi selettivi o gli spazi dei nomi che si vogliono utilizzare. Si può usare la parola chiave **all** al posto del nome della design unit o del nome dell'elemento per indicare tutti gli elementi di quello spazio.

I vari sistemi di sviluppo sono in genere forniti di alcune librerie di corredo che contengono gli elementi standard del linguaggio ed eventuali elementi proprietari. Gli elementi standard principali (come i tipi *bit* e *integer*) sono contenuti nella libreria **std** e sono automaticamente visibili senza la necessità di una *use*. Invece gli elementi IEEE-1164 (come il tipo *std_logic*) sono inseriti nel package **std_logic_1164** della libreria **ieee** e devono essere resi visibili esplicitamente con una *use*.

SINTASSI:

Definizione di un package:

```
package nome_package is
  {dichiarazione}
end [package] [nome_package] ;
```

Definizione di un package body:

```
package body nome_package is
  {dichiarazione}
  {definizione}
end [package body] [nome_package] ;
```

Riferimenti agli elementi riusabili:

```
library nome_libreria {, nome_libreria} ;
use nome_selettivo {, nome_selettivo} ;
```

ESEMPIO:

Ad esempio, ecco il semplice package che abbiamo utilizzato nel lavoro svolto per la tesi, allo scopo di rendere scalabile il progetto:

```
package T_Sconstants is
constant NBIT_SUBTRIG : integer :=32;
constant NBIT_TRIG : integer := 8;
constant NBIT_DATAIN : integer := 16;
constant NBIT_DATAOUT : integer := 16;
constant NBIT_ADDR : integer := 16;
constant NBIT_DEBUG : integer := 16;
```

Il package T_Sconstants contiene delle costanti per la definizione della dimensioni degli array di segnali. Grazie a questo package, possiamo cambiare a piacimento le costanti e di conseguenza aggiornare in un colpo solo tutte le dimensioni degli array che fanno riferimento al package (come per esempio il numero di subtrigger in ingresso alla nostra trigger box).

2.3 Ambiente di sviluppo QUARTUS II

Per scrivere il codice vhdl di questo lavoro, l'utilizzo di un ambiente di sviluppo adeguato si è reso necessario. In questo caso, la scelta è caduta su QUARTUS II sviluppato dall' ALTERA, produttore della FPGA "target" del nostro codice. QUARTUS II fornisce un ambiente di sviluppo completo e soprattutto tutte le possibili soluzioni a problemi di stesura di codice sia per FPGA che per CPLD.
Abbiamo già mostrato, fig 2.9, come QUARTUS II permetta di eseguire tutte le fasi necessarie per passare da un sorgente vhdl al file sintetizzato, pronto per essere in seguito caricato sulla FPGA.

Una chiara interfaccia grafica permette un semplice utilizzo del software in tutte le fasi descritte in figura, così come un'interfaccia a linea di comando, permettendo allo sviluppatore uno scambio di

interfaccia per una fase o per un'altra, a seconda dell'esigenze. All'avvio, l'interfaccia grafica si presenta come in fig 2.10.

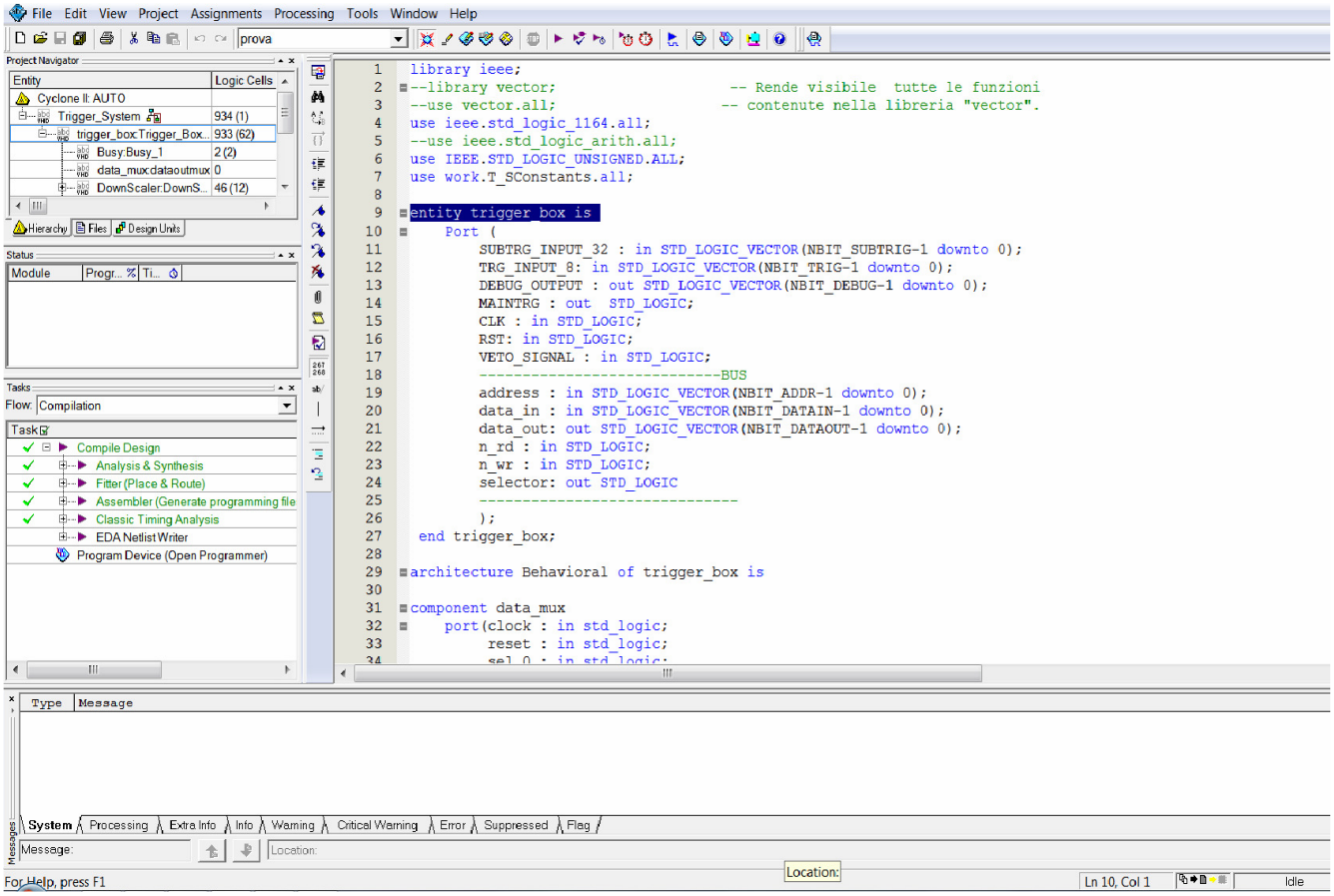


Fig 2.10 Interfaccia Grafica iniziale di QUARTUS II

CAPITOLO 3

Descrizione della Trigger Box

3.1 Introduzione

La Trigger Box realizzata per questo lavoro di tesi è suddivisa in cinque moduli principali, cui si aggiungono tre moduli *ausiliari* chiamati *scaler* che servono ai moduli principali come contatori.
I trigger di primo livello che arrivano dai rivelatori o gruppi di rivelatori dell'apparato sperimentale sono chiamati *subtrigger*. La presente versione della Trigger Box accetta in ingresso 32 segnali di subtrigger. Questi 32 ingressi vengono analizzati e modificati da appositi moduli logici per produrre 8 segnali logici, detti *trigger* che identificano vari tipi di eventi di collisione ai fini dell'acquisizione[3].
Una porta logica OR, avente in ingresso gli 8 segnali di trigger, produce il cosiddetto *Main Trigger* che rappresenta il segnale più importante e lo scopo principale del dispositivo: esso infatti abilita la conversione di tutti gli ADC e degli altri sistemi per l'acquisizione e predispone il calcolatore, tramite un *interrupt*, alla successiva lettura dei dati.

Le funzioni svolte dai vari moduli della trigger box possono essere così suddivise:

1) Rendere tutte uguali le lunghezze dei segnali di ingresso (intendendo "lunghezza" la durata del periodo per cui mantengono lo stato logico "vero").

2) Calcolare una serie di funzioni logiche a partire dagli ingressi di sub trigger: ciascuna funzione è associata a un particolare tipo di evento che interessa acquisire.

3) Bloccare la generazione del Main Trigger quando il sistema di acquisizione non è pronto.

[3] Il numero di subtrigger ed il numero di trigger può essere facilmente modificato, prima della sintesi del progetto, tramite le costanti NBIT_SUBTRIG e NBIT_TRIG contenute nel package T_Sconstants (vedi paragrafo 2.2.6)

4) Stabilire in che percentuale un certo tipo di evento deve contribuire al Main Trigger, mediante eventuale riduzione della frequenza dei singoli segnali di trigger (downscaling).

5) Generare il Main Trigger che dà il via all'acquisizione (conversione degli ADC e interrupt della CPU).

6) Memorizzare in un registro, detto *bit-pattern*, la situazione dei trigger che ha dato luogo a un Main Trigger.

7) Contare i trigger generati *pre-veto, post-veto, post-downscaling* (servono, ad esempio, a valutare la percentuale di "tempo morto" del sistema di acquisizione).

3.2 Schema della Trigger Box

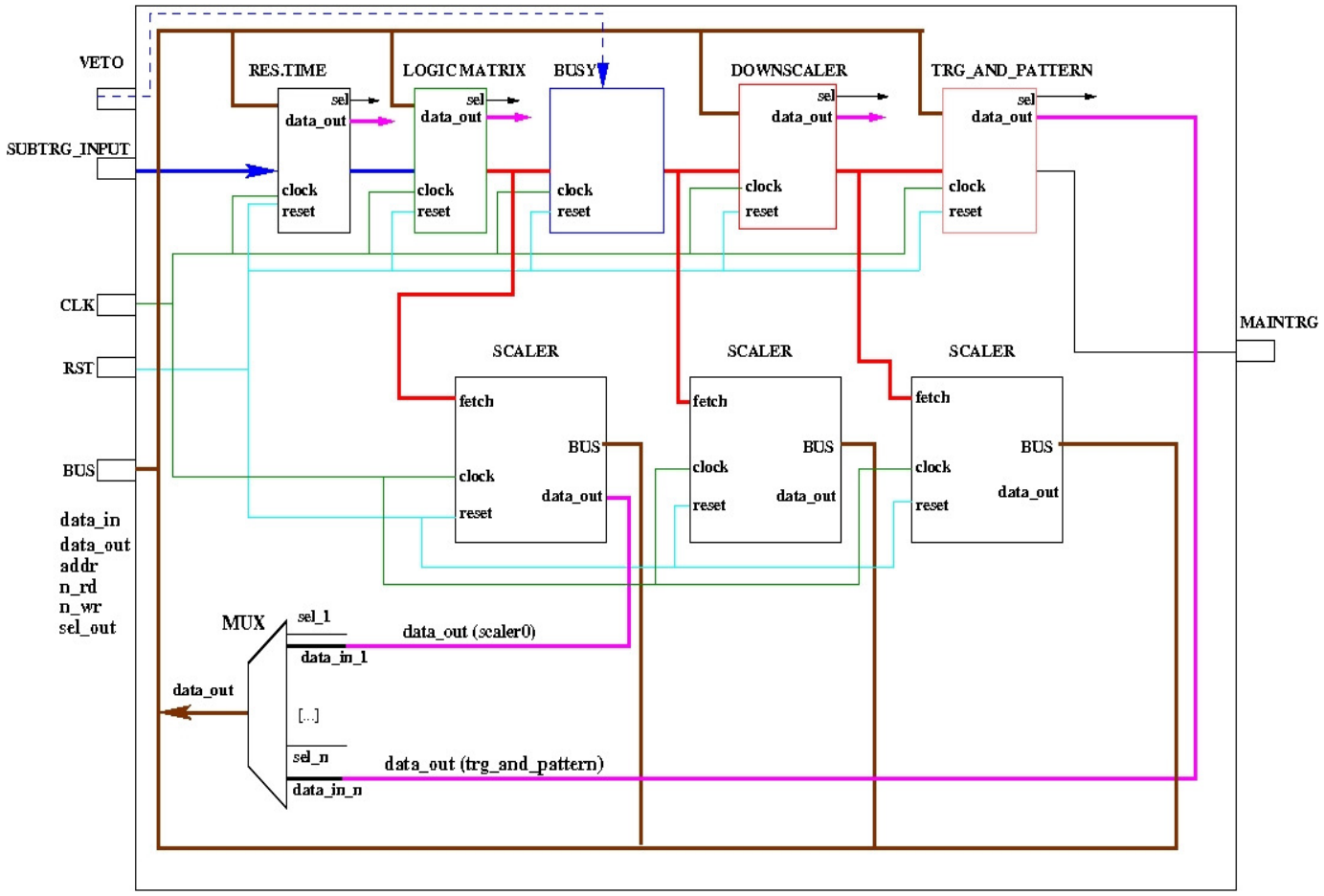


Fig 3.1 Schema Trigger Box

La figura 3.1 rappresenta nell'insieme la Trigger Box. Ci sono 4 ingressi ed un bus per l'interfacciamento della trigger box con la cpu di controllo.
I 4 ingressi sono, in ordine, "**VETO**", "**SUBTRG_INPUT**", "**CLK**", "**RST**". L'unica uscita è "**MAINTRG**".

- **VETO**: il segnale di veto permette il blocco dall'esterno della generazione del trigger.

- **SUBTRG_INPUT**: si tratta, piuttosto che di un singolo segnale, di un bus di sola lettura costituito dagli ingressi che provengono dai rivelatori.

- **CLK**: rappresenta il clock di sistema, che scandisce le operazioni, impostato a un periodo di 25 ns.

- **RST**: è il segnale di reset, necessario per forzare il sistema in una situazione nota, con l'azzeramento di tutti i contatori, l'uscita dei vari moduli posta a "0" etc.

- **BUS**: si tratta di un insieme di segnali per il trasferimento di dati tra la Trigger Box ed una CPU esterna. Un comune BUS del tipo impiegato, ad esempio, nei personal computer è un'insieme di linee conduttrici condivise dalle varie periferiche, le quali a turno possono prendere il controllo delle linee e usarle per trasferire i propri dati. Tali linee sono solitamente ripartite fra linee "dati", linee "d'indirizzo" e linee di controllo. Particolari circuiti di uscita permettono alle periferiche di "disconnettersi" dalle linee stesse (i cosiddetti circuiti "tri-state"). Questa situazione non è di solito riproducibile in una FPGA programmata in VHDL in quanto un dato segnale VHDL può essere "scritto" da uno ed un solo processo (vedi paragrafo 2.2.2). Per questo motivo nel nostro BUS ogni modulo ha un suo segnale *data_out* per l'output dei suoi dati. I vari *data_out*, successivamente, sono connessi in ingresso a un multiplexer (in basso a sinistra in fig 3.1). Per scrivere nei registri della trigger box è sufficiente un solo segnale "**data_in**" comune a tutti i moduli. Il bus prevede anche un segnale "**Addr"** per l'indirizzamento dei registri e i segnali **n_rd** e **n_wr** che abilitano rispettivamente la lettura o la scrittura dei registri stessi. La larghezza dei segnali data_in, data_out e addr è controllata da costanti inserite nel package T_Sconstant ed è attualmente impostata a 16, coerentemente con quanto richiesto dal bus della scheda CAEN V1495.

Il segnale di uscita è chiamato **MAINTRG**, cioè il trigger principale. Tale segnale richiede l'acquisizione e la memorizzazione delle informazioni elaborate dai vari blocchi che compongono l'apparato sperimentale.

3.2 I MODULI DELLA TRIGGER BOX

3.2.1 RESOLVING TIME

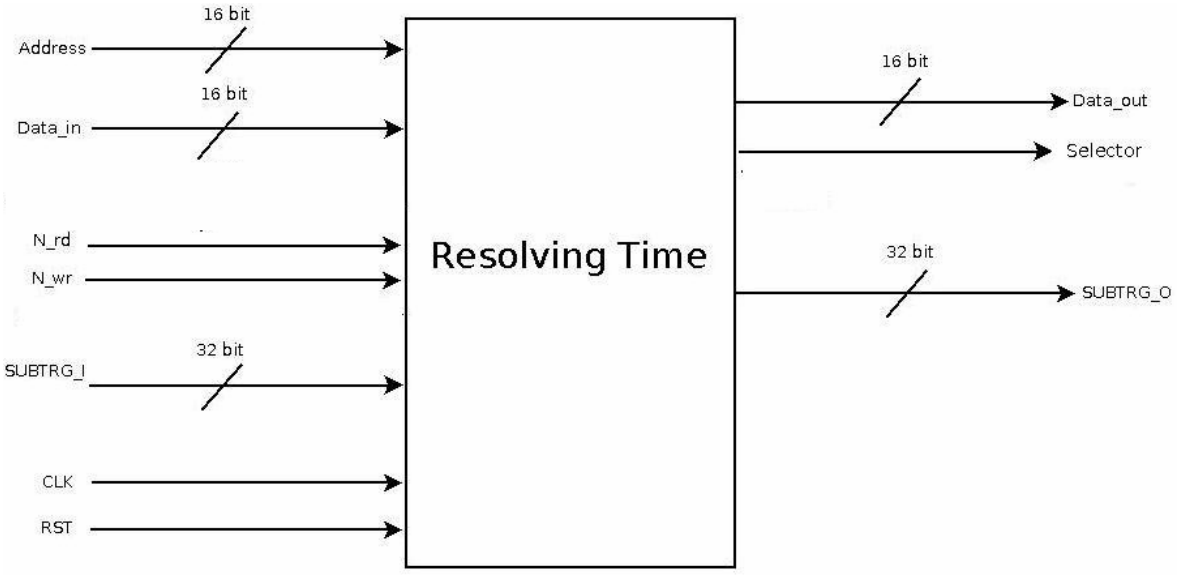


fig 3.2 Resolving time

```
entity Resolving_Time is

    port(   SUBTRG_I : in STD_LOGIC_VECTOR(NBIT_SUBTRIG-1 downto 0);
            SUBTRG_O : out std_logic_vector(NBIT_SUBTRIG-1 downto 0);
            CLK : in STD_LOGIC;
            RST : in STD_LOGIC;
            ---------------------------BUS
            address : in STD_LOGIC_VECTOR(NBIT_ADDR-1 downto 0);
            data_in : in STD_LOGIC_VECTOR(NBIT_DATAIN-1 downto 0);
            data_out: out STD_LOGIC_VECTOR(NBIT_DATAOUT-1 downto 0);
            n_rd : in  STD_LOGIC;
            n_wr : in  STD_LOGIC;
            selector: out  STD_LOGIC
            ----------------------------
     );
end Resolving_Time;
```

Listato 3.1 Dichiarazione dell'entity Resolving Time.

Lo scopo del modulo Resolving Time (la cui dichiarazione vhdl è nel listato 3.1) è definire, in modo indipendente dalla durata degli ingressi, il massimo intervallo di tempo entro il quale due eventi possono essere considerati *coincidenti* . La coincidenza di n segnali si ha quando ad un dato istante gli n segnali sono tutti veri. Nella figura 3.3 è mostrato il principio di funzionamento del modulo. I segnali di uscita hanno tutti la stessa durata T_r indipendentemente dalla durata dei segnali di ingresso. Si nota come qualunque sia la durata degli ingressi e in qualunque ordine si presentino, la coincidenza si ha per $|t_1 - t_2| < T_r$. E' possibile impostare il tempo risolutivo tramite il bus.

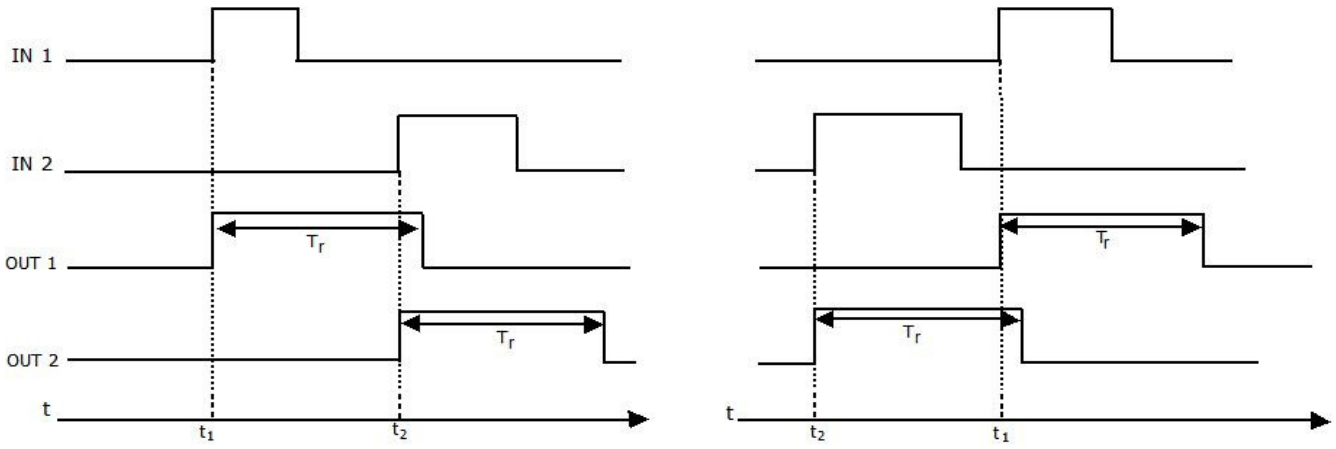


Fig 3.3 Resolving period

```
COMPONENT BIT_REGISTER_16

   PORT(         Data_in: IN STD_LOGIC_VECTOR (NBIT_DATAIN-1 DOWNTO 0);
                 CLOCK: IN STD_LOGIC;
                 RESET: IN STD_LOGIC;
                 Data_Out: OUT STD_LOGIC_VECTOR(NBIT_DATAOUT-1 DOWNTO 0)
         );
end component;

component simple_counter

   PORT
   (
         clock         : IN STD_LOGIC ;
         cnt_en        : IN STD_LOGIC ;
         sclr          : IN STD_LOGIC ;
         q             : OUT STD_LOGIC_VECTOR (5 DOWNTO 0)
   );

end component;
```

Listato 3.3 dichiarazione componenti resolving time.

Ricordiamo che le costanti NBIT_SUBTRIG, NBIT_TRIG, NBIT_ADDR, NBIT_DATAIN, NBIT_DATAOUT, che compaiono in questi e nei seguenti listati, sono definite nel package T_Sconstants.vhd e sono attualmente impostate rispettivamente a 32, 8, 16, 16, 16.
All'interno del Resolving Time (listato 3.2) troviamo il componente: **simple_counter**. Simple_counter è una *megafunction* fornita dall'ambiente di sviluppo Quartus II.[4]
Il compito di **simple_counter** è quello di contare i cicli di clock trascorsi da quando un dato ingresso è diventato "vero".
A ciascuno dei segnali in input (chiamati SUBTRG_I) è associato un contatore di tipo simple_counter. Quando un ingresso diventa "vero", la corrispondente uscita (resolve_out) viene posta "vera" anch'essa e viene abilitato il contatore tramite il segnale counter_start (primo if nel ciclo del listato 3.4)

```
RESOLVING:PROCESS(CLK,RST)
BEGIN
   IF RISING_EDGE(CLK)THEN

         selector <='0';
         CICLO:FOR i IN NBIT_SUBTRIG-1 DOWNTO 0 LOOP
               counter_reset(i) <= '0';
               IF  (SUBTRIG_I(i) = '1' and resolve_out(i) = '0')THEN
                     counter_start(i) <= '1';
                     resolve_out(i)<= '1';
               END IF;

               IF(resolve_out(i) = '1' and
                    conv_integer((to_matcher(i))) =
conv_integer(register_data_out))THEN
                              counter_reset(i) <= '1';
                              counter_start(i) <='0';
                              resolve_out(i) <= '0';
               end if;

         end loop;

   END IF;
END PROCESS resolving;
```

Listato 3.4 Processo del resolving time

Quando il conteggio del contatore è pari al contenuto del registro, il contatore viene fermato e la corrispondente uscita diventa "falsa": il segnale vero avrà quindi la lunghezza desiderata. Nel listato 3.4 è anche compreso il controllo del flusso di esecuzione che effettua il confronto tra le uscite dei 32

[4] Le *megafunctions* sono blocchi "prefabbricati" di codice che implementano funzioni comuni come ad esempio un contatore di segnale, e sono ottimizzate per il particolare dispositivo programmabile che si intende impiegare.

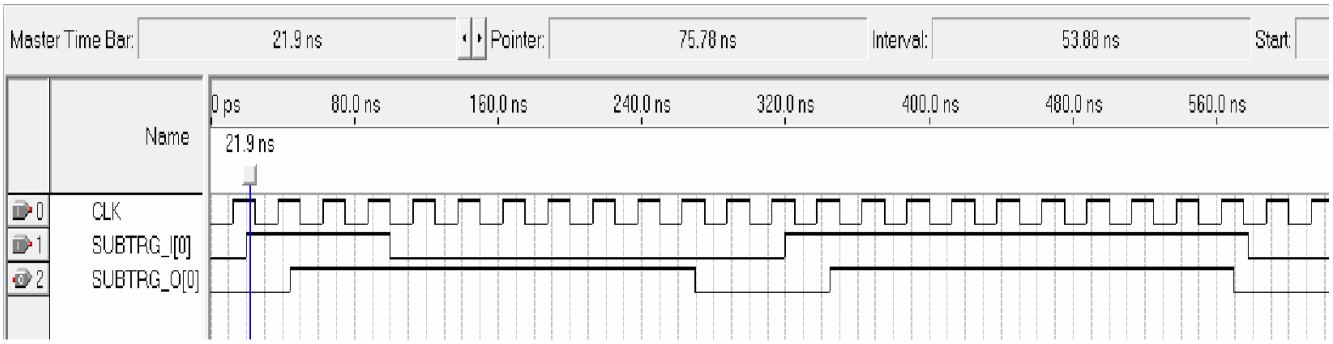
contatori (raccolte in un array di vettori di segnali, chiamato **to_matcher**) e il valore scritto nel registro. Si tratta di questa parte del codice:

```
IF(CONV_INTEGER((TO_MATCHER(I))) = conv_INTEGER(register_data_out + 1))THEN
counter_reset(i) <= '1';
counter_start(i) <='0';
resolve_out(i) <= '0';
end if;
```

Listato 3.5 porzione di codice vhdl di Resolving Time.

La funzione CONV_INTEGER accetta come parametro un *std_logic_vector* e converte il valore binario espresso dal parametro in un intero. La conversione a integer è necessaria a causa tipizzazione del VHDL, perché i segnali to_matcher(i) e register_data_out non hanno la stessa larghezza in bit: il confronto verrà quindi fatto tra numeri interi. Se il confronto dà esito positivo, il segnale **counter_reset (i)** e Il segnale **counter_start (i)** vengono posti rispettivamente a '1' e '0' bloccando il **simple_counter** relativo al segnale i-esimo che ha raggiunto la durata richiesta.
Infine, lo *std_logic_vector* **resolve_out** viene messo a 0, indicando la fine del prolungamento del segnale secondo il periodo voluto.
Nella simulazione 1 è mostrato il risultato di una simulazione della risposta del modulo, ottenuta grazie a Quartus II. La simulazione tiene conto delle caratteristiche fisiche della FPGA impiegata, compresi i ritardi introdotti dai vari componenti. Tutte le simulazioni qui riportate sono state ottenute allo stesso modo.

Simulazione Resolving Time:



Simulazione 1: Resolving Time

Si può notare come l'input SUBTRG_I[0] sia prolungato secondo il periodo voluto (in questo caso il periodo risolutivo è impostato a 9 cicli di clock). Il modulo inserisce anche un ritardo massimo di un periodo di clock dovuto al

fatto che il segnale in ingresso viene campionato sul fronte ascendente del clock: SUBTRG_I[0], nella figura, viene riconosciuto come “vero” sul primo fronte ascendente del segnale CLK successivo alla transizione 0 → 1 dello stesso SUBTRG_I[0].

3.2.2 LOGIC MATRIX

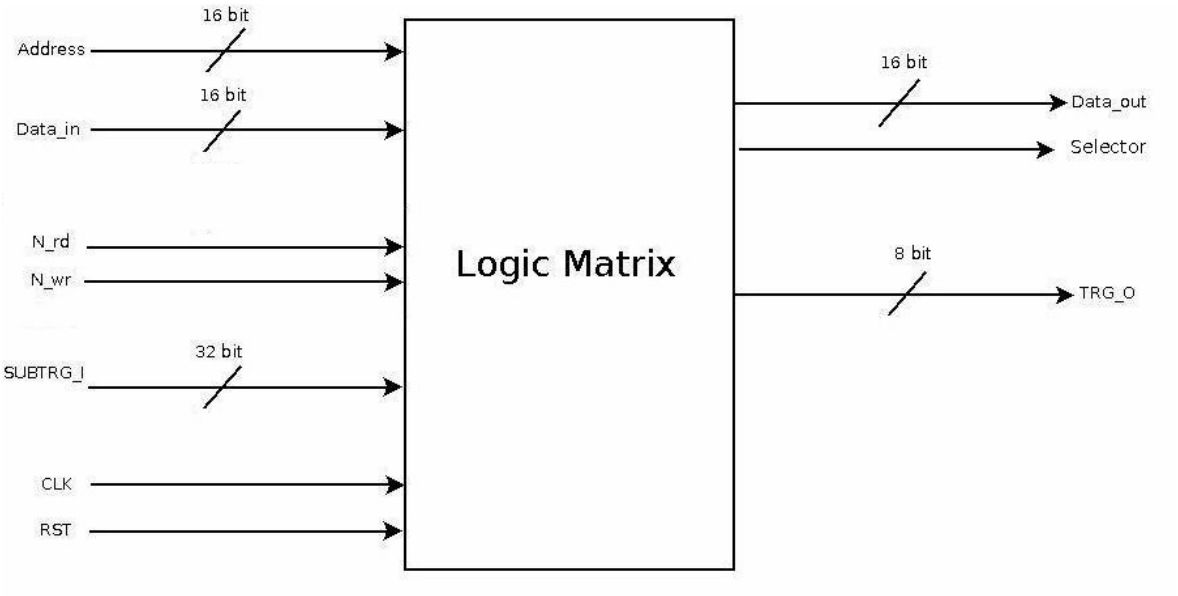


fig. 3.4 Logic Matrix Module

La Logic Matrix è il modulo che trasforma i trigger di primo livello (i 32 SUBTRG_I d’entrata) in trigger di secondo livello (gli 8 TRG_O d’uscita).

```
generic (base_addr : integer :=0);
port(       SUBTRG_I : in std_logic_vector(NBIT_SUBTRIG-1 downto 0);
            TRG_O : out STD_LOGIC_VECTOR(NBIT_TRIG-1 downto 0);
            CLK : in STD_LOGIC;
            RST : in STD_LOGIC;
            ----------------------------BUS
            address : in STD_LOGIC_VECTOR(NBIT_ADDR - 1 downto 0);
            data_in : in STD_LOGIC_VECTOR(NBIT_DATAIN - 1 downto 0);
            data_out: out STD_LOGIC_VECTOR(NBIT_DATAOUT -1 downto 0);
            n_rd : in  STD_LOGIC;
            n_wr : in  STD_LOGIC;
            selector: in std_logic
            ----------------------------
    );
```

Listato 3.6 Interfaccia Logic Matrix

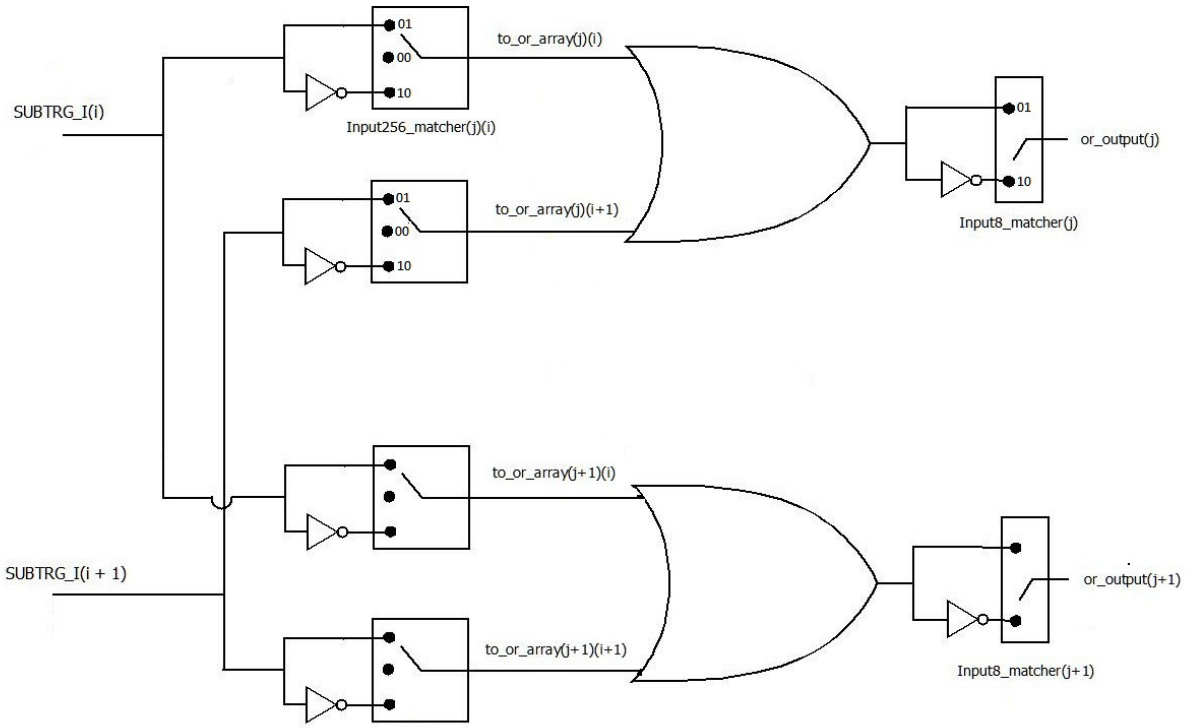


Fig 3.5 Schema logico del modulo Logic Matrix

Grazie alle ben note leggi di De Morgan [12] possiamo ottenere qualsiasi funzione logica che presenti operatori come *AND*, *OR* e *NOT* tramite le sole operazioni *OR* e *NOT.*
Come si evince dalla figura 3.5, dati due ingressi che rappresentano i segnali da trattare, asserendo o negando tali ingressi, applicando a essi l'operazione *OR* e nuovamente asserendo o negando tale segnale, avremo in uscita il risultato della funzione logica richiesta. Nel modulo, i 32 ingressi vengono copiati 8 volte per produrre 8 funzioni logiche distinte. Per ottenere ciò si è reso necessario l'uso di tre strutture dati:

```
1. type input256_decision_array is array(0 to (NBIT_SUBTRIG * NBIT_TRIG) - 1 )of
   std_logic_vector(1 downto 0);

   signal input256_matcher : input256_decision_array;
2. type input8_decision_array is array(NBIT_TRG - 1 downto 0) of
   std_logic_vector(1 downto 0);
   signal input8_matcher : input8_decision_array;

3. type to_or_array_signal is array (0 to NBIT_TRG - 1) of std_logic_vector
   (NBIT_SUBTRIG - 1  downto 0);
   signal to_or_array : to_or_array_signal;
```

Listato 3.7 Strutture dati della Logic Matrix

input256_matcher è un array di 256 registri a 2 bit, ciascun registro è un segnale di tipo **std_logic_vector.** Tale array serve a memorizzare, per ciascuna delle 8 equazioni logiche, il codice che indica quale operazione logica applicare ai 32 subtrigger di ingresso (asserire, negare o non considerare il segnale).
To_or_array è un array di (8 *32) segnali che raccoglie i segnali opportunamente modificati, che fanno da ingresso agli 8 OR.
Input8_matcher è un array di 8 registri da due bit. Il codice contenuto in ciascun registro stabilisce se il risultato dell'OR dei 32 segnali to_or_array(i) debba essere negato o lasciato invariato. La negazione è utile nel caso si voglia realizzare un AND negando l'OR degli ingressi a loro volta negati.
Il ciclo che permette di calcolare gli ingressi per gli 8 or, è riportato nel listato 3.8:

```
or_input_decisor:for j in nbit_trig-1 downto 0 loop
          FOR I IN NBIT_SUBTRIG-1 DOWNTO 0 LOOP
                                case (CONV_INTEGER(input256_matcher((32*j)+i)))is
                                                    when 1 => to_or_array(j)(i) <=
subtrg_i(i);                                                                  when 2 =>
to_or_array(j)(i) <= (not subtrg_i(i));
   when others => to_or_array(j)(i) <= '0';
          end case;
          end loop;
   end loop;
```

Listato 3.8 modifica segnali di entrata

Il costrutto case ha come espressione da valutare il codice presente nell'**input256_matcher**: il valore binario "01" copia in to_or_array l' i-esimo segnale senza modificarlo, il valore binario "10" lo inverte prima di copiarlo nell'elemento di to_or_array, mentre "00" e "11" forzano a '0' il segnale. Il segnale corrispondente all'i-esimo segnale dei subtrigger, eventualmente negato, è associato all' i-esimo segnale dell'array **to_or_array** per ognuna delle 8 funzioni logiche che ci prefiggiamo di calcolare, individuate dall'indice j.
Una volta impostati correttamente gli ingressi possiamo calcolarne l'*OR* confrontandoli con un array di segnali, *compare_to,* inizializzato a '0': l'uscita di ciascuna funzione assumerà il valore logico '1' quando almeno uno degli ingressi è 1 (listato 3.9):

```
or_cycle:FOR J IN NBIT_TRIG-1 DOWNTO 0 LOOP
                if (to_or_array(j) = compare_to)then
                       or_output(j) <= '0';
                else or_output(j) <= '1';
                end if;
          end loop;
```

Listato 3.9 Or dei segnali

Un ulteriore *loop* consente invece di completare la funzione logica asserendo o negando ciascun degli 8 risultati dell'*OR* precedente sulla base del contenuto dei registri input8_:atcher:

```
output_decisor: for i in (NBIT_TRG – 1) downto 0 loop
   case (CONV_INTEGER(input8_matcher(i))) is
   when 1 => trg_o(i) <= or_output(i);
   when 2 => trg_o(i) <= not or_output(i);
   when others => trg_o(i) <= '0';
          end case;
end loop;
```

Listato 3.10 Loop finale

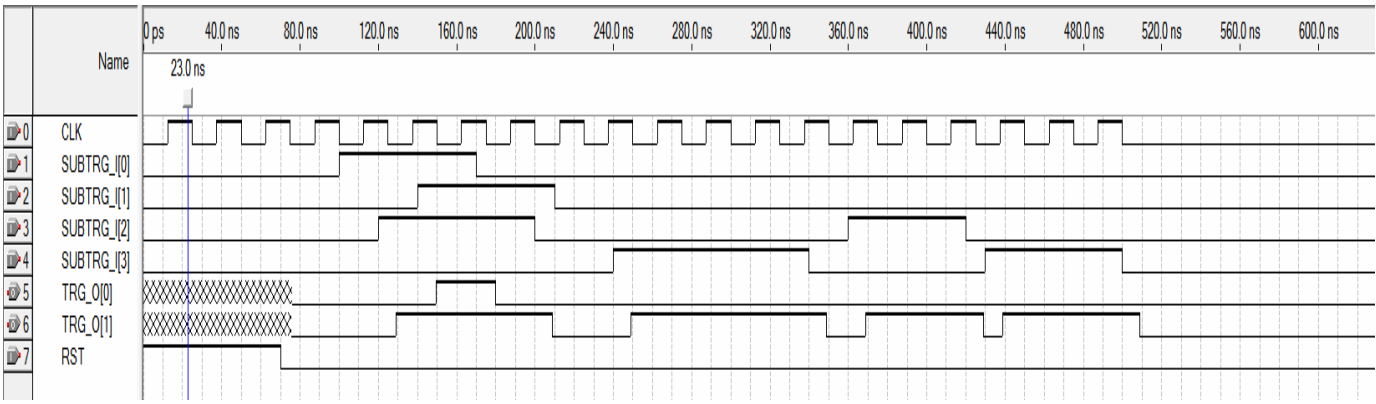
Ognuna delle 8 uscite (**TRG_0)** quindi sarà una diversa funzione dei 32 subtrigger d'ingresso.
E' naturalmente previsto un sistema per impostare i codici da associare a ogni segnale tramite il bus di cui proponiamo un listato esemplificativo riguardante l'impostazione dei codici per l'**input256_matcher**:

```
read_AND_WRITE256: for i in 0 TO (NBIT_SUBTRIG*NBIT_TRIG)-1 loop

          IF((to_integer(unsigned(address)) = (base_addr+ i))AND N_RD ='1') then

                data_out(1 downto 0) <= input256_matcher(i);

          ELSIF ((to_integer(unsigned(address)) = (base_addr+ i))AND N_WR ='1')THEN
                INPUT256_MATCHER(I) <= DATA_IN( 1 DOWNTO 0);
          END IF;
end loop;
```

Listato 3.11 Scrittura o lettura codice di modifica input256_matcher

I *subtrigger*, in ingresso alla Logic Matrix, diventano così trigger di secondo livello in uscita e verranno in seguito chiamati *trigger* .

Simulazione Logic Matrix:



Simulazione 2: Logic Matrix

Nella simulazione 2, riportata in figura, la Logic Matrix è stata configurata in modo che l'uscita TRG_O[0] sia ottenuta dall' AND dei primi due ingressi, SUBTRG_I[0] e SUBTRG_I[1]. L'uscita TRG_O[1] è invece l'OR degli ingressi SUBTRG_I[2] e SUBTRG_I[3]. Notare che il segnale AND sulla prima uscita ha una durata uguale alla sovrapposizione dei valori logici '1' dei primi due ingressi.

3.2.3 VETO (Busy)

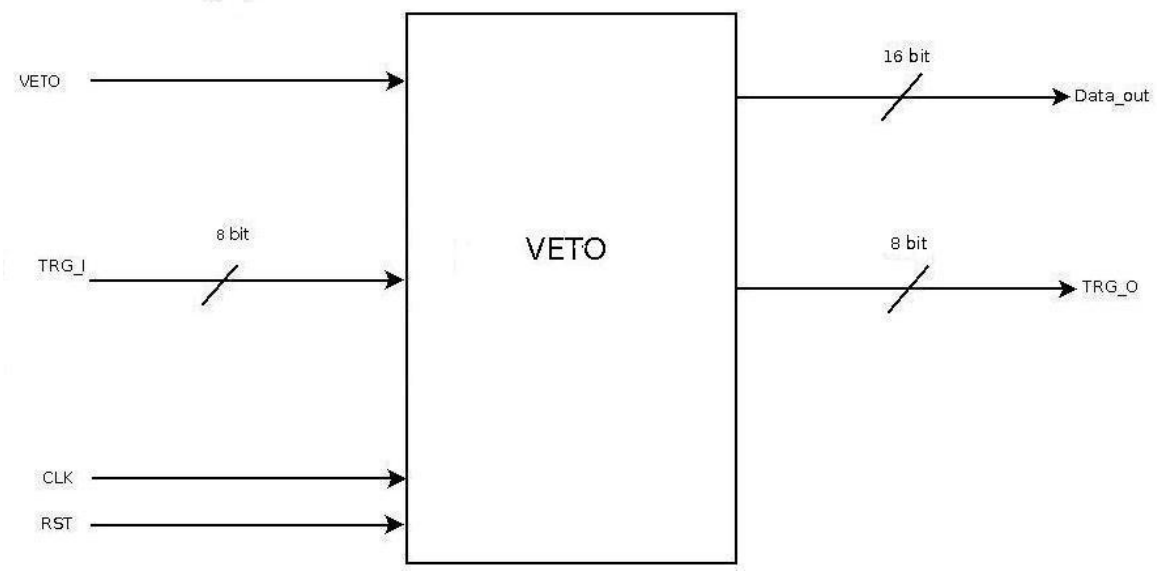


fig. 3.6 VETO (Busy) MODULE

```
entity Busy is

    port(TRG_I : in STD_LOGIC_VECTOR(NBIT_TRIG-1 downto 0);
           TRG_O : out STD_LOGIC_VECTOR(NBIT_TRIG-1 downto 0);
            CLK : in STD_LOGIC;
            RST : in STD_LOGIC;
            VETO_SIGNAL : in STD_LOGIC
            );
    end Busy;
```

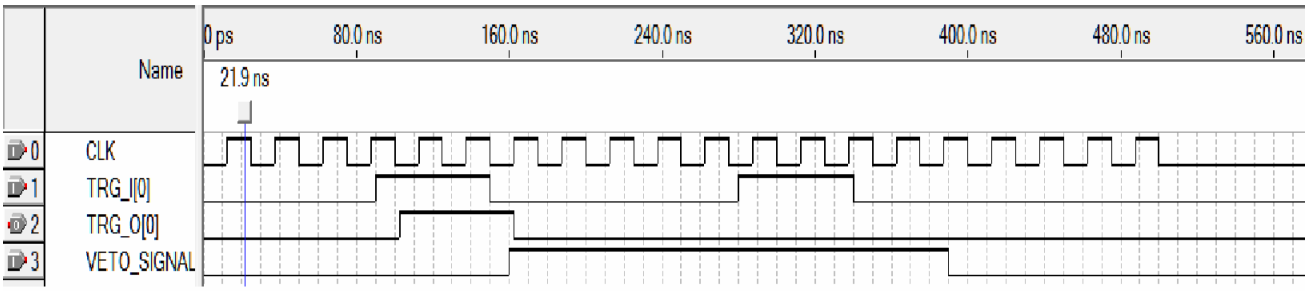
Listato 3.12: Dichiarazione del modulo Veto (Busy)

Il listato 3.12 si riferisce all'entity vhdl del modulo **Veto**. Esso presenta complessivamente 11 segnali di input che sono nell'ordine gli 8 *TRG_I*, corrispondenti agli 8 TRG_O in uscita dalla Logic Matrix, *CLK, RST* e *VETO_SIGNAL* e 8 segnali di output *TRG_O.*
Il Veto può interrompere il flusso di trigger di secondo livello qualora l'apparato sperimentale, sollecitato da un precedente **MAINTRG,** sia in una fase di acquisizione e elaborazione dei dati, fase in cui non è opportuno far affluire ulteriori trigger di secondo livello. Una volta riconosciuto il verificarsi del **MAINTRG**, che indica all'apparato sperimentale la possibilità di acquisizione dei dati, il sistema d'acquisizione asserisce il segnale VETO_SIGNAL in modo tale da "bloccare" i TRIG_I in entrata. Come si nota dal listato 3.13, la logica di questo modulo è molto semplice: se VETO_SIGNAL è vero, tutte le uscite sono forzate al valore falso, altrimenti sono poste uguali ai corrispondenti ingressi.

```
process(clk,rst)
begin
if rising_edge(clk)then
    if (Veto_signal = '1')then
           Signal_Block: for i in 0 to NBIT_TRIG-1 loop
                        TRG_O(i) <= '0';
           end loop;
           elsif(veto_signal = '0')then
                  TRG_O <= TRG_I;
           end if;
end if;
end process;
```

Listato 3.13 descrizione comportamentale del modulo Veto (Busy)

Simulazione Veto:



Simulazione 3: Veto

La simulazione 3 ci mostra che quando il segnale *VETO_SIGNAL* è attivo, all' uscita TRG_O[0]il valore logico è sempre '0', anche quando TRG_I[0] è '1'

3.2.3 DOWNSCALER

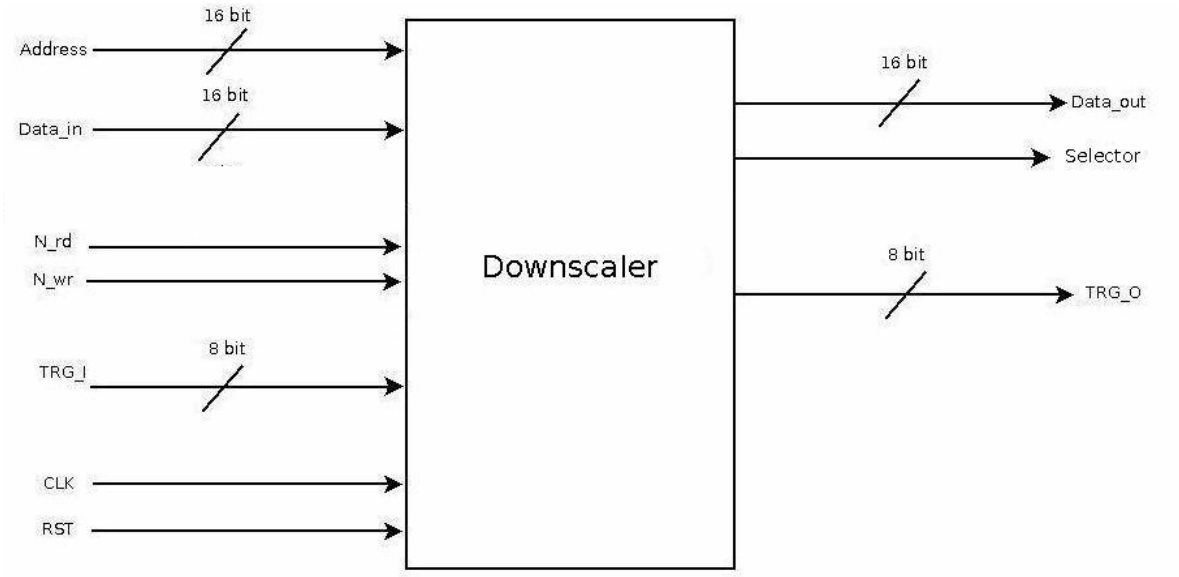


fig 3.7 Downscaler Module

```
entity DownScaler is

    port(TRG_I : in STD_LOGIC_VECTOR(NBIT_TRIG-1 downto 0);
            TRG_O : out STD_LOGIC_VECTOR(NBIT_TRIG-1 downto 0);
            RST : in STD_LOGIC;
            CLK :in Std_Logic;
           ----------------------------BUS
           address : in STD_LOGIC_VECTOR(NBIT_ADDR-1 downto 0);
           data_in : in STD_LOGIC_VECTOR(NBIT_DATAIN-1 downto 0);
           data_out: out STD_LOGIC_VECTOR(NBIT_DATAout-1 downto 0);
           n_rd : in  STD_LOGIC;
           selector: out  STD_LOGIC;
           n_wr : in  STD_LOGIC
           );

end DownScaler;
```

Listato 3.14 Dichiarazione Entity Downscaler.

Il modulo Downscaler - la cui dichiarazione è presente nel listato 3.14 - è preposto alla riduzione, secondo un certo fattore, della frequenza dei segnali di trigger provenienti dall'input. I fattori di riduzione sono impostabili separatamente per gli 8 trigger di ingresso mediante 8 registri da 16 bit;

```
entity Counter is
    port(         clock     : in std_logic;
                  reset     : in std_logic;
                  clock_in  : in std_logic;
                  value_out : out std_logic_vector(NBIT_DEBUG-1 downto 0)
          );

end Counter;
```

Listato 3.15 Dichiarazione dell'entity Counter.

Ogni trigger di ingresso è inviato all'ingresso di clock (**clock_in**) di un contatore . L'entity di un singolo contatore è mostrata nel listato 3.15. Il processo che implementa il contatore è mostrato nel listato 3.16: il valore attuale del conteggio è memorizzato nella variabile temp, che viene "scritta" ad ogni clock nel segnale counter.

```
signal counter : Integer :=0;
count: process(clock,reset)

variable temp : Integer:=0;
variable old_value : Integer:= 0;
begin
    if (reset = '1')then
            temp := 0;
            counter <= 0;
            old_value := 0;

          elsif (clock'event and clock = '1' )then

                if(clock_in= '1' and old_value = 0)then
                      old_value := 1;
                      temp := temp + 1;

                elsif(clock_in= '0' and old_value = 1) then
                    old_value := 0;
                end if;

           counter <= temp;
          end if;

    end process count;
value_out <= std_logic_vector(to_unsigned(counter,NBIT_DATAOUT));
```

Listato 3.16 Processo del contatore

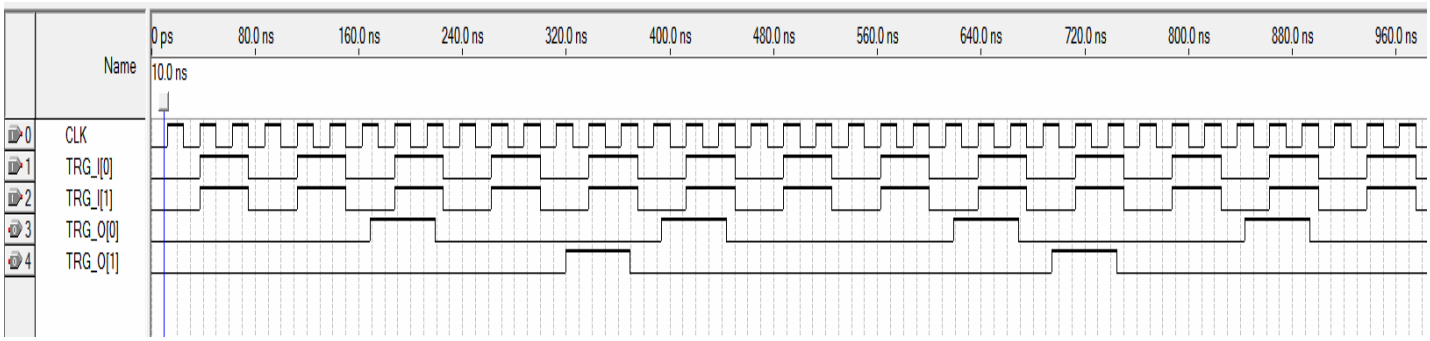
Tale valore viene aggiornato ad ogni transizione 0 → 1 del clock ed è sempre presente all'uscita value_out.
All'interno del process principale del Downscaler, Il numero delle transizioni 0 → 1 dell'ingresso, ottenuto grazie al contatore, viene confrontato con il fattore di riduzione ad ogni fronte ascendente del clock CLK (istruzione IF nel listato 3.17):

```
if(rst = '1')then
          CICLO_reset:FOR I IN NBIT_TRIG-1 DOWNTO 0 LOOP
                counter_reset(i) <= '1';
                trg_o(i) <= '0';
                data_out(i) <= '0';
          end loop;

    elsIF RISING_EDGE(CLK)THEN
          CICLO:FOR I IN NBIT_TRIG-1 DOWNTO 0 LOOP
                counter_reset(i) <= '0';
                trg_o(i) <= '0';
                IF(CONV_INTEGER((To_matcher(i)))) = CONV_INTEGER(div_factor(i)))THEN
                      trg_o(i)<= '1';
                      counter_reset(i) <= '1';
                end if;
          end loop;
    end if;
```

Listato 3.17 Processo del Downscaler.

div_factor è un array di segnali che raccoglie i fattori di divisione (impostabili dall'utente tramite il bus). L'array di segnali **to_matcher** raccoglie in un unico array le uscite value_out dei contatori. Ciascun conteggio, contenuto in to_matcher(i) è confrontato con un corrispondente div_factor(i). In caso di uguaglianza, il segnale di uscita corrispondente verrà asserito e sarà azzerato il corrispondente contatore.

Simulazione Downscaler:



Simulazione 4: Downscaler

Nella simulazione del Downscaler riportata in figura, è stato impostato un fattore di riduzione 2 per il bit 0 di TRG_I e un fattore 4 per il bit 1. Si noti come il segnale TRG_O[0] viene asserito dopo la transizione di due segnali nel rispettivo TRG_I[0] e invece il TRG_O[1] dopo quattro transizioni nel rispettivo TRG_I[1].

3.2.4 TRIG AND PATTERN

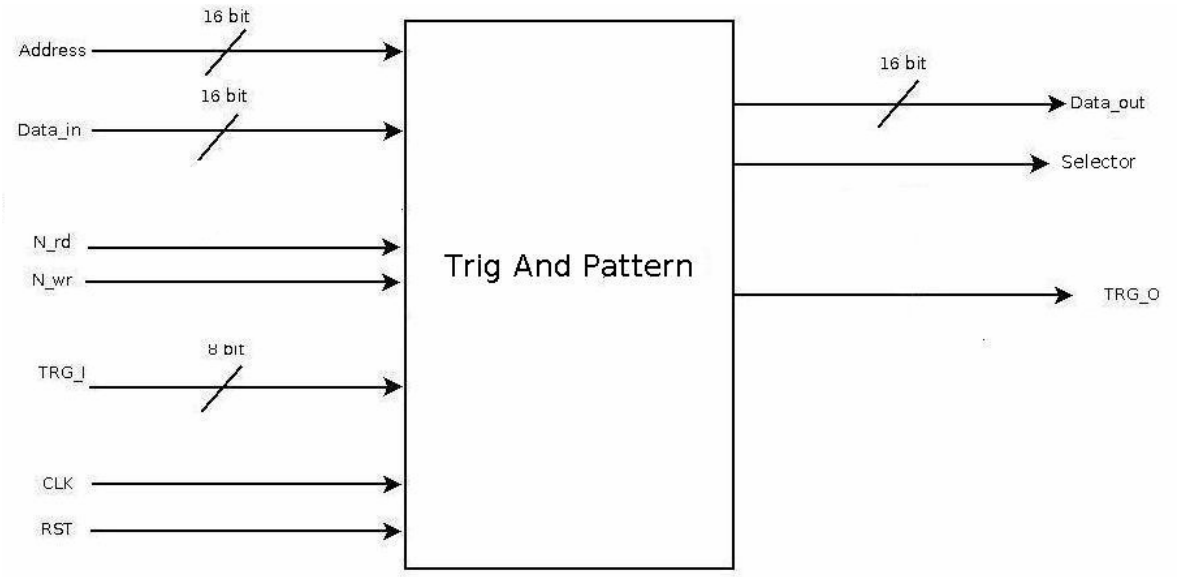


Fig 3.6 Trig And Pattern Module

```
entity Trg_And_Pattern is
    generic(base_addr : integer := 0);
    port(  TRG_I : in STD_LOGIC_VECTOR(Nbit_trig -1 downto 0);
           TRG_O : out STD_LOGIC;
           CLK : in STD_LOGIC;
           RST : in STD_LOGIC;
           ----------------------------BUS
           address : in STD_LOGIC_VECTOR(NBIT_ADDR - 1 downto 0);
           data_in : in STD_LOGIC_VECTOR(NBIT_DATAIN - 1 downto 0);
           data_out: out STD_LOGIC_VECTOR(NBIT_DATAOUT - 1 downto 0);
           n_rd : in  STD_LOGIC;
           n_wr : in  STD_LOGIC;
           selector: out  STD_LOGIC
         ------------------------------
           );
end Trg_And_Pattern;
```

Listato 3.10 dichiarazione dell'entity Trg_And_Pattern

Il modulo Trig and Pattern è (come risulta visibile nello schema generale della trigger box nella figura 3.1) l'ultimo nella catena di moduli che gestiscono i trigger. Esso genera il segnale di trigger principale, in pratica l'*OR* dei trigger in ingresso, e memorizza i segnali di trigger che hanno dato luogo al **MAINTRG.** La "fotografia", memorizzata in una variabile apposita, dei bit rappresentanti i trigger di secondo livello verrà poi consultata tramite interrogazione del bus. L'informazione sul "pattern" dei trigger che ha prodotto l'acquisizione, memorizzata evento per evento, è molto utile nella

successiva fase di analisi dei dati dell'eseprimento. La funzione del modulo è realizzata mediante il processo riportato nel listato 3.11.

```
clocked:process(clk)
   begin
          if (RST = '1')then
                mainout <=    '0';

          elsif rising_edge(clk) then
                if (trg_i = compare_to) then
                      mainout <= '0';
                else mainout <= '1';

          if((to_integer(unsigned(address)) = (base_addr))AND N_RD ='1') then
                      data_out(NBIT_TRIG-1 downto 0) <= bit_pattern;
          end if;

end process clocked;
```

Listato 3.11 clocked process.

Esso si occupa di ottenere l'OR degli 8 trigger di secondo livello, usando la stessa implementazione presente nel listato 3.7 della Logic Matrix. L'uscita dell'or (chiamata **mainout**) sarà quindi asserita quando uno dei trg_i avrà valore logico '1'; tale uscita rappresenterà il trigger principale della Trigger Box .
Naturalmente è anche previsto l'accesso ai trigger che hanno contribuito alla generazione del trigger principale, grazie all'accesso al registro **bit_pattern** che memorizza i valori dei trg_i.
Il listato successivo presenta la logica che permette, sul fronte di salita del trigger principale, di memorizzare in bit_pattern i trig_I che hanno prodotto il trigger di uscita:

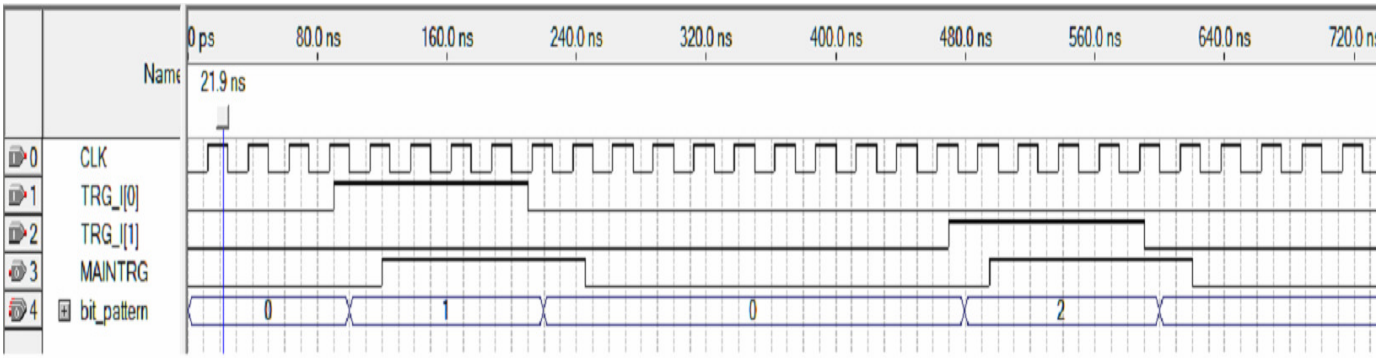
```
   trg_o <= mainout;

   latching:process(mainout)
          begin

                if rising_edge(mainout)then
                      bit_pattern <=trg_i;

                end if;
   end process latching;
```

Listato 3.10 particolare del codice Trig And Pattern.

Il segnale bit_pattern, che viene aggiornato sul fronte del segnale mainout, si comporta a tutti gli effetti come un registro, composto da tanti flip flop di

tipo D quanti sono i trigger. I trig_I costituiscono gli ingressi dei flip-flop e mainout è il loro clock comune

Simulazione Trig And Pattern:



Simulazione 5: Trig and Pattern

La simulazione 5 mostra la risposta del modulo a due ingressi TRG_I[0] e TRG_I[1].
Il segnale *MAINTRG* viene asserito ogni qualvolta ci sia una transizione 0 → 1 su uno degli ingressi, mentre lo std_logic_vector *bit_pattern* memorizza, per ciascuna transizione 0 → 1 del MAINTRG, il valore della parola TRG_I.

3.2.5 Multiplexer

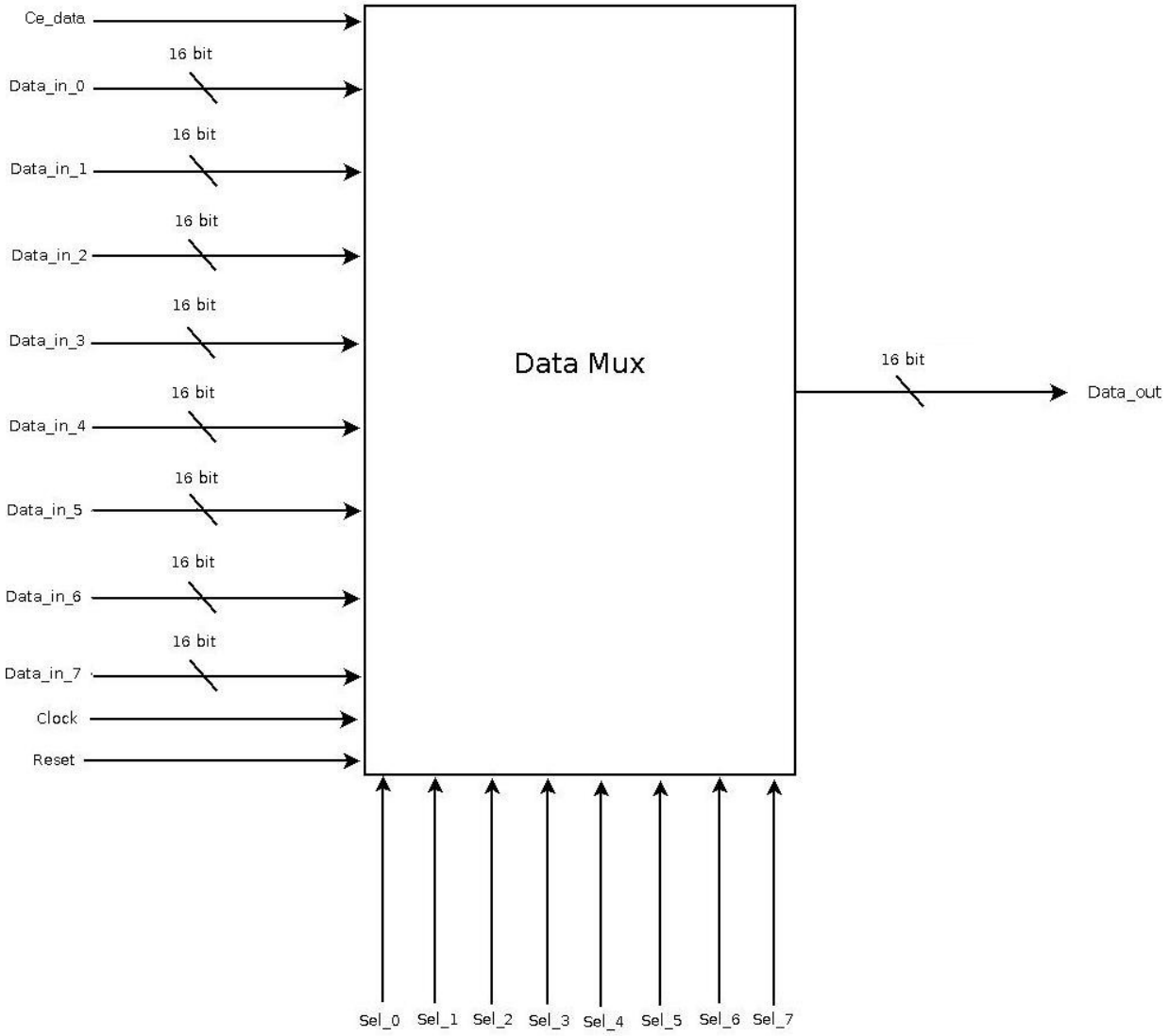


Fig 3.7 Data_Mux module

Come abbiamo spiegato all'inizio della sez 3.2, il BUS che permette lo scambio di dati con una CPU esterna ha bisogno di un modulo multiplexer per i dati in uscita della trigger box.

Il problema di selezione di segnali si presenta sempre in elettronica digitale quando due o più moduli devono condividere uno stesso canale di comunicazione o BUS. Il modulo che implementa il multiplexer è **Data_Mux**. Il funzionamento è basato sul semplice processo del listato 3.10:

```
architecture behave of data_mux is
begin
 mux_process : process(clock)
 begin
      if ce_data = '1' then
           if sel_1 = '1' then
              data_out <= data_in_1;
           elsif sel_2 = '1' then
              data_out <= data_in_2;
           elsif sel_3 = '1' then
              data_out <= data_in_3;
           elsif sel_4 = '1' then
              data_out <= data_in_4;
           elsif sel_5 = '1' then
              data_out <= data_in_5;
           elsif sel_6 = '1' then
              data_out <= data_in_6;
           elsif sel_7 = '1' then
              data_out <= data_in_7;
           else
              data_out <= data_in_0;
           end if;
      end if;
   end process;
end behave;
```

Listato 3.10 Descrizione comportamentale Data_Mux.

Il multiplexer ha 8 segnali di ingresso (da **Data_in_0** a **Data_in_7**), a 16 bit ciascuno, associati ai data_out provenienti dai vari moduli, un clock, un reset, un segnale che abilita l'uscita (**ce_data)** e un segnale di uscita (**Data_out).** Ogni segnale di selezione (da **sel_0** a **sel_7**) è associato a un particolare modulo: sarà quindi uno di questi 8 segnali che decideranno quale dei Data_in verrà pilotato sul Data_out del multiplexer. Quando un qualsiasi modulo riconosce sul bus l'indirizzo di un proprio registro ed insieme il segnale n_rd, deve porre sul proprio data_out il valore del registro ed asserire il proprio segnale selector. Il segnale selector, connesso a uno dei sel_0 – sel_7 del DataMux, commuta il multiplexer in modo che il data_out del modulo piloti il data_out in uscita della Trigger Box.

3.2.6 Scaler

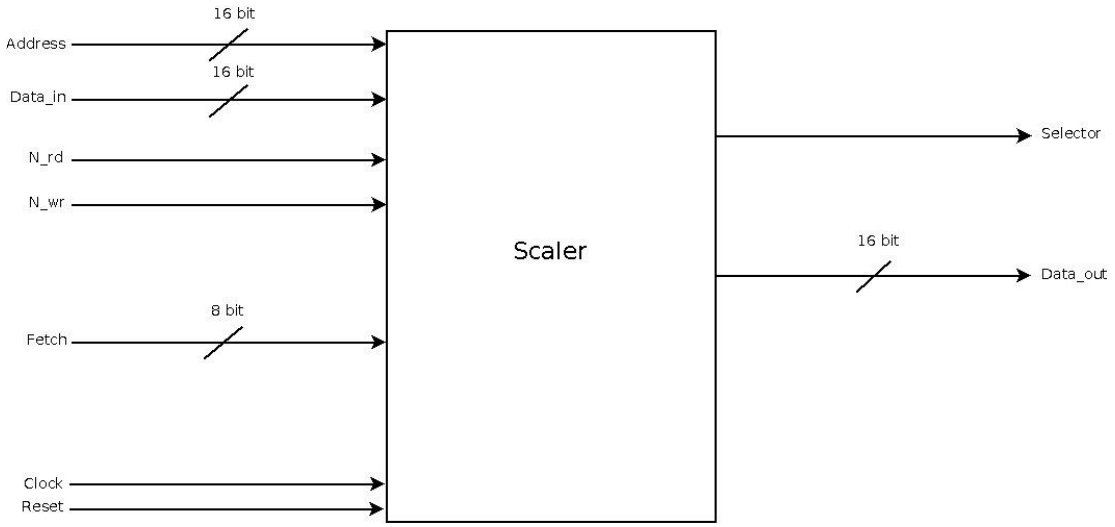


Fig. 3.8 Interfaccia Scaler

Un ulteriore funzione svolta dalla trigger box è quella di contare i segnali di trigger di secondo livello in uscita dalla Logic Matrix. Allo scopo di tenere sotto controllo il sistema di acquisizione, è utile conoscere la frequenza dei trigger sia prima che dopo il modulo Veto, oltre che all'ingresso del Trig and Pattern, cioè dopo il downscaling. La funzione di contare le transizioni 0 → 1 dei segnali di trigger è affidata al modulo Scaler. Tre moduli di questo tipo prelevano i segnali da contare rispettivamente prima del veto, fra il veto e il Downscaler e dopo il Downscaler, come si vede in figura 3.1. Un esempio delle informazioni ottenibili grazie agli scaler è il cosiddetto *tempo morto* del sistema di acquisizione, espresso come la frazione di trigger che sono stati ignorati perché giunti quando il veto era vero. Esso si stima dalla formula (N – M) /N x 100 dove N sono i conteggi dello scaler pre-Veto e M sono quelli dello scaler post-Veto.

```
entity Scaler is
     generic(base_addr: integer :=  0);
     port(clock         :in  STD_LOGIC;            -- Ingresso di clock.(rise edge.)
          reset         :in  STD_LOGIC;            -- Ingresso di reset (att. alto.).
          fetch         :in  std_logic_vector(NBIT_TRIG-1 downto 0);
                        ----------------------------BUS
          address : in STD_LOGIC_VECTOR(NBIT_ADDR-1 downto 0);
          data_in : in STD_LOGIC_VECTOR(NBIT_DATAIN-1 downto 0);
          data_out: out STD_LOGIC_VECTOR(NBIT_DATAOUT-1 downto 0);
          n_rd : in  STD_LOGIC;
          n_wr : in  STD_LOGIC;
          selector: out  STD_LOGIC
          );
```

```
end Scaler;
```

Listato 3.11 dichiarazione scaler

Nel listato 3.11 è riportata la entità dello Scaler.
Oltre ai segnali d'entrata clock e reset, abbiamo un segnale di 8 bit, **fetch**, che viene connesso al bus di sola lettura che collega i vari moduli (connessioni in rosso nella fig 3.1). All'interno di ciascun scaler sono istanziati gli 8 contatori (vedi listato 3.12) del tipo impiegato anche per il Downscaler. I conteggi dei vari contatori possono essere letti mediante il bus della trigger box, grazie al processo di decodifica di indirizzo, mostrato nel listato 3.13.
Il listato seguente riporta l'istanziazione degli 8 counter:

```
-------------------------------Instantiate Counter
contatori: for kcounter in NBIT_TRIG-1 downto 0 generate
      conta : Counter

      port map(  clock => clock,
                 reset => reset,
                 clock_in  => fetch(kcounter),
                 value_out => cnt(kcounter)
               );
end generate contatori;
---------------------------------
```

Listato 3.12 dichiarazione componenti

```
selection: process(clock,reset)
   begin
   if (reset = '1')then
          selector <='0';
   elsif rising_edge(clock)then
          if(n_rd = '1')then
                 decode_addr: for i in 0 to NBIT_TRIG-1 loop
                        if(to_integer(unsigned(address)) = (base_addr+ i)) then
                               selector<= '1';
                               data_out <=cnt(i);
                        end if;
                 end loop;
                 else selector <='0';
          end if;
   end if;
end process selection;
```

Listato 3.13 decodifica indirizzo

CONCLUSIONI

In questo lavoro è stato realizzato un dispositivo per la gestione dei "trigger" (Trigger Box) in un sistema di acquisizione utilizzato in esperimenti di fisica nucleare. In passato le funzioni realizzate da tale Trigger Box erano ottenute mediante dispositivi elettronici discreti interconnessi fra loro, che implementavano tutte o quasi le funzioni previste in questa tesi. Grazie alla tecnologia FPGA è stato possibile integrare tali funzioni su un unico dispositivo logico programmabile. Tale dispositivo (il chip Cyclone EP1C20F400C6) è montato su una scheda prodotta dalla ditta CAEN di Viareggio che permette sia di programmare il dispositivo sia di accedere ai suoi registri tramite il bus industriale VME: in questo modo oltre a eliminare la complicazione delle connessioni esterne fra moduli discreti è anche possibile ottenere una piena programmabilità di tutte le funzioni della Trigger Box tramite una CPU inserita sullo stesso bus.
Il codice che descrive le funzioni logiche della Trigger Box è stato scritto in VHDL (VHSIC Hardware Description Language).
Il compito principale della Trigger Box è ottenere da un numero elevato di ingressi logici di subtrigger un certo numero di segnali di trigger con i quali attivare l'acquisizione degli eventi di interesse dal punto di vista fisico.
Per mezzo delle funzioni sviluppate nel codice, si possono definire:

- La durata dei segnali di subtrigger (Resolving Time).
- Le funzioni logiche da calcolare sulla base dei subtrigger per ottenere i trigger (Logic Matrix).
- Il tempo morto del sistema di acquisizione (Veto).
- L'eventuale riduzione della frequenza dei segnali di trigger (Downscaling).

Inoltre la Trigger Box genera il segnale di Main Trigger per il sistema di acquisizione e incorpora dei dispositivi per la misura della frequenza dei diversi segnali di trigger (Scaler).

Questo lavoro è stato svolto nell'ambito del gruppo di Fisica del Nucleo del Dipartimento di Fisica dell'Istituto Nazionale di Fisica Nucleare (sezione di Firenze). Una volta realizzata la programmazione, il gruppo ha verificato il corretto funzionamento della Trigger Box una volta programmata sulla scheda CAEN V1495. Dato che la verifica ha avuto esito positivo, il gruppo intende utilizzare il sistema negli esperimenti di prossima realizzazione.

Bibliografia

[1] THE VMEbus SPECIFICATION, Motorola, 1985

[2] CAEN V1495 General Purpose VME Board- Technical Information Manual, CAEN, 2009

[3] G. F. Knoll, Radiation Detection and Measurement, 3rd ed., Wiley 2000

[4] M. Bini et al., Nucl. Instr. And Meth. A 515 (2003) 497G.

[5] http://it.wikipedia.org/wiki/Programmable_logic_device

[6] R. C. Jaeger, Microelectronic Circuit Design, Mc Graw Hill, 1997

[7] Cyclone Device Handbook, Altera (disponibile su www.altera.com)

[8] http://en.wikipedia.org/wiki/Hardware_description_language

[9] L. Cardini, Realizzazione di filtri digitali mediante dispositivi logici programmabili per il trattamento di segnali generati dai rivelatori, tesi di laurea in informatica, Università di Firenze, 2004

[10] http://en.wikipedia.org/wiki/VHDL

[11] J. Millman & A. Grabel, Microelectronics, Mc Graw Hill 1987

[12] L. Carlucci Aiello, F. Pirri, “Strutture, logica, linguaggi”