UNIVERSITA' DEGLI STUDI DI GENOVA

Facoltà di Ingegneria

Studio e analisi di algoritmi di corner detection per architetture di calcolo basate su dispositivi a logica programmabile

Tesi di laurea di:

Enrico Patrone

Massimo Schenone

Relatore: Chiar.mo Prof. Ing. G. Cannata

Università degli Studi di Genova
DIST - MACLAB

Indice

Indice……..1

Capitolo1……...…12

Introduzione……12

Capitolo 2……..……14

Stato dell'arte…… …14

2.1 Features…….21

2.2 Algoritmi per il rilevamento dei corner…….25

2.2.1 Smith (S-USAN) Corner Detector……...25

2.2.2 Kitchen-Rosenfeld Corner Detector……28

2.2.3 Harris Corner Detector……29

Capitolo 3……31

Le FPGA……...…31

3.1 Introduzione……..…31

3.2 I blocchi logici……..32

3.3 La scelta della FPGA da utilizzare……35

3.4 Le interconnessioni……36

3.5 La metodologia seguita per l'analisi……..37

3.6 L'ambiente di sviluppo Xilinx ISE 6.3 I……...37

Capitolo 4……..…41

Telecamera……..…..41

4.1 Introduzione……..….41

4.2 Videocamera digitale OV7640…….….42

4.2.1 Specifiche……44

4.2.2 Modalità operative……..46

4.2.3 Codifica Pixel……...47

Capitolo 5……...…....49

Implementazione dei filtri……...……49

5.1 Introduzione……49

5.2 Realizzazione filtro con Simulink……...….52

5.3 Realizzazione filtro in aritmetica a precisione finita……...57

5.3.1 Generalità…….57

5.3.2 Realizzazione filtri……...58

Capitolo 6…….……..61

Corner detection di Forstner…….…….61

6.1 Analisi dell'immagine……..61

6.2 Calcolo del gradiente di un'immagine…….……63

6.3 Filtro di Sobel……...…65

6.4 Schema dell'algoritmo di Forstner……..66

6.5 Risultati simulativi……..……69

6.5.1 Confronto con un algoritmo di corner detection interamente sviluppato via software…….…..….71

Capitolo 7……..……72

Simulazione dell'algoritmo di Forstner con aritmetica a precisione finita…….…..….72

7.1 Dati in uscita dalla videocamera……..……73

7.2 Filtraggio preliminare…….…..…73

7.3 Calcolo delle componenti del gradiente…….….77

7.4 Prodotti delle componenti del gradiente…….77

7.5 Filtraggio gaussiano…….....78

7.6 Calcolo della corneress……...…….79

7.7 Risultati simulativi……...81

7.7.1 Simulazioni su immagini preparate ad-hoc……82

7.8 Conclusioni……..84

Capitolo8……...86

Simulazione con Xilinx System Generator……....86

8.1 Introduzione……86

8.2 Primo approccio simulativo……86

8.3 Prove su immagini ad-hoc per la definizione dei dati……..……..…88

8.4 Analisi del blocco di calcolo della corneress…….……...93

8.5 Latenza totale del circuito……94

8.6 Sogliatura dell'uscita……...95

8.6.1 Risultati simulativi con soglia……96

Capitolo 9……..98

Interfaccia per videocamera OV7640……98

9.1 Interfaccia per formato YUV 4:2:2..……....98

9.2 Interfaccia per formato RGB 555..……....100

Capitolo 10……..….104

Conclusioni e sviluppi futuri…….……104

Appendice A……...…107

I formati colori…….107

Appendice B……115

Rappresentazione Fixed-Point su Matlab……...…….115

B.1 Generalità……...115

B.2 Tool Fixed-Point Blockset……...……118

B.3 Rappresentazione numeri con segno…….121

Appendice C……...…123

Guida a Xilinx System Generator……123

Appendice D……...……129

Software utilizzati……...….129

Famiglie *Xilinx*…….130

BIBLIOGRAFIA…….……132

Indice figure

Figura 2.1 : immagine di ingresso (T), filtro (W), immagine di uscita (T')……..15
Figura 2.2 : esempio di generazione di uscita di due *pixels* consecutivi……..……16
Figura 2.3: esempio che mostra che solo (*n*-1)righe precedenti più *n pixels* devono essere memorizzati……17
Figura 2.4: grafico della complessità computazionale al variare di n……...….18
Figura 2.5: vengono visualizzate nell'immagine i tre tipi di *feature*: una regione uniforme, un contorno(*edge*) e un *corner*……..21

Figura 2.6: (a) una regione di immagine smoothed, (b) i valori di intensità e (c) il grafico della funzione di intensità luminosa I(*x*,*y*)……...……22

Figura 2.7: un bordo (a), (b) i valori di intensità, (c) il grafico della funzione di intensità luminosa I(*x*,*y*). Si noti la variazione dell'intensità in una sola direzione……23

Figura 2.8: un corner(a) dell'immagine del tipo "giunzione a L": (b) mostra la variazione dei valori d'intensità luminosa attorno al punto e (c) è il grafico della funzione intensità di colore I(*x*,*y*). Si noti la variazione dell'intensità in due direzioni……24

Figura 2.9: maschere per il ritrovamento dei corner in diverse posizioni di un'immagine: (a) nucleo posizionato esattamente sul corner, la USAN è un quarto dell'area (b) rileva un edge (c) in una regione uniforme i pixel hanno per lo più la stessa intensità (d) la maschera così posizionata, secondo la soglia di Smith, non rileva ancora il corner, perché la USAN è troppo grande……26

Figura 2.10: La maschera di Smith per la ricerca delle USAN. Si notano il nucleo (in nero, il pixel centrale), il quadrato 5x5 (in grigio scuro), e i tre pixel per lato (in grigio chiaro) per l'approssimazione dell'area circolare……...…….27

Figura 3.1: confronto tempi di sviluppo FPGA-ASIC……..32
Figura 3.2: Schema a blocchi di una FPGA generica……...…….34
Figura 3.3: visualizzazione *Project Navigator*…….…….39
Figura 4.1: matrice che rappresenta l'immagine……41
Figura 4.2: videocamera digitale OV7640……43
Figura 4.3: dati di targa OV7640……...…43
Figura 4.4: microcontrollore per il controllo della videocamera……...……45
Figura 4.5: OVTDTool è usato per leggere e scrivere in specifici registri all'interno della *camerachip* OV7640…….……..46

Figura 4.6: rappresentazione immagine in uscita da OV7640…….……..47
Figura 4.7: (a) uscita RGB 555, (b) uscita RGB 565, (c) uscita YUV…….…….47
Figura 5.1: maschera di convoluzione……...50
Figura 5.2: filtro generico di dimensione **3*3**……52
Figura 5.3: prodotto colonna-riga con W_t scalare……53
Figura 5.4: filtro che utilizza la separabilità della maschera di convoluzione……...54
Figura 5.5: (a) filtro Sobel, (b) filtro Prewitt, (c) filtro Smoothing……..…..……55

Figura 5.6: (a) immagine originale, (b) (d) (f) elaborate con lo schema di figura (5.2),
(b) filtro di *smoothing*, (d) filtro di *Prewitt,* (f) filtro di *Sobel*.
(c) (e) (g) elaborate con lo schema di figura (5.4), (c) filtro di *smoothing*,
(e) filtro di *Prewitt,* (g) filtro di *Sobel*……55-56

Figura 5.7: relazioni non lineari rappresentanti arrotondamento e troncamento……..57

Figura 5.8: filtro realizzato in Fixed-Point, il blocco giallo converte i valori *double* in *fixed point* mentre il blu viceversa……...58

Figura 5.9: *dialog box* Simulink……59
Figura 6.1: plotting della gaussiana……...……61
Figura 6.2: matrice di *pixel*……63
Figura 6.3: maschere per il calcolo della derivata prima approssimata……63
Figura 6.4: maschere di *Sobel*……...…65

Figura 6.5: schema algoritmo *Forstner* con *Simulink*: i blocchi verdi eseguono i filtri di *Sobel*, quelli rossi *smoothing* con gaussiana……66

Figura 6.6: maschera gaussiana **5*5**……..…66
Figura 6.7: blocco di calcolo della *corner detection*……67
Figura 6.8: immagine da elaborare……...…..…68
Figura 6.9: listato *Matlab* che converte l'immagine dal formato RGB a scala di grigi…..……69
Figura 6.10: listato *Matlab* per visualizzazione risultato……...69
Figura 6.11: risultato immagine elaborata……..……70
Figura 6.12: (a) risultato ottenuto via *software*, (b) risultato ottenuto dalla nostra simulazione……71

Figura 7.1: schema realizzato con blocchi *fixed-point*……...72
Figura 7.2: parametri del blocco di conversione *double fixed-point*……....73
Figura 7.3: maschera usata per il pre-filtraggio……..74
Figura 7.4: tabella dati……...…74
Figura 7.5: esempio di assegnamento parametri per blocco moltiplicatore ……75
Figura 7.6: esempio assegnamento parametri con coefficienti frazionari……76
Figura 7.7: schema di calcolo per l'algoritmo di Forstner……79

Figura 7.8: (a) immagine di test; (b) localizzazione dei *corner*; (c) visualizzazione dell'ampiezza dei *corner* su scala logaritmica……...……81
Figura 7.9: immagine da elaborare……82
Figura 7.10: risultato immagine elaborata……83
Figura 7.11: differenza tra algoritmo di tipo *double* e *fixed-point*……84
Figura 7.12: visualizzazione ampiezza dei *corner*……85
Figura 8.1: schema realizzato con *System Generator*……86
Figura 8.2: *function block* del blocco moltiplicatore……...…87
Figura 8.3: immagine a massimo contrasto possibile……89
Figura 8.4: valori limite del gradiente in uscita dal filtro di gradiente……90
Figura 8.5: configurazione *function block* del filtro del gradiente……91
Figura 8.6: (a) approssimazione della derivata parziale lungo la direzione y,
(b) approssimazione della derivata parziale lungo la direzione x……..92

Figura 8.7: blocco che esegue il calcolo della *corneress*……...……93
Figura 8.8: schema completo con selezione dei *corner*……95
Figura 8.9: schema del blocco di sogliatura……...……95
Figura 8.10: risultato dell'algoritmo di *corner detection*……...…96
Figura 8.11: risultato dell'algoritmo con soglia 100……..97
Figura 9.1: formato YUV della videocamera OV7640……..98
Figura 9.2: circuito per la selezione della componente "Y"……..99
Figura 9.3: circuito che riporta alla configurazione originale……100
Figura 9.4: formato RGB 555 di OV7640……...…100
Figura 9.5: circuito di interfaccia fra OV7640 e filtro generico……101
Figura 9.6: schema di filtro generico……...102
Figura 9.7: circuito che riporta alla codifica originale……102

Figura A.1: sintesi additiva e sottrattiva……..107

Figura A.2: Lo spazio colore RGB. Si nota che il nero è nell'origine (0,0,0), dove le componenti sono nulle; il bianco di conseguenza è in (1,1,1), dove le tre componenti si sommano; le tre componenti "pure" si trovano negli spigoli, dove è massimo uno dei tre valori negli assi coordinati e nulli gli altri due…….….109

Figura A.3: diagramma CIE di cromaticità: (a) mostra il range di colori, detto "gamma" prodotto dai monitor RGB (la forma triangolare), e quello dei dispositivi di stampa (la forma irregolare); (b) mostra la mancanza di uniformità percettiva……..…...111

Figura A.4: (a) e (b) lo spazio colore HSV: la tonalità è l'angolo di escursione tra le tinte, la saturazione è la distanza di un colore (punto) dall'asse piramidale, l'intensità è la distanza dalla punta della piramide . (a) In rosso è segnato l'asse, in corrispondenza della saturazione nulla, che indica la scala di grigio……...….112

Figura B.1: rappresentazione numero binario…….….116
Figura B.2: *dialog box* del blocco *fixed-point gain*……...….120
Figura B.3: elenco blocchi *fixed-point*……121
Figura B.4: *range* di rappresentabilità……...……..122
Figura C.1: *Simulink Library Browser* ……123
Figura C.2: *System Generator dialog box*……125
Figura C.3: *Gateway In dialog box*……...…...126
Figura C.4: dialog box dell'icona "*Resource Estimator*"……127

Capitolo 1

Introduzione

L'analisi delle immagini assistita dall'utilizzo del calcolatore è un'importante branca dell'informatica, della matematica e dell'ingegneria dell'informazione. Il sistema visivo umano manifesta un'incredibile capacità di elaborazione dei dati che gli pervengono attraverso gli occhi sotto forma di radiazione dall'ambiente esterno. Un tale sistema naturale può elaborare rapidamente grandi moli di dati, con grande flessibilità e senza sforzo apparente. Programmare al calcolatore degli algoritmi che permettano di simulare la visione umana, seppure con grande approssimazione, presenta delle difficoltà. Nonostante le capacità di calcolo dei calcolatori siano sempre crescenti, ancora non si riescono ad avere dei sistemi che possano competere con il sistema visivo umano.

L'obiettivo primario dell'elaborazione delle immagini è quello di rendere esplicito il contenuto informativo dell'immagine stessa; quali aspetti debbano essere evidenziati in un'immagine, dipende evidentemente dalla natura dell'applicazione alla quale si fa riferimento. Più in particolare diremo che il contenuto informativo di un'immagine può essere esplicitato a diversi livelli, differenti al variare del set di caratteristiche che si vogliono evidenziare e dalle associazioni che si intendono mettere in atto.

Performance e costi sono entrambi importanti parametri nella valutazione dei moderni sistemi elettronici. E' noto il fatto che il costo è direttamente legato alle performance, ma i progettisti sono costantemente all'inseguimento di alte prestazioni a fronte di bassi costi. Processori *General-purpose* sono ampiamente usati per implementare algoritmi di convoluzione-filtraggio. Molte volte tutte le funzionalità offerte da un *general-purpose* non sono necessarie o richieste dall'applicazione, così le funzionalità inutilizzate diventano un costo *overhead*. Inoltre, la disponibilità commerciale dei *general-purpose* spesso non incontra

le desiderate richieste di ottenere alte prestazioni a bassi costi; questo porta ad utilizzare processori *special-purpose* nell'implementazione di algoritmi di calcolo *real-time*.

Lo scopo di questa tesi è lo studio di un'architettura *special-purpose* per la realizzazione di algoritmi di *corner detection* basata su dispositivi a logica programmabile.

Capitolo 2

Stato dell'arte

La convoluzione è una delle operazioni fondamentali richieste nel processo di elaborazione di immagini digitali. Essa è usata in operazioni di filtraggio lineare come *smoothing*, *denoising*, *edge detection* ed altro [1, 2].

Un'immagine digitale può essere rappresentata con un *array* di numeri in uno spazio 2-D. Ogni numero (o *pixel*) è individuato dalla riga e dalla colonna che ne indicano le coordinate mentre il valore numerico rappresenta a seconda della codifica utilizzata il colore, l'intensità luminosa ecc. Una delle più semplici, codifica la luminanza attraverso un "livello di grigio"; i livelli di grigio sono tipicamente rappresentati con un *byte* o numero binario 8-*bit unsigned*, in decimale il *range* varia da 0 a 255.

L'equazione (2.1) mostra l'algoritmo di convoluzione discreta in 2-D, dove T è il valore di ingresso dell'immagine, W è il valore dei coefficienti del filtro, e T' è l'immagine in uscita[2].

$$T'[x,y] = W[x,y] * T[x,y] = \sum_{I=0}^{N-1} \sum_{J=0}^{N-1} W[I,J] T\left[x - J + \left(\frac{n-1}{2} \right), y - J + \left(\frac{n-1}{2} \right) \right] \qquad (2.1)$$

La figura(2.1) mostra la definizione per T', W, T. Assumendo che T abbia dimensione di $(i * j)$ *pixels* e W abbia dimensione di $(n * n)$ *pixels*, T' assumerà la dimensione di $(i * j)$ *pixels*.

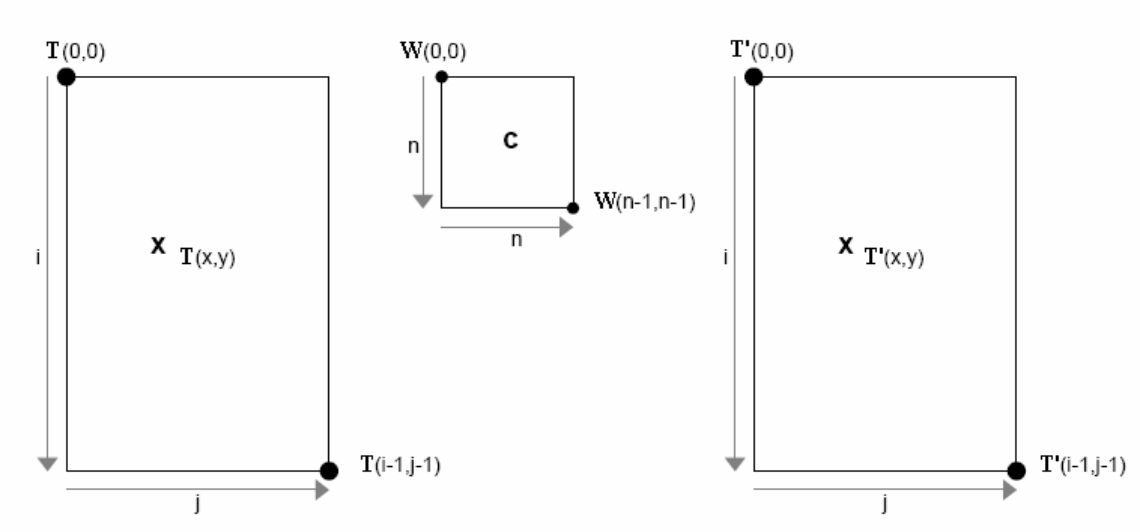


Fig. 2.1 : immagine di ingresso (T), filtro (W), immagine di uscita (T').

La convoluzione può essere pensata come operazione su una finestra mobile [2]. Come mostra l'equazione (2.1), un *pixel* di uscita di T'[*x*,*y*] è ottenuto ponendo il punto centrale del filtro W (denotato con **c** in figura (2.1)) nell'angolo in alto in T[*x*,*y*]. Tutte le sovrapposizioni dei *pixels* T sono moltiplicate con i corrispondenti coefficienti del filtro W, successivamente, tali prodotti sono sommati per generare un *pixel* T'[*x*,*y*]. Il successivo *pixel* di uscita è ottenuto facendo scorrere la finestra W di un *pixel* a destra e ripetendo il procedimento visto sopra. La figura (2.2) illustra l'idea delle operazioni mediante finestra mobile. W è prima centrata su T[3,4] per fornire T'[3,4] poi si sposta su T[3,5] per dare T'[3,5].

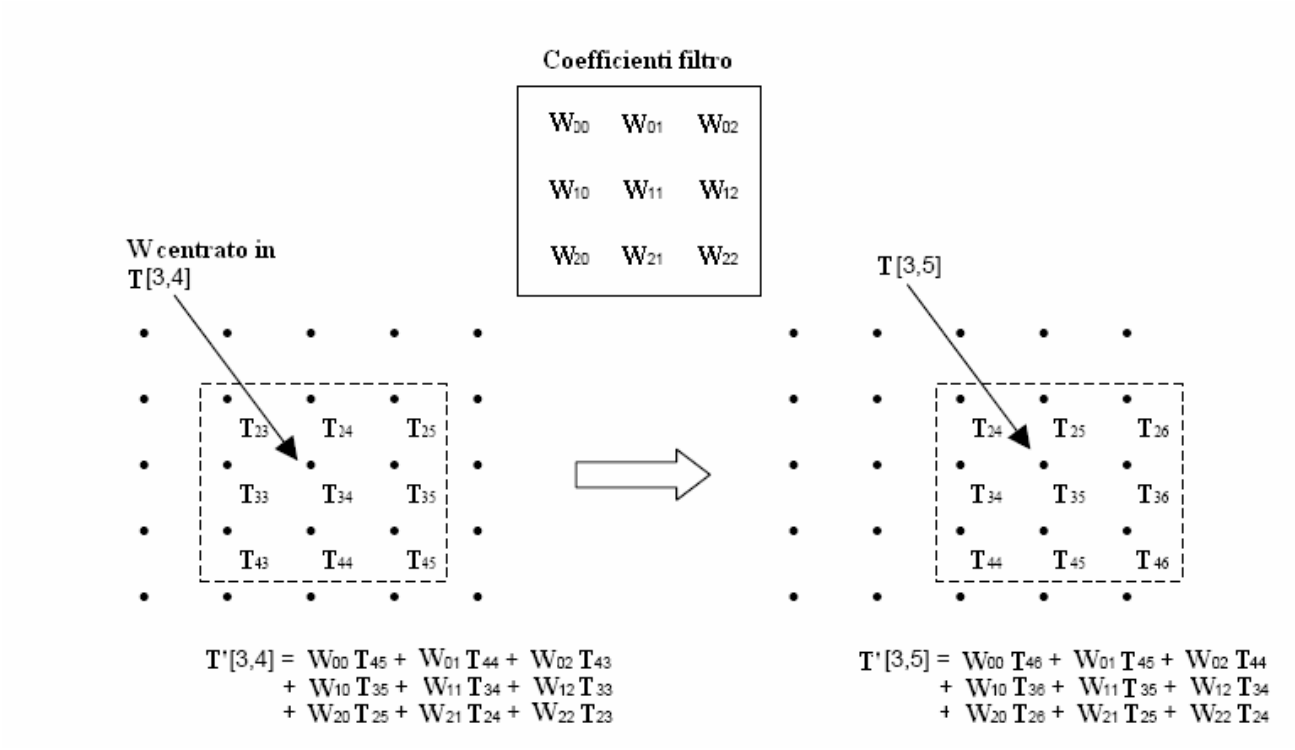


Fig. 2.2: esempio di generazione di uscita di due *pixels* consecutivi

Dalla figura (2.3) si desume che quando un *pixel* di uscita è elaborato, è richiesto l'accesso all'intera riga precedente o ad una porzione di essa. In generale sono necessarie solo (*n*-1) righe più *n pixels* di ingresso. La figura (2.4) mostra un esempio di filtro di dimensione 3*3 dove l'area ombreggiata evidenzia i *pixels* da mantenere in memoria.

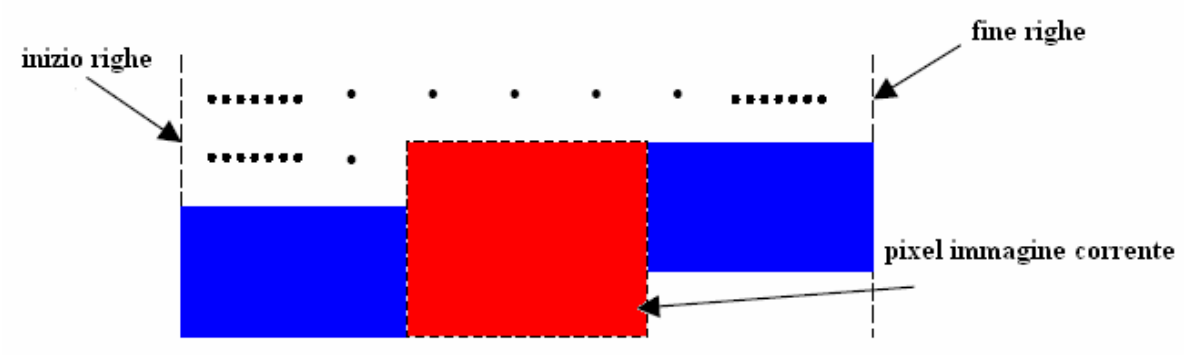


Fig. 2.3: esempio che mostra che solo (*n*-1)righe precedenti più *n pixels* devono essere memorizzati.

La convoluzione è una parte vitale dell'*image processing* e può essere eseguita via *software* o *hardware* [3]. Molti sforzi sono stati spesi per velocizzare il processo attraverso l'implementazione *hardware* [3, 4, 5, 6, 7]. Questo è dovuto al fatto che la convoluzione è un algoritmo computazionalmente oneroso come mostrato nell'equazione (2.1). Per esempio, con un filtro di dimensione 5*5, il calcolo di ogni *pixel* in uscita richiede 25 operazioni. Aumentando la finestra di convoluzione, il numero di operazioni richieste si incrementa in mo considerevole come mostrato in figura (2.4):

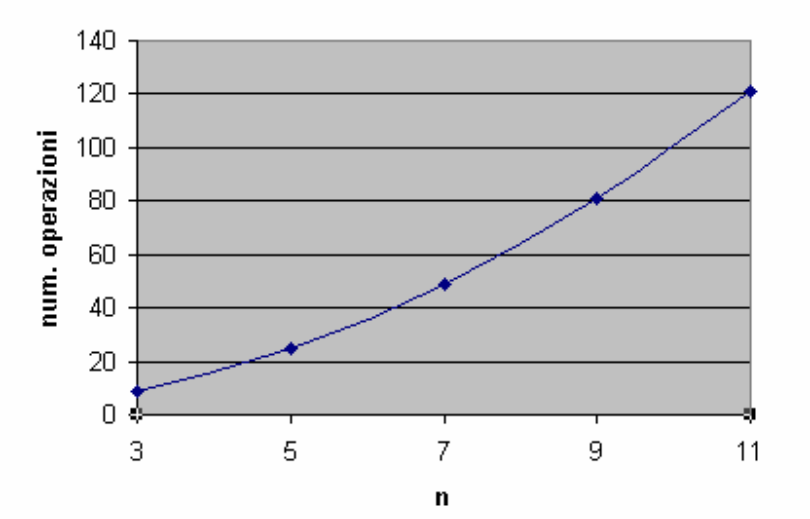


Fig. 2.4: grafico della complessità computazionale al variare di n.

Bosi e Bois in [3] proposero di utilizzare un' FPGA per velocizzare il processo di calcolo. In [4, 5, 6, 7], lo scopo principale dell'architettura della convoluzione proposta è quello di rispettare i vincoli *real-time* dell'*image-processing*.

Hsieh e Kim in [4] proposero un'architettura VLSI altamente "*pipelined*".

Il parallelismo ed il *pipelining* sono due tecniche che consentono di aumentare il *rate* di operazioni. In particolare con il parallelismo si agisce replicando più volte la stessa struttura hardware, mentre, il *pipelining* divide le operazioni da eseguire in sotto-operazioni, ognuna delle quali impiega una frazione del tempo totale necessario per completare il calcolo. Ogni sotto-operazione è assegnata ad una parte di hardware diversa, chiamata *pipe stage*. In questo modo la tecnica di *pipelining* permette di iniziare l'operazione successiva prima che quella precedente sia terminata. Il tempo totale di completamento non dipenderà quindi dal tempo di completamento dell'istruzione totale, ma dalla frequenza con cui i risultati escono dalla *pipeline*.

Al contrario del parallelismo, il grosso merito del *pipelining* è quello di fornire concorrenza nei calcoli senza la duplicazione di strutture hardware, e di conseguenza senza costi aggiuntivi. Lo svantaggio è dato dal fatto che si hanno problemi nel caso in cui le varie

operazioni eseguite debbano interagire tra di loro. In questi casi può accadere che una nuova istruzione non possa entrare nella pipeline finchè la precedente non è finita. In tal modo il guadagno di tempo ottenuto con la *pipeline* verrebbe ridotto. Si osservi inoltre che il più lento *pipe stage* determina il passo della *pipeline*. Di conseguenza è importante dividere le operazioni in sotto-operazioni di lunghezza uguale. Quando le criticità esposte non possono essere risolte via hardware, diventa importante ottimizzare il flusso di operazioni che compongono la pipeline.

L'obiettivo di questa tesi è lo studio e l'analisi di una architettura per *l'image-processing real time*, di tipo embedded su strutture robotiche.

I sistemi *embedded* sono diffusi ormai nelle più disparate apparecchiature elettroniche che funzionano grazie ad un micro-calcolatore e ad un firmware che li controlla.

Un sistema *embedded* è un computer dedicato ad uno scopo particolare, che si trova completamente all'interno dell'apparato che esso controlla. Ogni sistema *embedded* deve soddisfare dei vincoli specifici ed eseguire dei compiti predefiniti, a differenza di un computer generico.

Le caratteristiche dei sistemi *embedded* che tanto successo riscuotono sul mercato sono riassumibili in:

1. Specificità: il sistema è progettato per svolgere una precisa funzione nota a priori, a differenza di quanto accade per i sistemi *general purpose*.
2. *Real-time*: tipicamente il sistema presenta dei requisiti real-time, ovvero l'esecuzione dei *task* assegnati deve rispettare dei vincoli temporali precisi.
3. Sistema distribuito: è possibile costruire sistemi complessi scomponendoli in sottosistemi dedicati, invece di dover usare un solo calcolatore per operazioni di controllo e supervisione.
4. Affidabilità: le logiche programmabili sono progettate per poter garantire un funzionamento continuato senza alcuna interruzione o blocco del sistema. Ciò è fondamentale in particolare nei sistemi *hard-real-time*.

5. Peso e dimensioni opportune, basso consumo energetico: sono aspetti tipicamente trascurati nei sistemi *general purpose*, ma che assumono caratteristiche salienti nei sistemi *embedded*.

Vista la bibliografia consultata e le nostre specifiche di lavoro abbiamo pensato di realizzare una struttura totalmente basata su FPGA senza ausilio di dispositivi esterni.

2.1 Features

I descrittori di basso livello delle immagini possono essere in generale classificati in 3 tipi: *features* di livello zero (zone uniformi), di primo livello (contorni), e di secondo (punti).

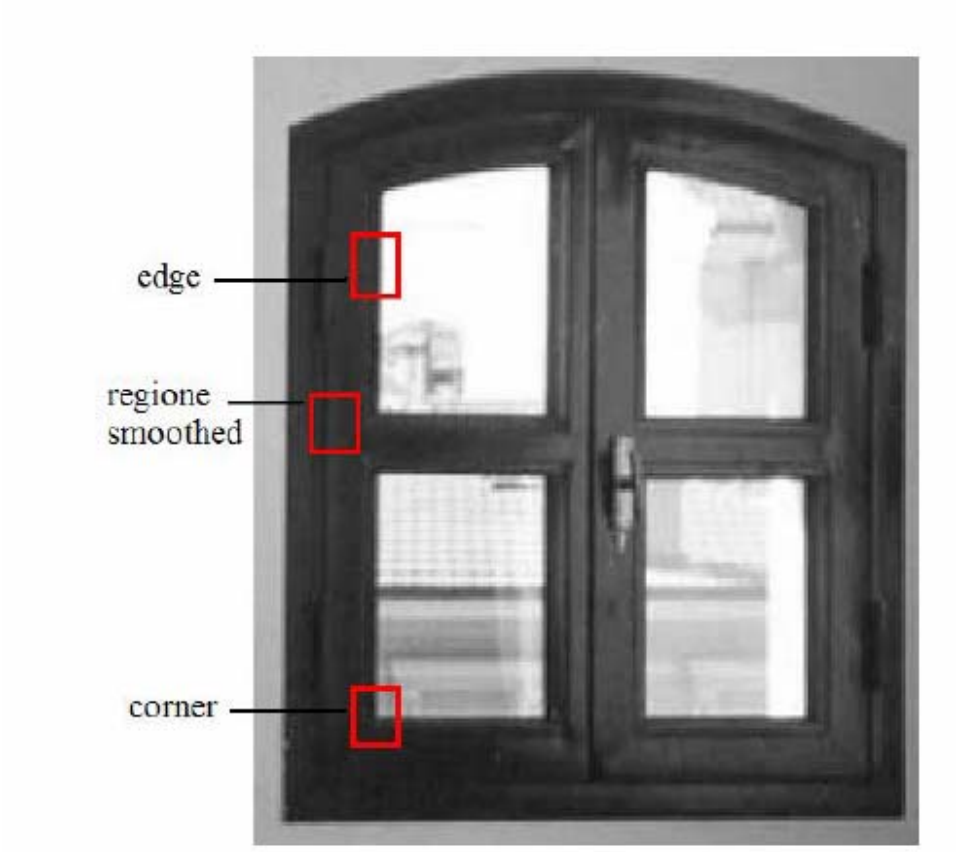


Fig. 2.5: vengono visualizzate nell'immagine i tre tipi di *feature*: una regione uniforme, un contorno(*edge*) e un *corner*.

Le regioni uniformi (*regions* o "*blobs*") normalmente corrispondono a pezzi di superfici (*patch*) *smoothed*, caratterizzate da variazioni "dolci" (*smooth*) dell'intensità.

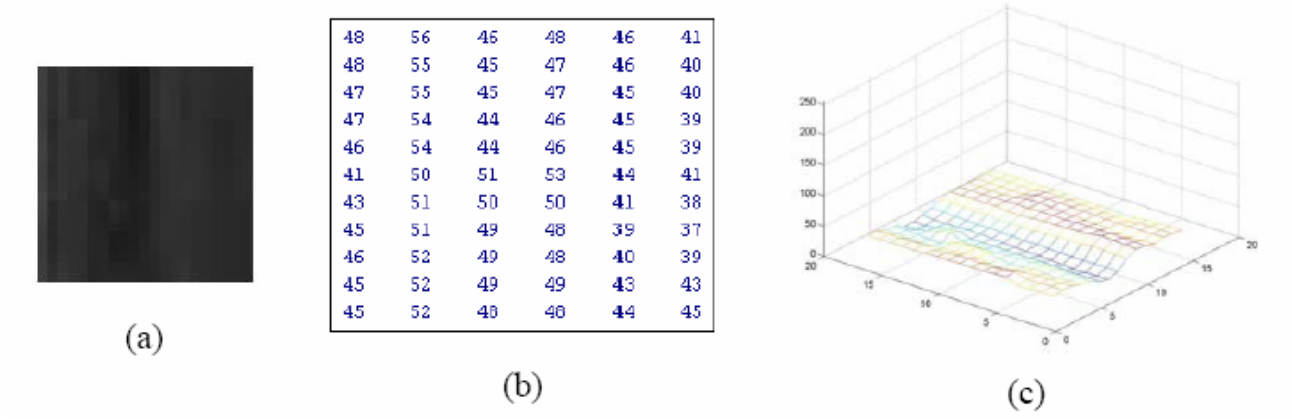


Fig. 2.6: (a) una regione di immagine smoothed, (b) i valori di intensità e (c) il grafico della funzione di intensità luminosa I(*x*,*y*).

Il *tracking* di queste regioni non è sempre facile: nonostante i recenti progressi (i descrittori statistici di Etoh e Shirai, le approssimazioni dei bordi di Meyer e Bouthemy), sarà necessaria ancora molta ricerca teorica e empirica prima di considerarlo affidabile [8].

Gli *edge*, o contorni, si hanno in corrispondenza di significative variazioni monodirezionali dell'intensità luminosa. Vengono generalmente rilevati o attraverso i massimi relativi nelle derivate di primo ordine delle immagini, opportunamente sfocate, o in corrispondenza degli zero-crossing nei laplaciani di gaussiana delle immagini.

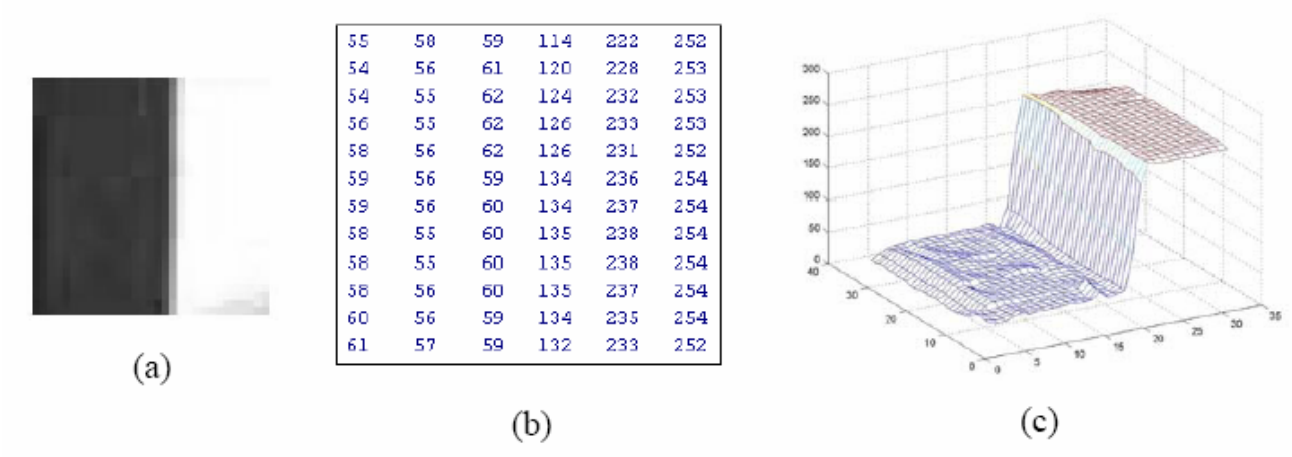


Fig. 2.7: un bordo (a), (b) i valori di intensità, (c) il grafico della funzione di intensità luminosa I(*x*,*y*). Si noti la variazione dell'intensità in una sola direzione.

I bordi sono sempre stati difficili da descrivere e inseguire. Nonostante i progressi introdotti con l'avvento degli *snakes*, curve arbitrarie utilizzate per modellare i contorni (citiamo a proposito l'edge detector di *Canny*), nel *tracking* degli *edge* attraverso sequenze di immagini, rimane il problema che i segmenti di contorno tendono a separarsi (*split*) e mescolarsi (*merge*) nell'evolversi del moto, con complicazioni considerevoli per il processo di *tracking*.

Point features, o *corner*, sono punti distinti dell'immagine localizzati in corrispondenza di discontinuità bidirezionali dell'intensità luminosa. Sono compresi punti di occlusione (giunzioni a *T*, *Y* o *X*) e discontinuità strutturali (gli spigoli, o giunzioni a *L*).

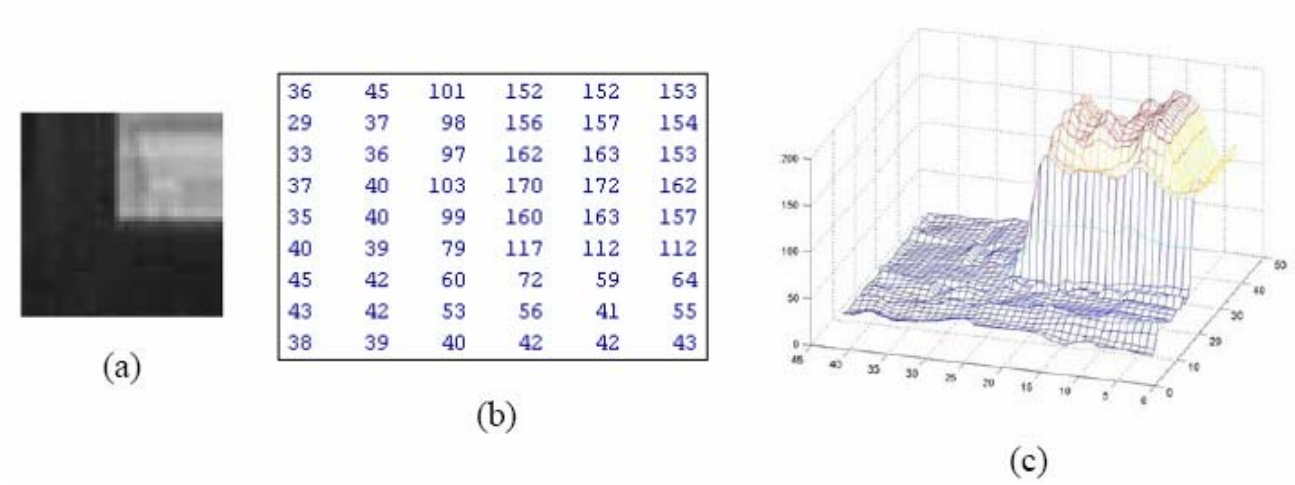


Fig. 2.8: un corner(a) dell'immagine del tipo "giunzione a L": (b) mostra la variazione dei valori d'intensità luminosa attorno al punto e (c) è il grafico della funzione intensità di colore I(*x*,*y*). Si noti la variazione dell'intensità in due direzioni.

I *corner* si impongono sugli *edge* come *features* nel *motion tracking*, perché in generale sono localizzabili accuratamente e ricorrono in immagini successive, cosa che li rende esplicitamente inseguibili. I *corner* sono inoltre spesso più abbondanti rispetto ai contorni rettilinei nel mondo naturale, cosa che li rende ideali da tracciare in un ambiente esterno o interno.

2.2 Algoritmi per il rilevamento dei *corner*

Negli ultimi anni sono stati pubblicati vari algoritmi per il rilevamento dei *corner*, che possiamo dividere in due gruppi.
Il primo riguarda tecniche basate sull'estrazione dei contorni e la successiva identificazione dei punti corrispondenti alla massima curvatura, o dove i segmenti di *edge* si intersecano.
Il secondo, e più grande gruppo, consiste di algoritmi che ricercano i *corner* direttamente a partire dall'intensità dei livelli di grigio dei pixel dell'immagine.
Noi ci concentreremo sul secondo gruppo di *point feature detector*, in particolare discuteremo brevemente gli algoritmi di *Kitchen-Rosenfeld* e di *Smith*, e ci concentreremo poi maggiormente su quello di *Harris* ed in particolare sulla variante di *Forstner*.

2.2.1 Smith (S-USAN) Corner Detector

Il rilevatore di *corner* di *Smith* è nato in relazione ad un progetto per la segmentazione di scene: l'*ASSET* (*A Scene Segmenter Establishing Tracking*).
E' un algoritmo molto particolare, poiché non si basa sulle derivate spaziali dell'immagine, né richiede *blurring*, ma utilizza un criterio "geometrico" abbastanza originale.
Il suo algoritmo si basa sulla valutazione della *USAN* (*Univalue Segment Assimilating Nucleus*), ossia dell'areola che circonda un pixel con luminosità media uniforme.

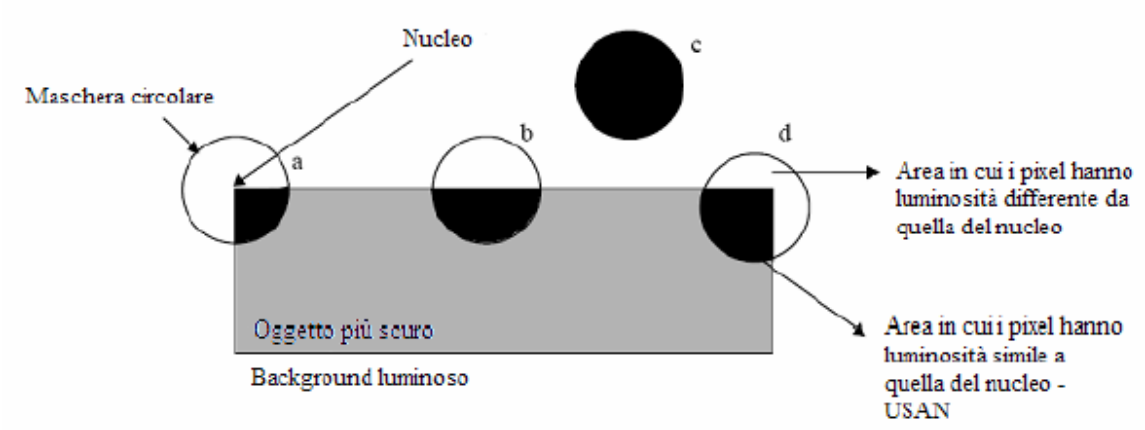


Fig. 2.9: maschere per il ritrovamento dei corner in diverse posizioni di un'immagine: (a) nucleo posizionato esattamente sul corner, la USAN è un quarto dell'area (b) rileva un edge (c) in una regione uniforme i pixel hanno per lo più la stessa intensità (d) la maschera così posizionata, secondo la soglia di Smith, non rileva ancora il corner, perché la USAN è troppo grande.

La maschera circolare nella pratica è approssimata con un quadratino 5x5, centrato nel pixel di interesse, con 3 pixel aggiunti al centro di ogni lato; l'intensità luminosa del pixel centrale, detto nucleo, viene confrontata con quella di tutti gli altri pixel della maschera, attraverso la seguente funzione C:

$$C(x, x_n) = 100e^{-\left(\frac{I(x)-I(x_n)}{t}\right)^6} \tag{2.2}$$

dove $X_n = (x_n, y_n)$ sono le coordinate del nucleo, X è la posizione di un generico pixel appartenente alla maschera, $I(x)$ è la sua intensità luminosa, e t è la soglia della differenza di luminosità, utile per determinare la variazione di intensità tra il nucleo e i pixel dell'areola che lo definisce come corner.

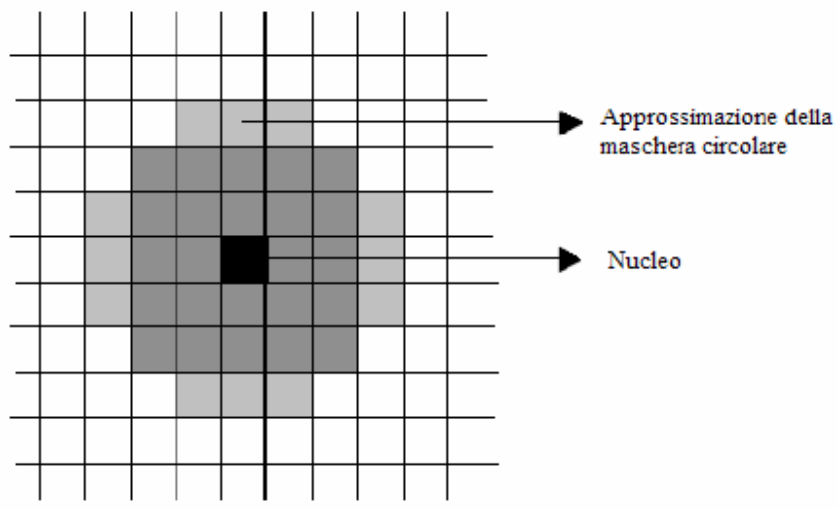


Fig. 2.10: La maschera di Smith per la ricerca delle USAN. Si notano il nucleo (in nero, il pixel centrale), il quadrato 5x5 (in grigio scuro), e i tre pixel per lato (in grigio chiaro) per l'approssimazione dell'area circolare.

Il calcolo di viene effettuato su tutti i pixel della maschera, per trovare lo scalare u:

$$u = \sum_{x} C(x, x_n) \tag{2.3}$$

che rappresenta il valore dell'area USAN, e viene opportunamente sogliato per stabilire se il nucleo è un corner.

I valori di C sono limitati, poiché $0 \leq C \leq 100$, e conseguentemente u è limitata, in particolare $0 \leq u \leq 3700$, essendo 37 i pixel della maschera circolare; possiamo fissare una soglia geometrica, che varia in funzione del tipo di *feature* che vogliamo rintracciare: per esempio g=1850 permette di rintracciare USAN inferiori o uguali a metà cerchio, quindi corner e contorni di oggetti; diminuendo g si influenza la forma dei *corner* ritrovati, e quindi progressivamente vengono rilevati solo i *corner* più spigolosi.

Una volta calcolate le *u* per tutti i pixel, viene creata un'immagine intermedia, $S(x)$, tale che:

$$S(x) = \begin{cases} 0, u(x) \geq g \\ g - u(x), u(x) < g \end{cases} \tag{2.4}$$

Con una scansione di $S(x)$, vengono dichiarati corner i massimi locali dell'immagine, su regioni di immagine di 5x5 pixel.

2.2.2 Kitchen-Rosenfeld Corner Detector

E' stato uno dei primi ad essere riportato in letteratura, ragion per cui è diventato il punto di partenza e il termine di confronto per la ricerca e per i successivi algoritmi per il tracciamento dei corner. L'algoritmo si basa sul calcolo del valore della *corneress* ς attraverso un'opportuna forma quadratica ricavata dalla magnitudo dei gradienti e dal *rate* di cambiamento del gradiente nelle due direzioni; più precisamente:

$$\varsigma = \frac{I_{xy} I_y^2 + I_{yy} I_x^2 - 2 I_{xy} I_x I_y}{I_x^2 + I_y^2} \tag{2.5}$$

Ove I è l'intensità di livello di grigio dell'immagine, I_x e I_y le derivate spaziali di primo ordine lungo *x* e *y*, ovvero le componenti del gradiente, e I_{xx} , I_{yy} e I_{xy} le derivate spaziali di secondo ordine. La funzione viene calcolata a livello locale, pixel per pixel, e un punto viene considerato saliente solo quando la sua *corneress* ς è inferiore ad una certa soglia: minore è il valore di ς , migliore è considerato il *corner*.

2.2.3 Harris Corner Detector

Il corner detector di Harris è stato usato con successo per la rilevazione dei punti salienti per il progetto di visione DROID 3D, e deve la sua popolarità all'affidabilità nel trovare le giunzioni a "L" e alla sua buona stabilità temporale, che lo rendono un corner detector attrattivo per un *tracker*; inoltre è meno sensibile al rumore dell'immagine rispetto agli altri algoritmi, perché basato interamente sulla derivata di primo ordine.

Proprio perché basato sulla derivata spaziale, lo smoothing dell'immagine è spesso richiesto per migliorarne le performance di calcolo e l'affidabilità del rilevamento. E' stato comunque mostrato che uno *smoothing* eccessivo potrebbe rendere meno accurata la localizzazione del *corner*.

Harris calcola, per ogni pixel dell'immagine, un valore di *corneress*, ς che rappresenta la bontà del punto trovato:

$$\varsigma = \frac{\langle I_x^2 \rangle + \langle I_y^2 \rangle}{\langle I_x^2 \rangle \langle I_y^2 \rangle - \langle I_x I_y \rangle^2} \tag{2.6}$$

Ove I_x , I_y ,sono le componenti del gradiente dell'intensità di grigio dell'immagine, calcolate con *kernel* 3x3 o 5x5; la notazione $\langle \bullet \rangle$ specifica che gli elementi sono convoluiti con *kernel* di gaussiane di deviazione standard σ, in modo tale che vengano mediati: questo garantisce un abbattimento dei picchi di derivata causati dall'inevitabile rumore delle immagini, per cui delle variazioni di colore in regioni quasi uniformi potrebbero dar luogo a massimi locali non desiderati nel gradiente. Un buon corner è definito se ha un piccolo valore di ς .

La *corneress* di *Harris* può anche essere interpretata come forma quadratica della matrice:

$$\mathbf{M} = \begin{pmatrix} \langle I_x^2 \rangle & \langle I_x I_y \rangle \\ \langle I_x I_y \rangle & \langle I_y^2 \rangle \end{pmatrix} \tag{2.7}$$

per cui la *corneress* diventa il rapporto tra la traccia della matrice e il suo determinante:

$$\varsigma = \frac{tr(M)}{\det(M)} \tag{2.8}$$

Incidentalmente, detti λ_1 , λ_2 gli autovalori di M, possiamo classificare il tipo di *feature* secondo i valori assunti dagli stessi, in particolare:

-$\lambda_1 \approx \lambda_2 \approx 0$: *blob*, regione uniforme.

- $\lambda_1 \approx 0, \lambda_2 >> 0$: *edge*, dove l'autovettore associato a λ_1 indica la direzione del bordo, e quello associato a λ_2 la componente normale ad esso;

- $\lambda_1 > \lambda_2 >> 0$: *corner*, la cui bontà è proporzionale al modulo dell'autovalore massimo.

In altre parole, l'elissoide formato dagli autovettori e autovalori di M può darci indicazioni strutturali sul tipo di *corner*; trasformazioni lineari non cambiano le proprietà dell'elissoide, tuttavia operazioni come il *bluring* o trasformazioni affini possono portare a deformazioni che portano alla perdita di tali proprietà.

Esistono numerose varianti dell'algoritmo di *Harris*, per esempio quella di *Förstner*, che determina i *corner* come massimi locali della *corneress* C:

$$C = \frac{1}{\varsigma} = \frac{\langle I_x^2 \rangle \langle I_y^2 \rangle - \langle I_x I_y \rangle^2}{\langle I_x^2 \rangle + \langle I_y^2 \rangle} \tag{2.9}$$

che non è altro che l'inverso della *corneress* di *Harris* ς .

Capitolo 3

Le FPGA

3.1 Introduzione

Tra le varie possibili soluzioni, particolarmente interessante risulta essere quella che prevede l'utilizzo di un circuito a FPGA. Questi dispositivi presentano il migliore compromesso tra flessibilità d'implementazione e consumo di potenza, rispetto a soluzioni in cui si utilizzano dei DSP o degli ASIC.

F.P.G.A è l'acronimo di *Field Programmable Gate Array* cioè circuiti a "*Gate Array*" programmabili elettricamente, o per meglio dire, come suggerisce il nome stesso, circuiti costituiti da un insieme di *blocchi logici*, che contengono al loro interno delle porte logiche, la cui funzione risulta programmabile tramite segnali elettrici. Inoltre, per aumentarne la flessibilità, risultano programmabili, elettricamente, anche le interconnessioni tra i blocchi stessi. Tali caratteristiche fanno sì che questi circuiti possano essere riconfigurati un numero elevato di volte senza che si abbiano cali di prestazioni. Ciò rende questi circuiti molto utili nella fase di "prototipizzazione" di un circuito integrato prima che esso sia avviato alla produzione industriale. Infatti, tramite lo stesso codice di descrizione dell'hardware necessario a generare l'integrato stesso, si può programmare la FPGA in modo che svolga le stesse funzioni e su di essa "validare" il progetto, con la possibilità di correggere gli eventuali errori, prima che inizi la produzione, permettendo un notevole risparmio di risorse.

I progressi della microelettronica, poi, hanno esteso le possibilità di queste strutture. La possibilità di realizzare facilmente funzioni complesse ottenendo, allo stesso tempo, circuiti in grado di operare a frequenze sempre più elevate, ha fatto sì che le FPGA siano entrate in competizione con gli stessi circuiti ASIC (Application Specific Integrated Circuit) figura (3.1).

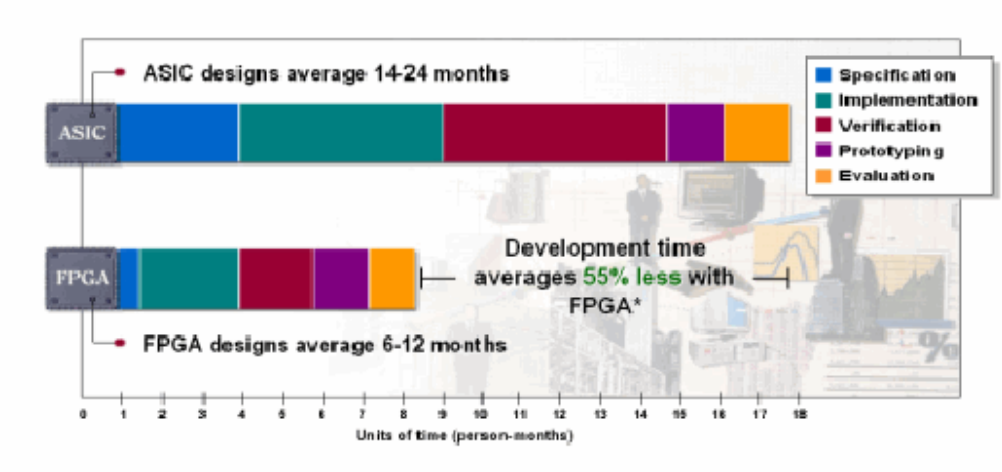


Fig.3.1: confronto tempi di sviluppo FPGA-ASIC.

Se, da un lato, si è cercato di aumentare la frequenza massima di utilizzo per tali circuiti, l'aspetto di riduzione della potenza consumata non è stato, per molto tempo, affrontato. Lo sviluppo di sistemi portatili ha, però, dato lo spunto per la ricerca di nuove soluzioni a basso consumo.

3.2 I blocchi logici

Una FPGA si può immaginare, come mostrato in figura (3.2), come un insieme di blocchi logici programmabili, chiamati *configurable logic block* (*C L B*) cioè blocchi logici configurabili, e circuiti di interconnessione, anch'essi programmabili, chiamati anche *switching blocks* (SB).

Ogni blocco logico risulta, a sua volta, costituito da *look-up table* (LUT), flip-flop e circuiti aggiuntivi quali, ad esempio, quelli per la generazione veloce dei riporti nelle somme. Per comprendere meglio tali circuiti risulta necessaria una breve spiegazione.

Ogni *look-up table* è caratterizzata da un numero k di ingressi che definisce anche il *parallelismo* del CLB. Questi ingressi possono essere combinati, all'interno della LUT, in modo da realizzare una funzione logica. Ciò è possibile perché la struttura di una LUT, in questo caso specifico, non è altro che quella di un *multiplixer* realizzato a *pass transistor*. Questi dispositivi, con la funzione di interruttori, vengono "aperti" o "chiusi" in base al contenuto di alcune celle di RAM statica, o SRAM. Tali celle sono programmate al fine di far svolgere al blocco la funzione desiderata. Questa organizzazione consente di ottenere una notevole flessibilità: variando i valori memorizzati all'interno delle celle SRAM, infatti, si ottengono funzioni tra loro molto diverse. Tale soluzione consente, in più, di ottenere una notevole flessibilità senza *sacrificare* troppa area di silicio.

Generalmente ogni CLB contiene due LUT, ognuna delle quali ha quattro ingressi. Tale numero di ingressi è quello che permette il migliore compromesso tra area occupata, numero di funzioni implementabili, potenza dissipata, ed ottimizzazione globale durante la sintesi logica di un circuito.

Nonostante la notevole flessibilità ottenibile con la struttura a look-up table, in ogni blocco logico sono anche aggiunti ulteriori circuiti logici.

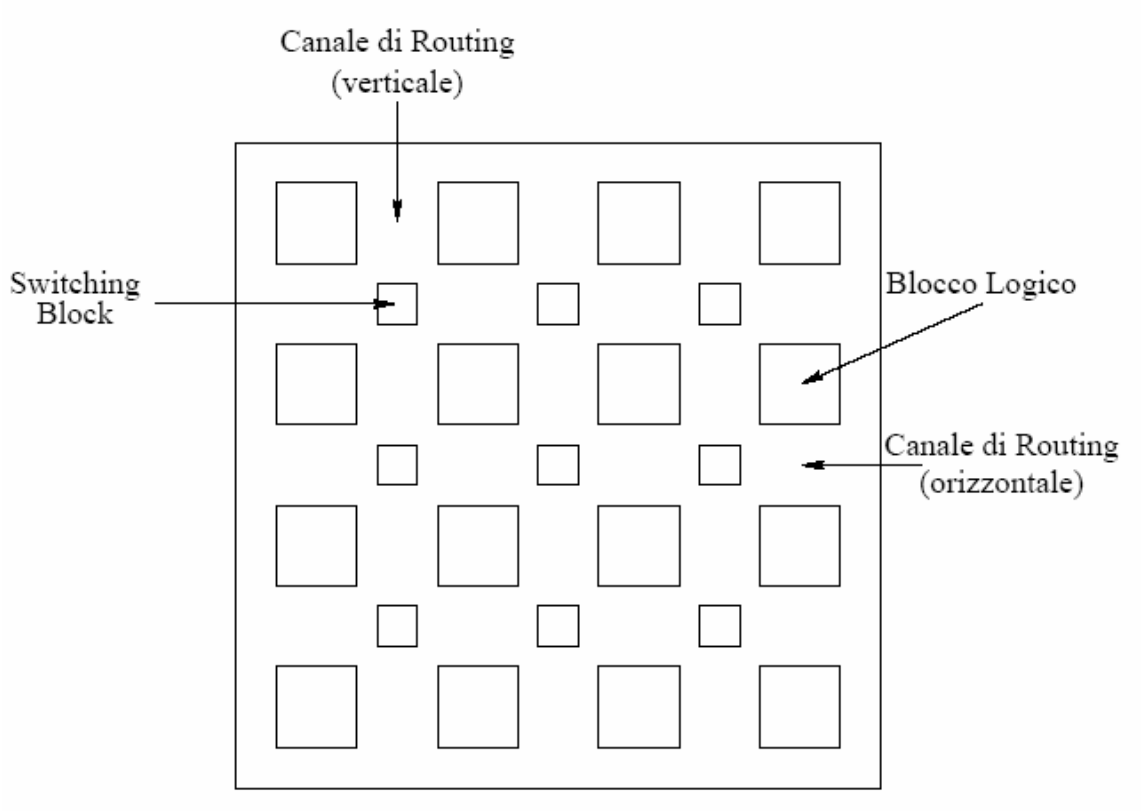


Fig.3.2: Schema a blocchi di una FPGA generica.

In particolare, per permettere la realizzazione di circuiti aritmetici più veloci, vengono inserite alcune porte exor, per le funzioni di somma tra bit, e circuiti di generazione veloce del carry, per realizzare sommatori ottimizzati in velocità. La realizzazione di queste funzioni dentro le look-up table, infatti, porterebbe a circuiti poco efficienti sia in termini di area, sia in termini di massime frequenze di funzionamento ottenibili. Queste porte aggiuntive permettono, quindi, di ottenere circuiti veloci con un costo, in termini di area occupata, ridotto.

Per permettere la realizzazione di circuiti sequenziali, e non soltanto combinatori, ogni CLB è dotato, anche, di circuiti flip-flop di tipo D. Tali circuiti, poi, possono essere utilizzati per realizzare dei livelli di pipeline facilitando la realizzazione di circuiti "veloci", cioè in grado di operare a frequenze più elevate rispetto ai precedenti dispositivi basati su tecniche di calcolo puramente seriale.

Come si può desumere da quanto detto, i blocchi logici permettono la realizzazione di funzioni logiche anche complesse, in modo efficiente e flessibile. Le ottimizzazioni che sono state studiate per tali strutture, permettono di affermare che le eventuali fonti di consumo non risiedano in questi circuiti, ma queste debbono essere ricercate altrove nelle FPGA.

3.3 Le interconnessioni

La seconda fonte di flessibilità in una logica programmabile, di cui le FPGA sono esempi, è la possibilità di selezionare le interconnessioni tra i blocchi logici. Ciò è reso possibile dalla presenza di canali di *routing* (vedi figura (3.2)), in cui "scorrono" linee di interconnessione globali tra un dato numero di CLB lungo una stessa colonna, o riga.
Per permettere la connessione tra canali e CLB e tra canali verticali ed orizzontali, sono inseriti dei circuiti detti *switching block*. Questi elementi sono realizzati tramite matrici di pass transistor che collegano tra loro le linee di interconnessione. La presenza di tali collegamenti è programmabile esternamente grazie ai valori che sono scritti, dall'esterno, in celle SRAM che pilotano i pass transistor, allo stesso modo in cui si programmano le LUT nei CLB.
Da quanto detto, si intuisce che le linee di interconnessione possono risultare molto lunghe, presentando così valori di resistenza serie che possono diventare significativi. Inoltre la presenza dei pass transistor negli *switching block*, fa sì che tali linee siano "caricate" da elementi capacitivi dovuti alle diffusioni di *drain* degli stessi *pass transistor*. La necessità di dover caricare questi elementi parassiti, e le perdite dovute alle resistenze serie, fanno sì che si crei una notevole dissipazione di potenza nelle interconnessioni, oltre a ritardi che non possono essere trascurati. Se la possibilità di programmazione rende le FPGA molto flessibili, tale facoltà diventa la principale fonte di inefficienza energetica di tali strutture.
Inoltre, è necessario considerare un'ulteriore interconnessione, di tipo "speciale", quale è la distribuzione del clock. Da quanto già detto nella descrizione dei CLB, ogni blocco logico contiene al suo interno dei *flip-flop* il cui funzionamento corretto è regolato dal segnale di cadenza. Quindi tale segnale deve essere distribuito all'intero chip su cui è realizzata la FPGA.

Ciò significa che la distribuzione del clock richiede interconnessioni dedicate molto lunghe e con un elevato carico capacitivo, rendendole fonte notevole di consumo di potenza.
Queste considerazioni permettono di affermare che le eventuali inefficienze nel consumo di potenza risiedono, in larga parte, nelle interconnessioni presenti.

3.4 La scelta della FPGA da utilizzare

Le esigenze di sviluppare un sistema attraverso *hardware high-performance* facilmente reperibile sul mercato hanno fatto ricadere la scelta su di una famiglia di dispositivi denominati *Spartan*3 e prodotti da *Xilinx* [9]. Tali circuiti sono nati per applicazioni low-power, hanno infatti una tensione di alimentazione da 1,2 V a 3,3 V. Questi possono utilizzare le memorie di configurazione come se fossero vere memorie RAM per i dati, che risulterà un punto chiave nella realizzazione del nostro progetto, evitando di utilizzare una memoria esterna guadagnando in termini di ingombro e circuiteria di controllo.
Oltre a queste sono anche inserite ulteriori celle di memoria, chiamate *Block RAM*, realizzate appositamente per le elaborazioni. I CLB, inoltre, contengono due LUT a quattro ingressi e porte, aggiuntive, studiate per rendere più veloci i circuiti aritmetici. Oltre a ciò, queste FPGA possono contare di quattro reti di distribuzione del clock in modo da poter utilizzare circuiti con regimi temporali differenti su di uno stesso chip.
La famiglia *Spartan*3 presenta una serie di dispositivi in grado di contenere un numero di *gate equivalenti* variabile da più di cinquantamila, nel dispositivo più ridotto, fino ad arrivare ad un numero superiore ai cinque milioni di unità nella FPGA più "grande". Una così estesa possibilità di scelte permette, così, di selezionare il dispositivo più adeguato alle proprie esigenze. Ovviamente le FPGA con un numero di gate più grande permettono la "implementazione" di circuiti molto complessi, ma alla stesso tempo, determinano un consumo di potenza più elevato.

3.5 La metodologia seguita per l'analisi

Se è vero che i risultati ottenuti dalle prove rappresentano l'aspetto fondamentale per la caratterizzazione delle FPGA, risulta altrettanto importante l'impiego di una metodologia generale, per la loro realizzazione. Questo è un requisito fondamentale perché permette un confronto uniforme con i risultati ottenuti su altri dispositivi.

Il nostro iter ha previsto una prima fase di sviluppo tramite Simulink mediante il quale è stato possibile testare il funzionamento degli algoritmi proposti. Una seconda fase, di avvicinamento alla realizzazione *hardware*, è stata l'utilizzo del toolbox Fixed-Point di Simulink in cui si è verificato il comportamento dei circuiti con matematica a precisione finita. A questo punto sfruttando il programma Xilinx Design Tools System Generator[10], che permette la comunicazione tra l'ambiente Matlab-Simulink[11] e il software Project Navigator di proprietà Xilinx, abbiamo simulato in condizioni reali il comportamento dell'architettura e avviato la generazione automatica del codice V H D L.

Tale codice è stato sintetizzato tramite un *tool* di sintesi, di proprietà della Xilinx XST(*Xilinx Synthesis Technology)* [12].

Questo tool produce, in uscita, un file in formato *edif* tale da essere accettato in ingresso dal programma che si occupa di allocare le risorse logiche e le loro interconnessioni, facendo il cosiddetto *Place & Route*, sulla FPGA. Questo passo di progetto è stato eseguito tramite un tool sviluppato dalla stessa *Xilinx* denominato *Web_Pack*. Utilizzando *ModelSim* prodotto da MTI è stato possibile fare simulazioni del circuito sia a livello funzionale sia tenendo conto dei ritardi dovuti alla realizzazione dopo il *Place & Route*. Tutti questi passi realizzano un *flow* di progetto tipico per la generazione di un qualsiasi tipo di circuito integrato.

3.6 L'ambiente di sviluppo Xilinx ISE 6.3 I

Per lo sviluppo della tesi è stato utilizzato l'ambiente di sviluppo *Xilinx ISE WebPack* 6.3i.L'ambiente *ISE* (*Integrated Software Environment*) è una suite *Xilinx* per la progettazione software di FPGA e CPLD *Xilinx*. I prodotti *ISE Foundation* permettono una progettazione basata su HDL o *Schematic* che risulta completamente integrata all'ambiente di sviluppo. *Project Navigator* permette di visualizzare e controllare sia i componenti del progetto schematico sia gli strumenti di sintesi:

• Generazione automatica di *testbench HDL* usando *Xilinx HDL Testebencher*;
• Simulazione Schematico usando *ModelSim XE 5.8c – Xilinx*;

ISE supporta le seguenti famiglie di dispositivi:
• *Virtex-II Pro™*
• *Virtex™/-E/-II*
• *Spartan™-II/-IIE/-3*
• *CoolRunner™ XPLA3/-II/-IIs*
• *XC9500™/XL/XV*
permettendo di iniziare un progetto con i seguenti tipi di sorgenti:
• *HDL, VHDL, Verilog HDL, ABEL (ISE Text Editor)*
• Schematici (*Schematic Editor*)
• *NGC/NGO*
• Descrizioni di Macchine a Stati Finiti (*StateCAD State Machine Editor*)
• *IP Cores (CORE Generator)*

Partendo da file sorgenti, *ISE* permette di verificarne velocemente le funzionalità mediante simulazione con ModelSim XE II 5.8c, che permette la visualizzazione dei segnali di interesse con le relative temporizzazioni tramite un'interfaccia facile e intuitiva. Gli schematici sorgenti possono essere creati usando gli strumenti Xilinx Synthesis Technology (XST) attraverso l'ambiente integrato (Figura 3.3).

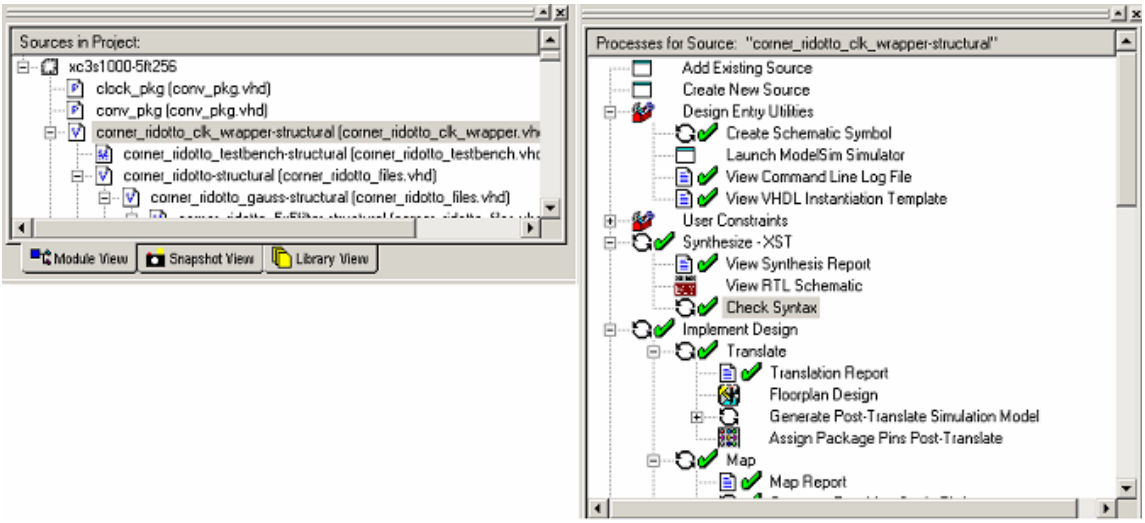


Fig.3.3: visualizzazione *Project Navigator*.

Nella finestra "*Sources in Project*" vengono elencati ordinatamente i listati V H D L che descrivono il progetto, mentre, nel "*Processes for Source*" sono elencati gli *steps* da seguire per la realizzazione del progetto: si noti che il *marcher* verde indica che la relativa fase di progettazione è valida.

L'interfaccia grafica del *Project Navigator* permette di gestire in maniera semplice l'intero processo di sviluppo.

Questo metodo di lavoro permette di sviluppare il progetto in sottoblocchi indipendenti, che successivamente verranno interconnessi per la realizzazione finale; il vantaggio è rappresentato dal fatto che i blocchi così creati possono essere verificati singolarmente in maniera indipendente attraverso i simulatori. Così facendo, si garantisce una maggiore sicurezza nella fase di sviluppo.

Una volta ottenuto il modello finale, si passa alla fase *DRC* (*Design Rule Check*) che segnala eventuali problemi relativi ai segnali, ai componenti o alle connessioni realizzate. Per procedere nello sviluppo del progetto, si eseguono i processi di generazione, traduzione e adattamento del file di programmazione del FPGA..

Ad ognuno dei tre processi è associato un documento di testo (identificato come *Report*) che raccoglie tutte le informazioni relative al processo eseguito evidenziando gli eventuali problemi che si sono presentati. Per la risoluzione è possibile collegarsi al sito della *Xilinx* (www.Xilinx.com) per visualizzare le soluzioni proposte relative agli errori.

Capitolo 4

Telecamera

4.1 Introduzione

Un metodo importante per rappresentare il mondo che ci circonda attraverso una macchina, consiste nella trasduzione di grandezze reali quali la luce in grandezze elettriche.
Si fa riferimento al principio sul quale si basa una telecamera: trasforma la luce riflessa o emessa dall'ambiente in segnali elettrici tramite fotorecettori.
Il componente fondamentale di una telecamera è il sensore, una matrice bidimensionale di fotorecettori ognuno dei quali è caratterizzato da una coppia di valori X e Y, ad indicare rispettivamente l'ascissa e l'ordinata.
Si definisce immagine la distribuzione spaziale della luminosità.
Un'immagine è quindi una funzione:

$$E = E(x, y)$$

con x, y : $1 \le x \le M$, $1 \le y \le N$
ove M , N definiscono le dimensioni dell'immagine.
La forma matriciale di un'immagine può essere scritta come mostrato in figura (4.1):

$$E(x,y) = \begin{bmatrix} e(x_1,y_1) & e(x_1,y_2) & \bullet\bullet & e(x_1,y_n) \\ e(x_2,y_1) & e(x_2,y_2) & \bullet\bullet & e(x_2,y_n) \\ \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \\ e(x_n,y_1) & e(x_n,y_2) & \bullet\bullet & e(x_n,y_n) \end{bmatrix}$$

Fig. 4.1: matrice che rappresenta l'immagine.

Il pixel è per definizione il singolo elemento della matrice *E*.
Nei calcolatori, le immagini vengono rappresentate come matrici di punti, i *pixel*, ai quali sono associati, a seconda della rappresentazione del colore, uno o più numeri.
L'uso del calcolatore ha inoltre favorito il nascere e lo svilupparsi della disciplina dell'*elaborazione digitale di immagini*, con molteplici fini ed innumerevoli campi di applicazione, quali il miglioramento della qualità, il restauro , la compressione, la visione artificiale. Quest'ultima branca comprende infine numerosi rami, come il riconoscimento di oggetti, il *tracking*, la ricostruzione tridimensionale, tutti accomunati dal fatto di basare la propria attività sull'analisi delle immagini.

4.2 Videocamera digitale OV7640

OV7640 è una camera digitale figura (4.2), prodotta da Omnivision [13], a colori basata su tecnologia CMOS; offre elevate prestazioni a fronte di bassi consumi e limitate dimensioni, che la rendono facilmente integrabile in dispositivi portatili quali telefoni cellulari, videogiochi ecc. La caratteristica principale di questo prodotto è l'assoluta versatilità in termini di possibili formati di uscita supportati, oltre a numerosi controlli per migliorare la qualità dell'immagine.

Fig. 4.2: videocamera digitale OV7640.

	Array Size	640 x 480 (VGA)
Power Supply	Core	2.5VDC ± 10%
	Analog	2.5VDC ± 4%
	I/O	2.25V to 3.3V
Power Requirements	Active	40 mW (30 fps, including I/O power)
	Standby	30 µW
Temperature Range	Operation	-10°C to 70°C
	Stable Image	0°C to 50°C
Output Formats (8-bit)		• YUV/YCbCr 4:2:2 • RGB 4:2:2 • Raw RGB Data
	Lens Size	1/4"
Maximum Image Transfer Rate	VGA	30 fps
	QVGA	60 fps
Sensitivity	B&W	3.0 V/Lux-sec
	Color	1.12 V/Lux-sec
	S/N Ratio	46 dB
	Dynamic Range	62 dB
	Scan Mode	Progressive/Interlaced
	Maximum Exposure Interval	523 x t_{ROW}
	Gamma Correction	0.45
	Pixel Size	5.6 µm x 5.6 µm
	Dark Current	30 mV/s
	Well Capacity	60 Ke
	Fixed Pattern Noise	< 0.03% of $V_{PEAK-TO-PEAK}$
	Image Area	3.6 mm x 2.7 mm
	Package Dimensions	11.43 mm x 11.43 mm

Fig. 4.3: dati di targa OV7640.

4.2.1 Specifiche

Elenchiamo brevemente le caratteristiche salienti della OV7640, per maggiori dettagli rimandiamo alla documentazione [14].

- Elevata sensibilità per operare in condizioni di bassa luminosità
- Alimentazione a 2,5 Volt
- Interfaccia SCCB (Serial Camera Control Bus)
- Immagini formati VGA (640*480), QVGA (320*240) nei formati :
    1. Raw RGB
    2. RGB (GRB 4:2:2)
    3. YUV (4:2:2)
    4. YCbCr (4:2:2)
- Funzioni di controllo sull'immagine:
    1. controllo esposizione (AEC)
    2. controllo del guadagno (AGC)
    3. bilanciamento del bianco (AWB)
    4. controllo luminosità (ABC)
    5. filtro (ABF) per disturbi a 60Hz
    6. calibrazione del livello del bianco (ABLC)
- Funzioni di controllo sul colore:
    1. *saturation*
    2. *hue*
    3. *gamma*
    4. *anti-blooming*
    5. *sharpness*
    6. *zero smearing*

In modalità *default* le funzioni sopraelencate sono gestite automaticamente in modo da trasmettere la miglior qualità di immagine al variare delle condizioni esterne.

Il *chip* contiene un set di registri, il quale tramite un tool fornito da *OmniVision* permette di scegliere le caratteristiche desiderate, imponendo in modo manuale i livelli ed i valori delle varie funzioni oltre al formato di uscita desiderato.

Per le operazioni di *setup* dei registri è necessario interfacciare OV7640 con un personal computer tramite il microcontrollore CY7C68013 figura(4.4) della *CYPRESS*[15] il quale permette di trasmettere al PC le immagini acquisite per apprezzarne i cambiamenti al variare del valore dei registri.

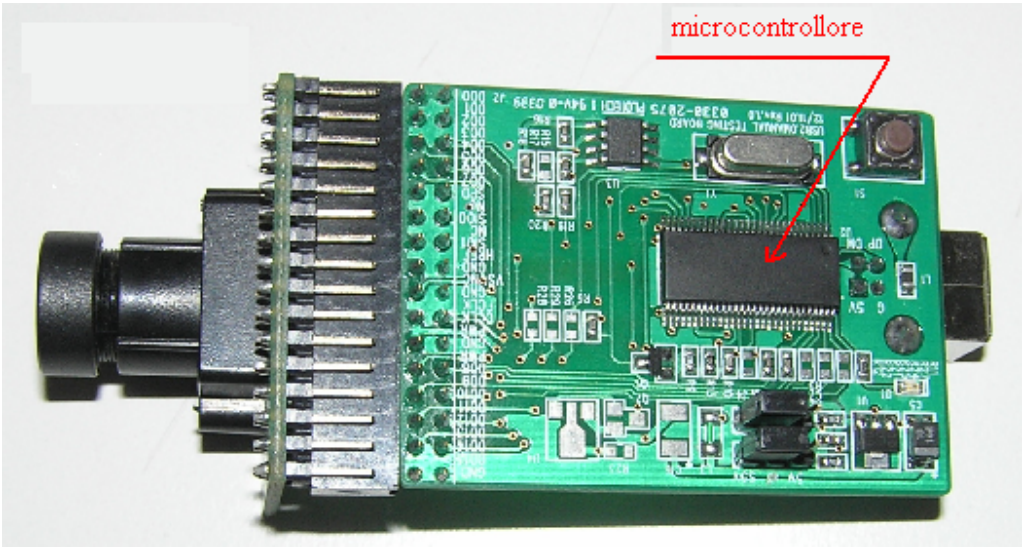


Fig. 4.4: microcontrollore per il controllo della videocamera.

4.2.2 Modalità operative

Una volta ottenute le specifiche desiderate è possibile memorizzare la configurazione in un file che è poi possibile caricare nel chip all'occorrenza.
In figura (4.5) mostriamo il *tool* per la modifica dei registri:

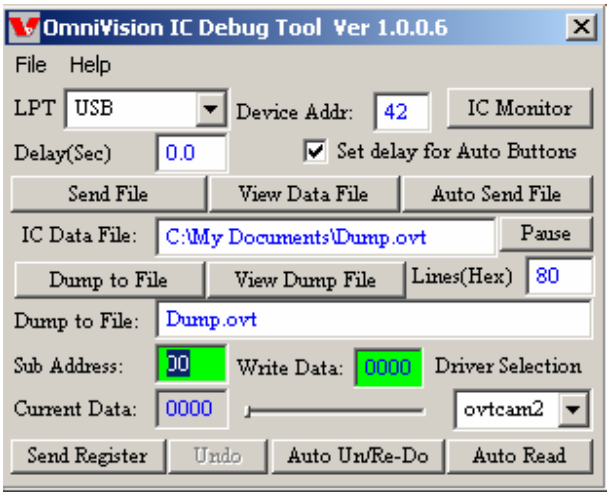


Fig. 4.5: OVTDTool è usato per leggere e scrivere in specifici registri all'interno della *camerachip* OV7640.

Per leggere il valore di uno specifico registro occorre inserire l'indirizzo nel campo ***Sub Address*** e il corrente valore apparirà nel campo ***Write Data***; se si vuole inserire nel chip uno specifico valore occorre digitarlo in ***Write Data*** e premere il tasto <tab>. Inoltre è possibile salvare la configurazione dei registri del chip in un file con i seguenti passi:

1. selezionare **NEW IC FILE** dal menù **FILE**;
2. aprire **Notepad** e salvare un file vuoto con nome ***.ovt;
3. nella casella **Dump to File** inserire ***.ovt e premere **Dump to File**.

A questo punto il file ***.ovt contiene i valori dei registri del chip OV7640, per visualizzarli occorre premere **View Dump File**.
E' anche possibile leggere i dati da un file e scriverli all'interno del chip, vediamo come:

1. selezionare **Open** dal menù **FILE**;
2. scegliere il nome del file che si vuole caricare;
3. premere su **Send File** per inviare i dati al chip.

4.2.3 Codifica Pixel

L'immagine di uscita è rappresentata come un array di byte come mostrato in figura (4.6):

Fig. 4.6: rappresentazione immagine in uscita da OV7640.

Ogni singolo pixel è codificato con due byte successivi come si vede in figura (4.7):

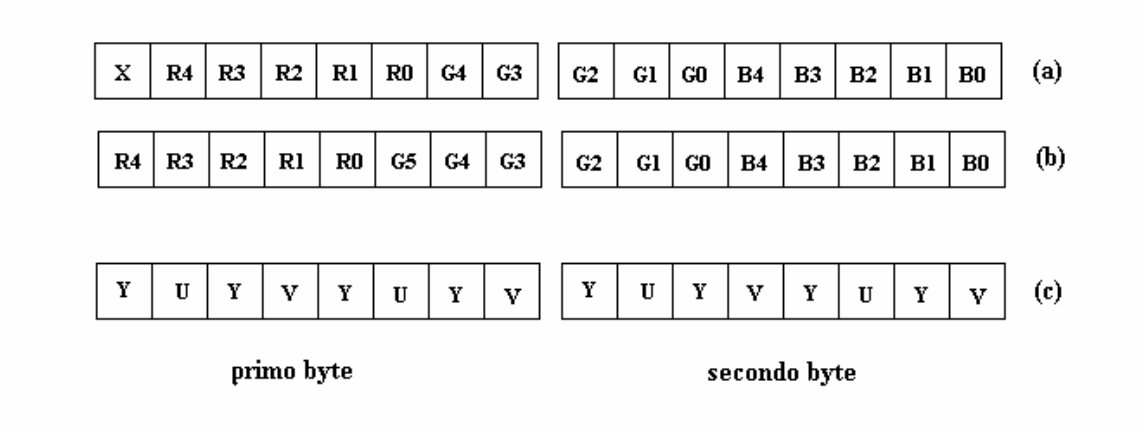


Fig. 4.7: (a) uscita RGB 555, (b) uscita RGB 565, (c) uscita YUV.

Nel nostro caso abbiamo preferito il formato RGB 555 al 565 per poter replicare in modo identico la realizzazione dei filtri, questo permette una estrema modularità del progetto.
Negli algoritmi di *corner detection* è stata scelta la codifica YUV ed in particolare la componente Y (luminanza) in quanto lavoriamo sulla scala di grigio.
Per ulteriori informazioni sui formati colore rimandiamo all'Appendice A.

Capitolo 5

Implementazione dei filtri

5.1 Introduzione

Con filtraggio di immagini si intende l'operazione con la quale si modifica l'immagine.
Esistono diverse tecniche per effettuare il filtraggio di immagini; noi discutiamo nel seguito elaborazioni locali utili per eliminare il rumore introdotto nel corso del processo di acquisizione.
Il filtraggio lineare di una immagine f di dimensione M*N con una maschera di dimensione $\left((m=2a+1)*(n=2b+1)\right)$ può essere descritto dalla seguente equazione (5.1):

$$g(x,y)=\sum_{s=-a}^{a}\sum_{t=-b}^{b} w(s,t)\, f(x+s,y+t) \qquad (5.1)$$

con $a=\frac{m-1}{2}$ e $b=\frac{n-1}{2}$

che fa riferimento ad una nomenclatura per pixel e coefficienti modificata come in figura (5.1), ancora per m=n=3:

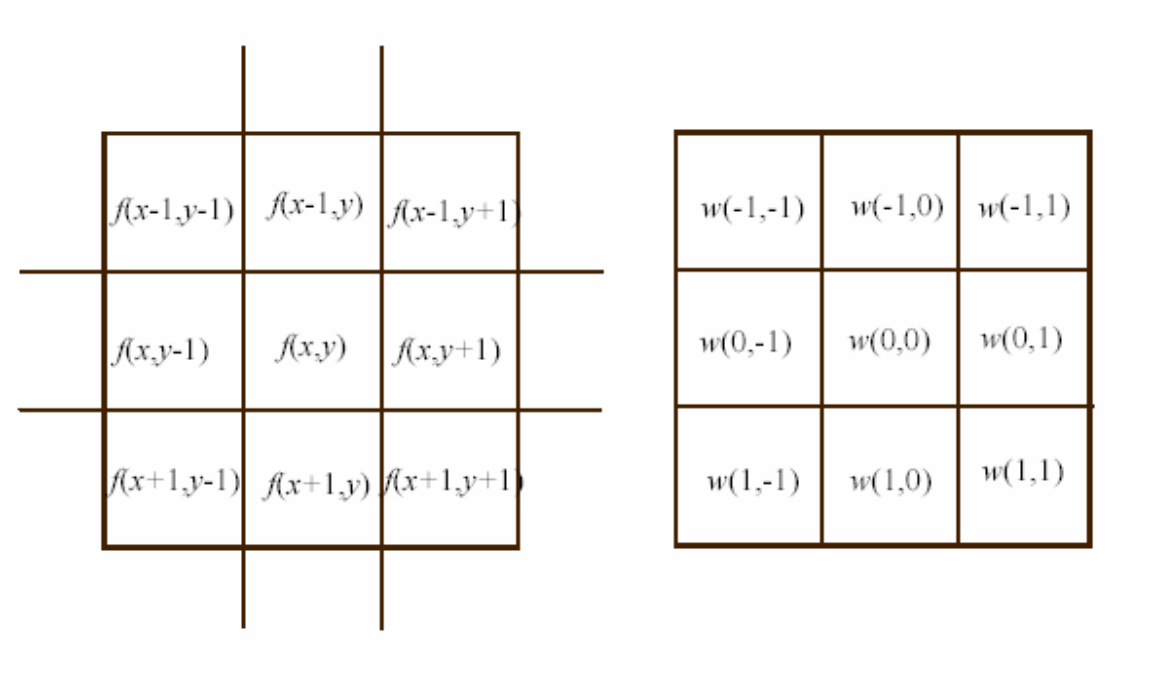


Fig. 5.1: visualizzazione pixel e coefficienti.

L'operazione è ripetuta per tutti i pixel dell'immagine. Questo processo è noto come *convoluzione* nel dominio della frequenza, da qui il termine di maschere di convoluzione.

Nota: attenzione particolare meritano i pixel di bordo, in quanto per essi occorre stabilire delle convenzioni per la definizione dell'intorno, che viene a cadere in parte fuori dall'immagine quando la maschera è vicino al bordo. La soluzione più semplice è limitare l'escursione del centro della maschera che supponiamo per semplicità quadrata, in modo che esso possa arrivare ad una distanza non inferiore a $(n - 1) / 2$ pixel dal bordo dell'immagine. Nel nostro caso abbiamo optato per una soluzione diversa: poiché i dati in ingresso scorrono sequenziali, svolgiamo la convoluzione di tutti i pixel e alla fine scartiamo i bordi dell'immagine ottenuta.

E' da notare che la dimensione delle strisce di pixel non corretti aumenta all'aumentare della dimensione della maschera.

Attraverso la convoluzione bidimensionale tra l'immagine originale e un'opportuna matrice di coefficienti otteniamo il risultato desiderato.

La convoluzione bidimensionale è definita dall'equazione (5.2):

$$T'_{x,y} = \sum_{i=-k}^{k} \sum_{j=-k}^{k} T_{x+i,y+j} * W_{i,j} \qquad (5.2)$$

Dove $T'_{x,y}$ è il valore del pixel convoluito, $T_{x,y}$ valore pixel attuale, $W_{i,j}$ coefficienti della maschera di convoluzione, $1<x<M$ e $1<y<N$, con M*N dimensione dell'immagine.

Per applicare la formula è richiesta la conoscenza del pixel corrente e dei suoi **m*n –1** pixel immediatamente vicini.

Supponendo di lavorare con **m=n=3** il numero di pixel da mantenere in memoria sono **(n-1)*N+n=2*N+3**.

Dal momento che tutti i pixel devono essere visitati, la maschera si sposta nella posizione adiacente fino a che l'intera immagine è stata visitata.

Nell'algoritmo da noi utilizzato anziché far scorrere la maschera di convoluzione sull'immagine, è quest'ultima che scorre all'interno della maschera come una sequenza continua di pixel.

Per il nostro lavoro è richiesta un'elaborazione di tipo *real-time*, ovvero avviene durante il processo di acquisizione senza dover memorizzare l'intera immagine in modo che il risultato sia disponibile riga per riga.

Passeremo ora ad analizzare la struttura che permette di ottenere questo tipo di filtraggio.

5.2 Realizzazione filtro con Simulink

Lo schema riportato in figura (5.2)[1] modella il diagramma di flusso associato ad un filtro lineare 2D assumendo la scansione per righe di una immagine con N colonne e M righe.

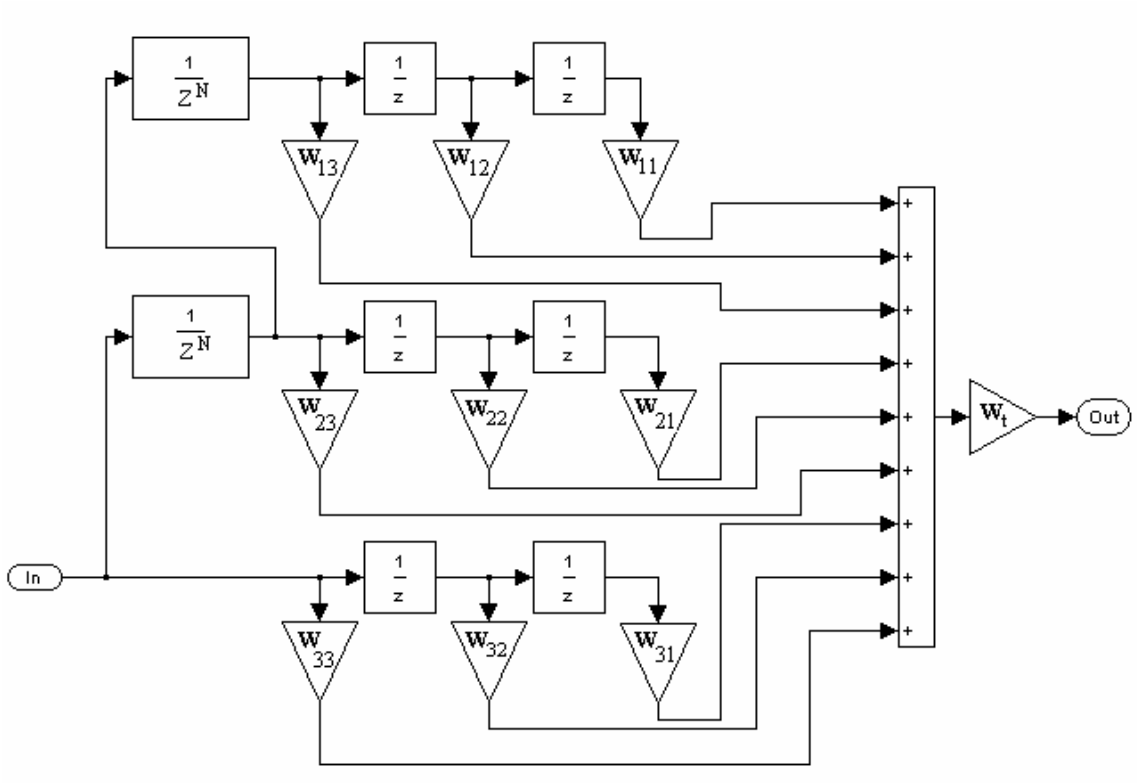


Fig.5.2: filtro generico di dimensione **3*3**.

I pixel entrano da **In** e scorrono le linee di ritardo fino a che il primo entrato viene moltiplicato per W_{11}. A questo punto in **Out** abbiamo il primo pixel elaborato che è $T'_{2,2}$ (la

[1] I modelli utilizzati per le simulazioni discusse in questa sezione abbiamo assunto l'impiego di dati espressi in formato *double*. L'effetto dell'implementazione a numeri interi sarà discussa nel seguito.

maschera è posizionata nell'angolo in alto a destra dell'immagine). Da questo punto in poi per ogni pixel entrante avremo $T'_{2,3}$, $T'_{2,4}$, ….. e così via fino ad arrivare a $T'_{m,n}$.

Considerando l'immagine come un vettore colonna costituito dalle righe trasposte della matrice originale, abbiamo perdita di informazione ai bordi poichè l'ultimo pixel della riga i-esima viene convoluito con il primo della riga i+1-esima. A causa di questo inconveniente è necessario scartare il bordo dell'immagine che avrà valori non significativi pertanto inutili.
Qualora la maschera sia scrivibile come prodotto colonna-riga figura (5.3),

$$W_t \begin{bmatrix} C_1 \\ C_2 \\ C_3 \end{bmatrix} \begin{bmatrix} R_1 & R_2 & R_3 \end{bmatrix} = W_t \begin{bmatrix} W_{11} & W_{12} & W_{13} \\ W_{21} & W_{22} & W_{23} \\ W_{31} & W_{32} & W_{33} \end{bmatrix}$$

Fig. 5.3: prodotto colonna-riga con W_t scalare.

.

posso sfruttare la proprietà di linearità della convoluzione, la formula (5.2) si può scrivere come in (5.3):

$$T'_{x,y} = \sum_{i=1}^{k} \left\{ \sum_{j=1}^{k} W_j^T * T_{x+i,y+j} \right\} * W_i \qquad (5.3)$$

mentre lo schema (5.2) si modifica come mostrato in figura (5.4):

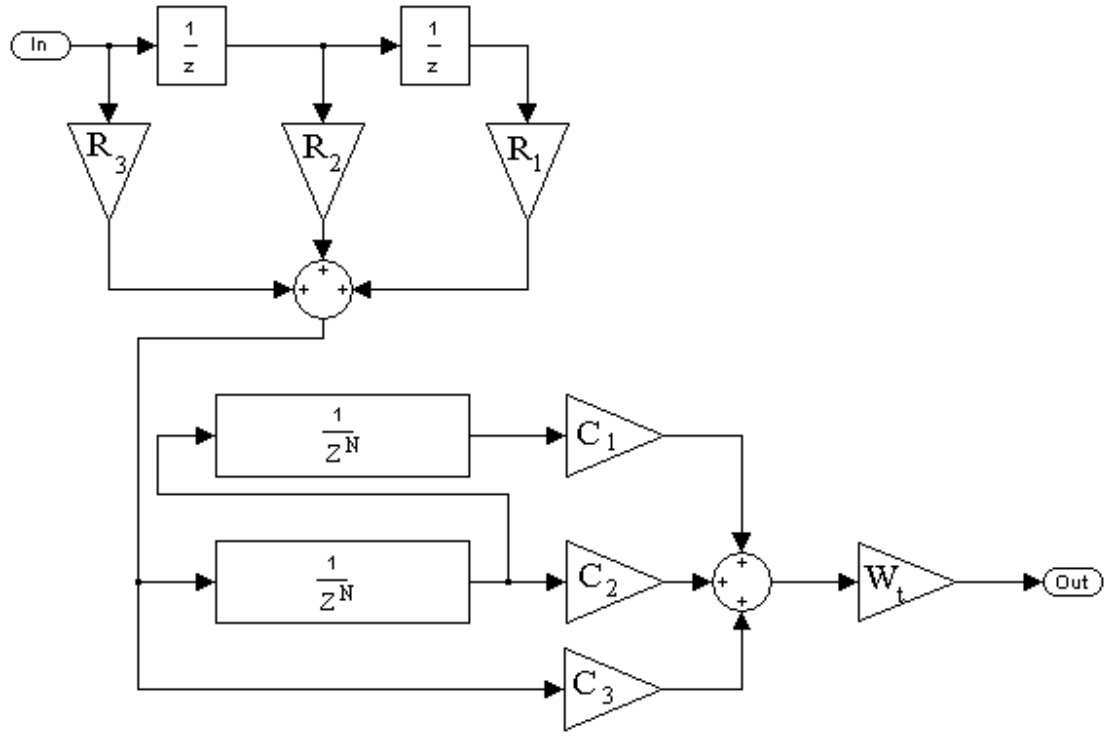


Fig. 5.4: filtro che utilizza la separabilità della maschera di convoluzione.

Il vantaggio del secondo metodo riguarda la diminuzione del numero di prodotti da compiere ovvero della quantità di risorse impiegate.

Si può notare che un filtro con maschera $n*n$ richiede normalmente n^2 prodotti, mentre con questo accorgimento si passa a $2n$.

Per verificare che i due schemi svolgessero le medesime funzioni, li abbiamo eseguiti in Matlab constatando risultati analoghi nelle prove simulative.

I test sono stati eseguiti implementando tre tipi di filtri: Sobel, Prewitt, Smoothing come in figura (5.5).

-1	0	1
-2	0	2
-1	0	1

(a)

-1	0	1
-1	0	1
-1	0	1

(b)

1/16 *

1	2	1
2	4	2
1	2	1

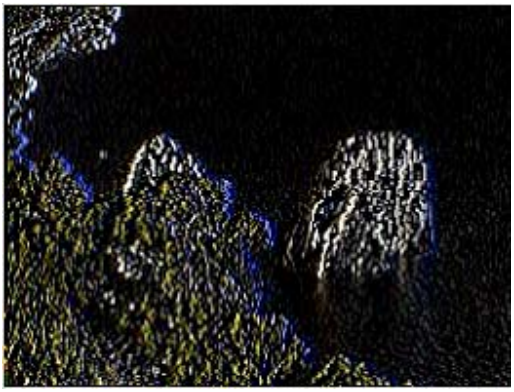
(c)

Fig. 5.5: (a) filtro Sobel, (b) filtro Prewitt, (c) filtro Smoothing.

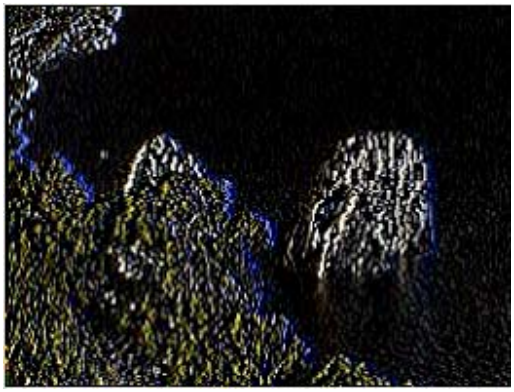
(a)

(b)

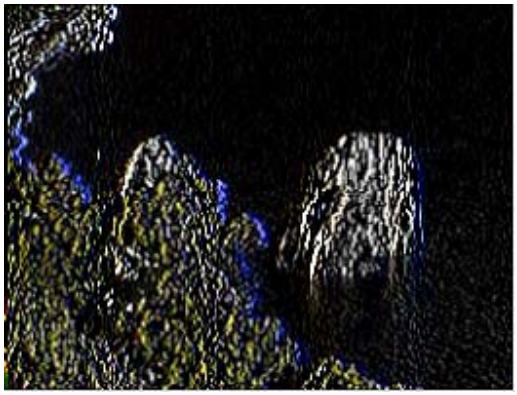
(c)



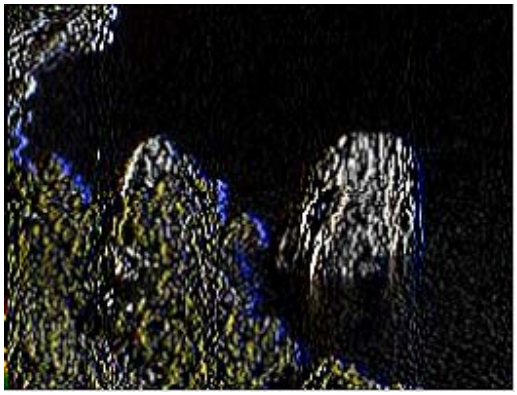
(d)



(e)



(f)



(g)

Fig. 5.6: (a) immagine originale, (b) (d) (f) elaborate con lo schema di figura (5.2), (b) filtro di *smoothing*, (d) filtro di *Prewitt,* (f) filtro di *Sobel*.
(c) (e) (g) elaborate con lo schema di figura (5.4), (c) filtro di *smoothing*, (e) filtro di *Prewitt,* (g) filtro di *Sobel*.

In figura (5.6) mostriamo i risultati ottenuti dalle simulazioni in ambiente *Simulink.*
Avendo in ingresso immagini espresse nel formato RGB, la convoluzione deve essere realizzata per ciascuna delle tre componenti separatamente. Questo è stato possibile mediante la funzione Matlab riportata in Appendice C.1.

5.3 Realizzazione filtro in aritmetica a precisione finita

5.3.1 Generalità

Gli algoritmi di elaborazione numerica utilizzano dati rappresentati con sequenze binarie in registri di lunghezza finita, il che genera problemi di vario genere.
Ogni volta che si presenta la necessità di convertire un segnale analogico in digitale il primo passo è il campionamento; successivamente, nel processo di quantizzazione, sono resi disponibili soltanto un numero finito di possibili valori per ogni campione. Il numero di possibili combinazioni binarie è limitato dalla risoluzione dei convertitori A/D impiegati e dal formato dei dati utilizzati.
Un problema si manifesta quando si devono effettuare operazioni matematiche sui dati quali moltiplicazione o addizione che generalmente portano a numeri che richiedono ulteriori bit per la loro rappresentazione ("overflow"). Per fronteggiare questo effetto si richiede che i dati di ingresso siano sufficientemente piccoli in modo da evitare la possibilità di overflow, se questo non è possibile si adattano i dati alla lunghezza desiderata attraverso arrotondamento o troncamento figura (5.7). Questi metodi portano minor precisione rispetto alla versione estesa.

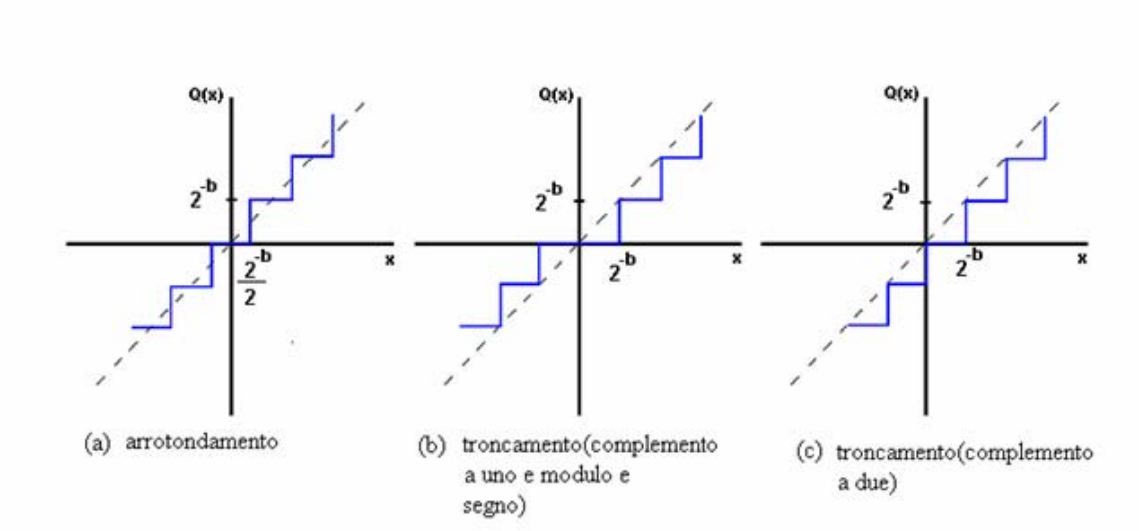


Fig. 5.7: relazioni non lineari rappresentanti arrotondamento e troncamento.

5.3.2 Realizzazione filtri

L'avvicinamento alla struttura *hardware* ha portato alla realizzazione dei precedenti schemi con aritmetica a precisione finita, utilizzando il Toolbox Fixed-Point Blockset di Matlab. Per ulteriori informazioni rimandiamo all'Appendice B.

Questa libreria permette di creare sistemi a tempo discreto che usano aritmetica *fixed point*; tramite Simulink si possono poi simulare gli effetti comunemente incontrati utilizzando questa aritmetica.

Lo schema di figura (5.2), analizzato in questo contesto, si presenta come mostrato in figura (5.8).

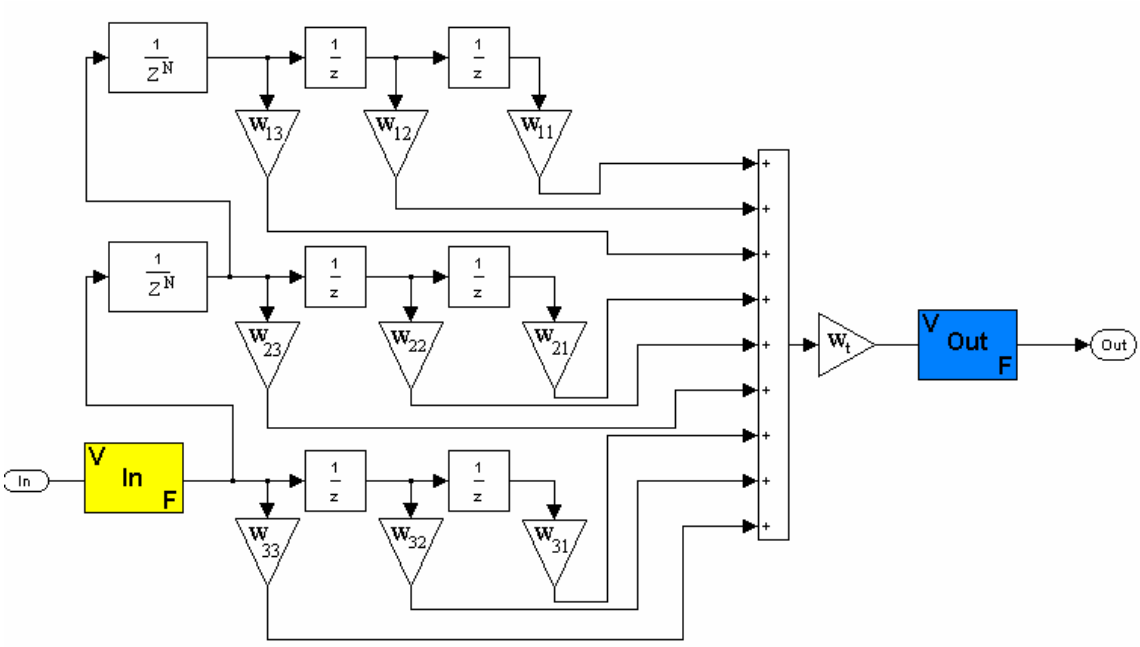


Fig.5.8: filtro realizzato in Fixed-Point, il blocco giallo converte i valori *double* in *fixed point* mentre il blu viceversa.

Nella costruzione di tale schema si deve tener conto che i dati hanno lunghezza finita e posizione fissa della virgola. Questo comporta limitazione nel *range* di rappresentabilità ed impone al progettista di stimare a priori i dati impiegati all'interno dello schema.
Il blocco giallo di figura (5.8) converte i dati *double* in *fixed point*. Con doppio *click* sul simbolo si apre il *dialog box* di figura (5.9), da cui si possono definire le proprietà da assegnare al dato.

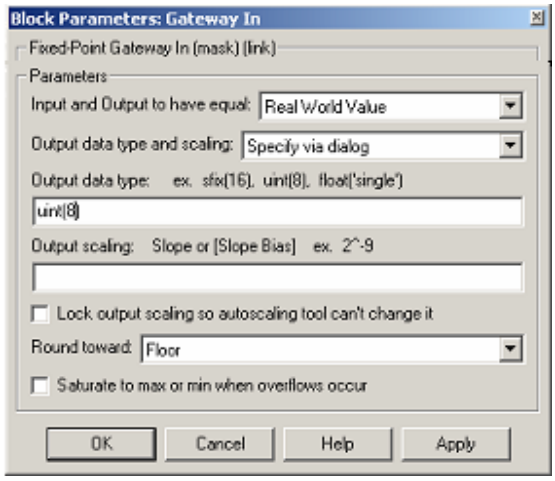


Fig. 5.9: *dialog box* Simulink.

In figura (5.9) è stata posta l'uscita di tipo uint(8), intero senza segno 8 bit.
Particolare attenzione deve essere posta sui blocchi moltiplicatori in quanto, qualora la costante fosse frazionaria, il limitato numero di cifre decimali potrebbe causare perdita di precisione. Esaminiamo il filtro di *smoothing*: il fattore W_t di normalizzazione (1/16) richiede almeno quattro cifre decimali per essere rappresentato.
La possibile perdita di precisione può non essere particolarmente rilevante sul risultato di un singolo filtro, ma in caso di algoritmi complessi, come quelli di *corner detection*, piccoli errori nei primi stadi possono ripercuotersi pesantemente sul risultato finale.

Capitolo 6

Corner detection di Forstner

6.1 Analisi dell'immagine

In questo capitolo vengono affrontati metodi di analisi per la ricerca di spigoli in immagini rappresentate in bianco e nero.

Il nostro cervello, che gode di proprietà molto più evolute di una macchina, è in grado di estrarre da un'immagine caratteristiche significative. Per descrivere una scena in modo esauriente attraverso algoritmi, abbiamo dovuto operare una serie di elaborazioni.

Una prima elaborazione, solitamente definita pre-elaborazione, consiste nel filtraggio passa basso gaussiano allo scopo di attenuare il rumore alle alte frequenze spaziali..

Un'elaborazione successiva porta alla ricerca delle caratteristiche salienti di una scena, solitamente i contorni.

Assumiamo che l'immagine $F(x,y)$ sia affetta da rumore in modo più o meno omogeneo.

In questo caso lo si può ridurre sostituendo ogni pixel con la media pesata dei pixel nell'intorno. Il suddetto processo è noto come *smoothing* o *blurring*.

Il filtro passa basso risulta ideale e non fisicamente realizzabile. Per questo si adopera un filtro con risposta all'impulso pari alla derivata di una gaussiana.

La funzione matematica della gaussiana è la seguente:

$$G_\sigma(x,y) = \frac{1}{2\pi\sigma^2} e^{\left(-\frac{(x^2+y^2)}{2\sigma^2}\right)} \tag{6.1}$$

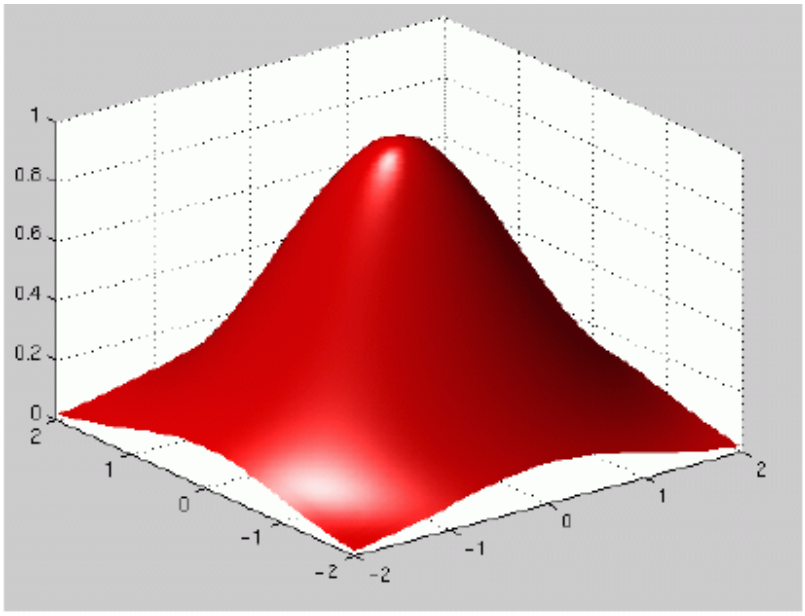


Fig. 6.1: plotting della gaussiana.

Come si può dedurre dalla figura (6.1), la gaussiana forma una media pesata tale che i pixel al centro assumano un peso maggiore di quelli in periferia.

L'uscita del filtro inoltre ha il massimo proprio in corrispondenza della discontinuità dello scalino.

Un'applicazione di grande utilizzo nel trattamento di immagini è il filtraggio con *kernel* gaussiano. Un accurato studio dei parametri della gaussiana, in particolar modo la scelta dell'apertura, è fondamentale per ottenere un buon filtraggio.

Se ad esempio l'apertura risulta più piccola di un *pixel*, lo smussamento avrà un piccolo (se non impercettibile) effetto perchè i pesi per tutti i *pixel* lontani dal centro saranno molto piccoli.

Al contrario una apertura non trascurabile porterà i *pixel* adiacenti a quello considerato ad avere pesi rilevanti. Questo vuol dire avere una grande dipendenza dai *pixel* adiacenti e la scomparsa del rumore, con lo svantaggio di perdere lievemente i dettagli.

Infine, una deviazione *standard* eccessiva causa la perdita di dettagli (probabilmente) importanti e sicuramente del rumore.

6.2 Calcolo del gradiente di un'immagine

Una parte fondamentale dell'algoritmo di *Forstner* è la ricerca dei contorni dell'immagine mediante il calcolo delle componenti lungo x e y del gradiente dell'intensità di grigio dell'immagine:

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix}^{T} \tag{6.2}$$

Lavorando con funzioni discrete, il calcolo delle componenti del gradiente (derivate parziali) si può approssimare con la differenza finita tra *pixels* adiacenti lungo la direzione scelta.
Una definizione usuale di derivata prima approssimata di una $f(x)$ è data dalle formule:

$$\begin{cases} \frac{\partial f}{\partial x} = f(x+1) - f(x) \\ \frac{\partial f}{\partial y} = f(y+1) - f(y) \end{cases} \tag{6.3}$$

La maniera più semplice di generare una approssimazione discreta del gradiente prende in considerazione la differenza mobile dei *pixel* nelle due direzioni figura (6.2):

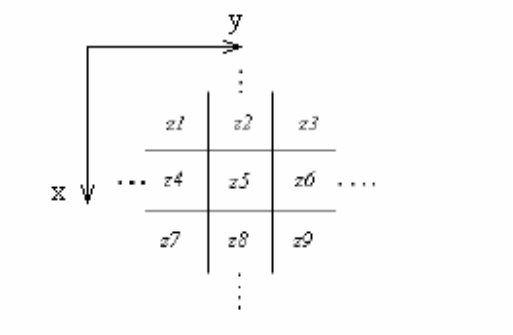


Fig. 6.2: matrice di *pixel*.

dove le componenti del gradiente nel *pixel* z5 sono:

$$G_x = z_8 - z_5 \quad e \quad G_y = z_6 - z_5 \tag{6.4}$$

Per mantenere una struttura analoga al filtraggio di *smoothing*, il calcolo (6.3) può essere operativamente eseguito convoluendo l'immagine originale con le maschere in figura (6.3):

0	0	0
0	-1	1
0	0	0

$\frac{\partial f}{\partial x}$

0	1	0
0	-1	0
0	0	0

$\frac{\partial f}{\partial y}$

Fig. 6.3: maschere per il calcolo della derivata prima approssimata.

6.3 Filtro di Sobel

Un'importante applicazione della teoria vista sopra è quella proposta da *Sobel*. Essa si basa principalmente sullo studio dell'approssimazione del modulo del gradiente; data una funzione $f(x, ,y)$, il gradiente è il vettore:

$$\nabla f = \begin{bmatrix} G_x \\ G_y \end{bmatrix} = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix} \tag{6.5}$$

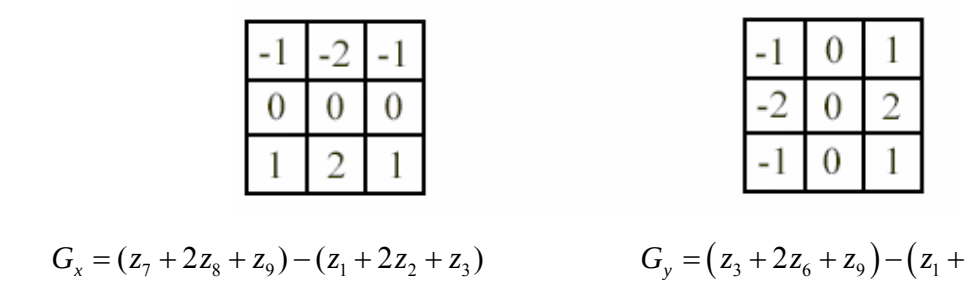
il cui modulo è:

$$|\nabla f| = \left[G_x^2 + G_y^2 \right]^{1/2} \approx |G_x| + |G_y| \tag{6.6}$$

L'approssimazione (6.7), mediante somma dei valori assoluti delle componenti, rende computazionalmente meno oneroso il calcolo del gradiente, in quanto si evitano calcoli quali elevamento a potenza e radice quadrata.

L'operatore di *Sobel* è in grado di effettuare simultaneamente sia la differenziazione lungo una direzione, sia una media spaziale lungo la direzione ortogonale. Questo riduce fortemente la sensibilità al rumore del filtro.

Nel rispetto della teoria di *Sobel* le maschere di figura (6.3) si modificano come in figura (6.4):



-1	-2	-1
0	0	0
1	2	1

$$G_x = (z_7 + 2z_8 + z_9) - (z_1 + 2z_2 + z_3)$$

-1	0	1
-2	0	2
-1	0	1

$$G_y = (z_3 + 2z_6 + z_9) - (z_1 + 2z_4 + z_7)$$

Fig. 6.4: maschere di *Sobel*.

6.4 Schema dell'algoritmo di Forstner

Lo studio dei filtri visti precedentemente in questo capitolo ci ha permesso di utilizzarli per costruire un modello simulativo dell'algoritmo. Richiamiamo la formula per la *corner detection*, (6.7), trattata nel capitolo 2:

$$C = \frac{\langle I_x^2 \rangle \langle I_y^2 \rangle - \langle I_x I_y \rangle^2}{\langle I_x^2 \rangle + \langle I_y^2 \rangle} \tag{6.7}$$

e illustriamo la sua rappresentazione schematica figura (6.5) attraverso *Simulink* che ci ha permesso di testarne il comportamento.

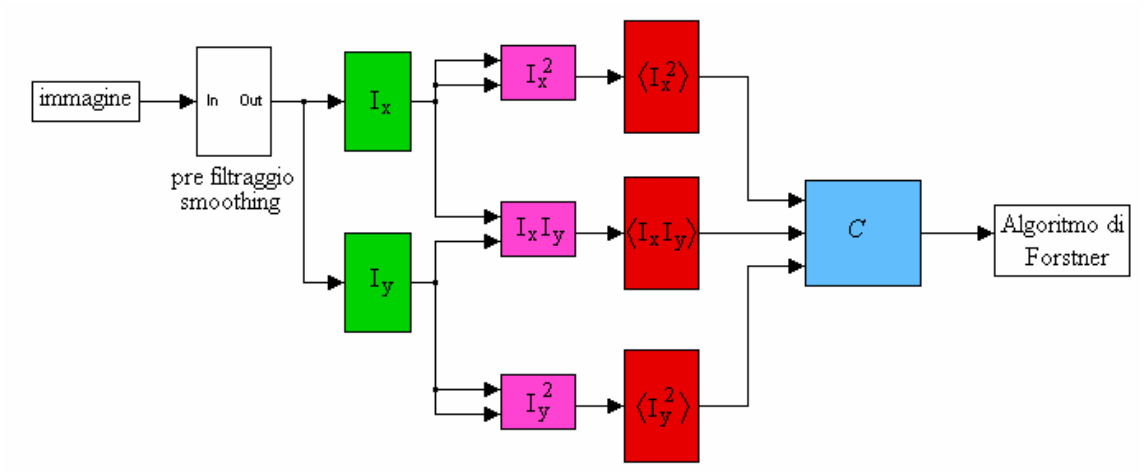


Fig. 6.5: schema algoritmo *Forstner* con *Simulink*: i blocchi verdi eseguono i filtri di *Sobel*, quelli rossi *smoothing* con gaussiana.

Prima di poter applicare all'immagine l'algoritmo, è necessario effettuare un preliminare filtraggio di *smoothing*, in seguito occorre applicare il filtro di *Sobel* in modo da ottenere le componenti necessarie per il calcolo come richiesto in (6.7).
Fondamentali risultano i filtri gaussiani (blocchi rossi) posti in ingresso all' unità di calcolo, i quali eliminano eventuali discontinuità introdotte dal filtraggio di *Sobel* (blocchi verdi) ed amplificate nel successivo stadio di elaborazione (blocchi magenta).

Operativamente sono stati eseguiti tramite un filtro gaussiano del tipo di figura (6.6):

$1/52 *$

1	1	2	1	1
1	2	4	2	1
2	4	8	4	2
1	2	4	2	1
1	1	2	1	1

Fig. 6.6: maschera gaussiana **5*5**.

Questa maschera è stata testata in una precedente tesi sviluppata all'interno del *Maclab* e risulta particolarmente adatta per immagini naturali.
Il blocco di colore azzurro di figura (6.5) esegue operativamente il calcolo (6.7), vediamone in dettaglio il funzionamento come illustrato in figura (6.7):

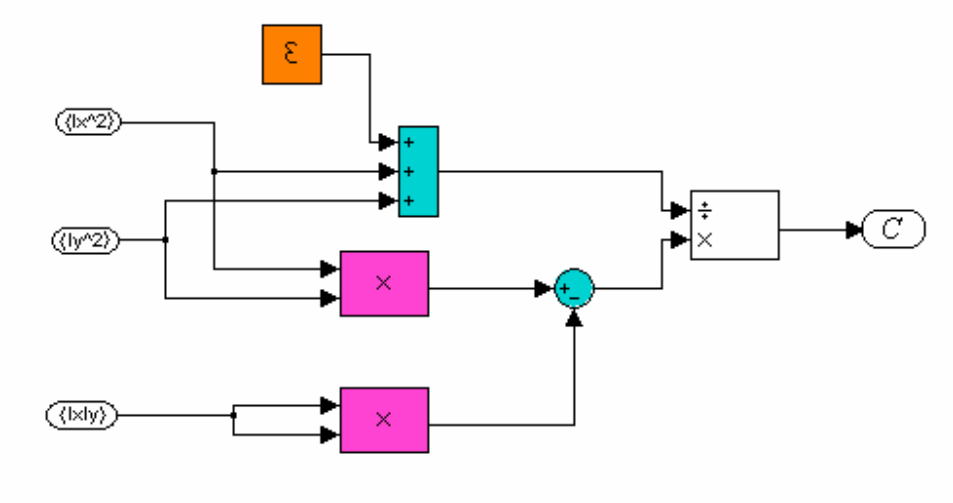


Fig. 6.7: blocco di calcolo della *corner detection*.

Poniamo l'attenzione sulla costante ε (blocco arancione) la quale non compare nella formula teorica (6.7), ma deve essere inserita nello schema per evitare la divisione per zero; questo accade qualora si proceda all'analisi di una zona dell'immagine uniforme, in quanto in queste zone le componenti del gradiente sono nulle.

6.5 Risultati simulativi

Illustriamo la procedura necessaria per simulare il funzionamento dell'algoritmo proposto. Scegliere un'immagine formato QVGA (320*240) come ingresso, nell'esempio utilizziamo la figura (6.8).

Fig 6.8: immagine da elaborare.

Per mezzo del listato di figura (6.9) si prepara l'immagine da inviare al simulatore.

```
image_gray=rgb2gray(image_RGB); %conversione dell'immagine in scala di grigi

image_gray=image_gray';         %dispone i pixel in un vettore di
image_gray=image_gray(:);       %dimensioni 320*240


image_gray=double(image_gray);  %converte i valori in double per
                                %poterli leggere da simulink

t=size(image_gray);             %crea un vettore di dimensione 320*240
t=(1:t)';                       %inizializzo i valori da 1 a (320*240)

image_gray=[t image_gray];      %prepara i dati per essere letti da
                                %simulink
```

Fig. 6.9: listato *Matlab* che converte l'immagine dal formato RGB a scala di grigi.

Al termine della simulazione è necessario ricostruire la matrice dell'immagine a partire da un *array* di valori. Questo è possibile attraverso il listato di figura (6.10).

```
corneress=reshape(corneress,320,240);  %costruiscono la matrice a
corneress=corneress';                  %partire da un array di valori
mesh(corneress);                       %visualizza il risultato
```

Fig. 6.10: listato *Matlab* per visualizzazione risultato.

In figura (6.11) è mostrato il risultato ottenuto eseguendo i passi sopra riportati.

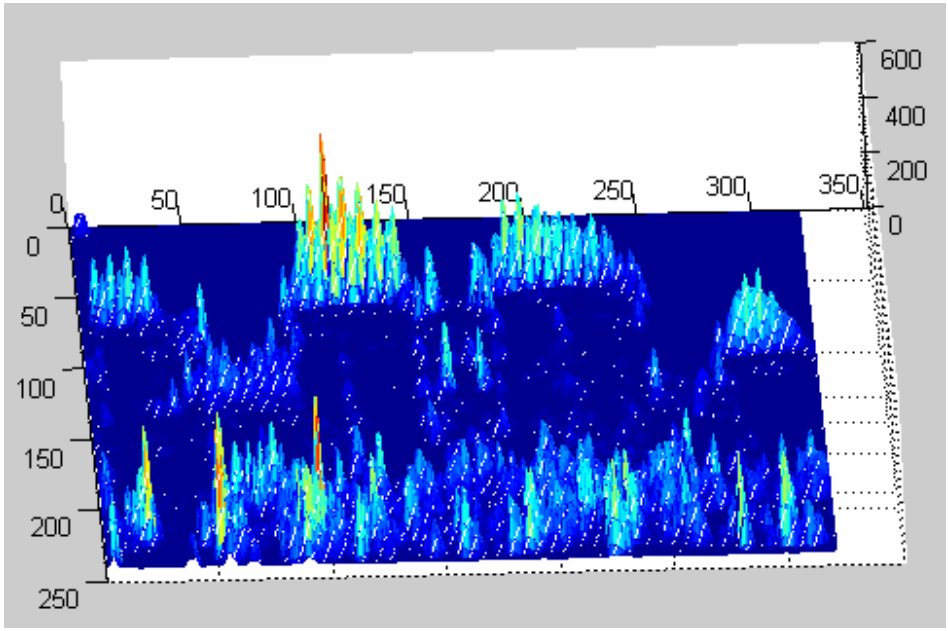
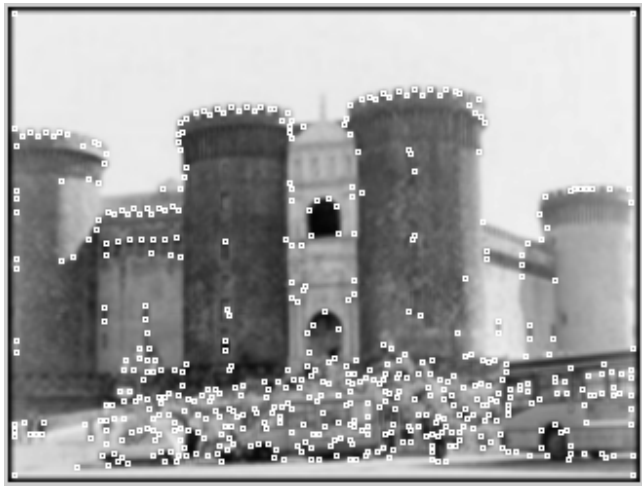


Fig. 6.11: risultato immagine elaborata con dati in virgola mobile.

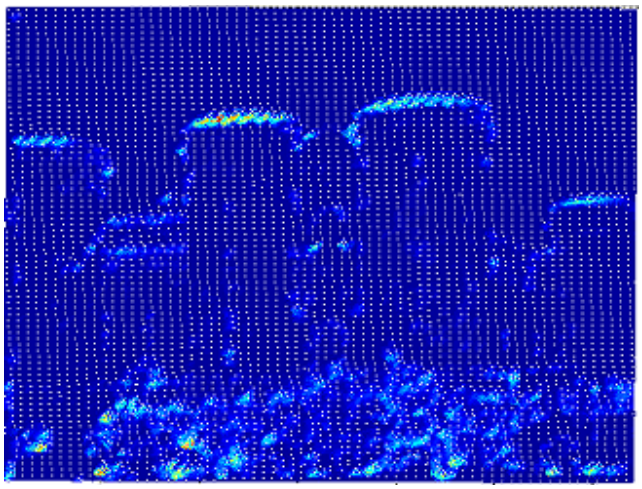
6.5.1 Confronto con un algoritmo di corner detection interamente sviluppato via software

Utilizzando la versione software dell'algoritmo di *Forstner*, sviluppata in una precedente tesi all'interno del *Maclab*, è stato possibile confrontare i risultati da noi ottenuti per verificarne il corretto comportamento.

La figura (6.12) mostra i *corner* trovati con l'algoritmo *software*.



(a)



(b)

Fig. 6.12: (a) risultato ottenuto via *software*, (b) risultato ottenuto dalla nostra simulazione.

Si può notare che le due tecniche evidenziano la maggior parte di *corner* presenti, ma in più la nostra simulazione fornisce un'ulteriore informazione: l'entità dei *corner*. Infatti dove i picchi sono più alti i *corner* sono più pronunciati.

Capitolo 7

Simulazione dell'algoritmo di Forstner con aritmetica a precisione finita

In questo capitolo tratteremo il comportamento dell'algoritmo con il *toolbox Fixed-Point Blockset* per analizzare i risultati che ci attenderemo dalla realizzazione *hardware*.
Importante è stato considerare il formato dei dati in uscita dalla telecamera ed in particolare la componente Y (luminanza) della codifica YUV.
Come già visto nel capitolo 4, i *pixel* sono rappresentati mediante 256 livelli di grigio (Y = 8 *bit*), e questo ci serve per decidere a priori la dimensione dei dati nei successivi *step* di elaborazione.

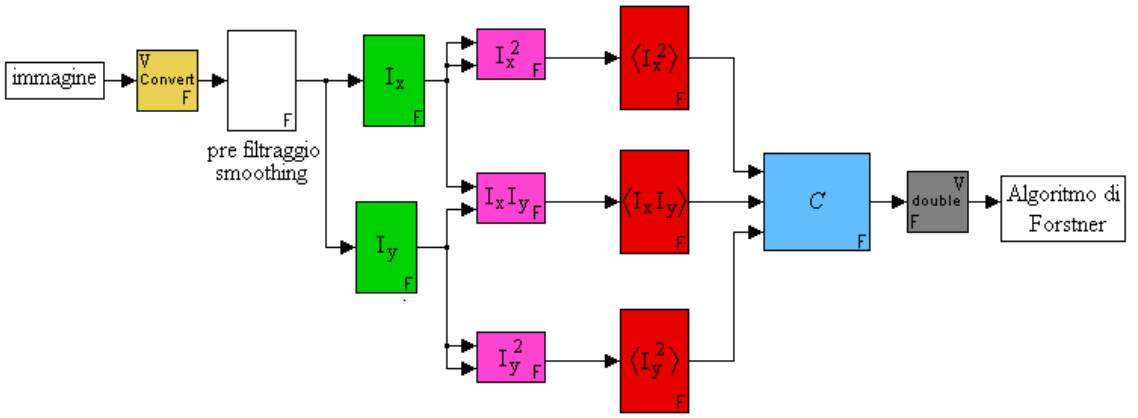


Fig. 7.1: schema realizzato con blocchi *fixed-point*.

In questo contesto necessita di particolare attenzione l'esecuzione delle operazioni, poiché il limitato range di rappresentabilità pone dei limiti rispetto alla versione con dati di tipo *double*.
Passiamo ad analizzare in dettaglio il comportamento dei dati nei vari stadi di elaborazione.

7.1 Dati in uscita dalla videocamera

Il primo blocco dello schema, che non fa parte dell'algoritmo vero e proprio, è utilizzato per convertire l'immagine nel formato a 8 bit, vedi figura (7.2).

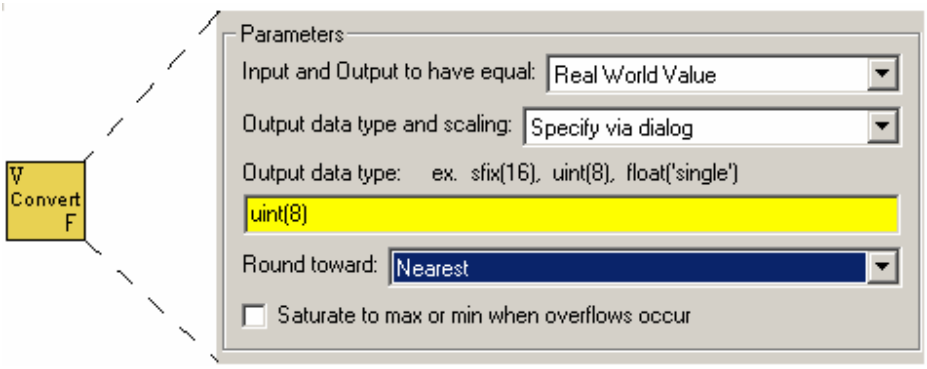


Fig. 7.2: parametri del blocco di conversione *double fixed-point*.

Il campo evidenziato in giallo mostra il formato dei dati in uscita: (*uint*(8)) indica numeri interi a 8 bit senza segno. Il campo blu permette di scegliere il tipo di arrotondamento, in quanto un numero reale può avere anche decimali che nella codifica *uint*(8) non sono previsti. Nel nostro caso la scelta è ricaduta su *Nearest*, che arrotonda al valore intero più vicino.
Terminata la conversione, i dati sono pronti per essere elaborati dall'algoritmo di *Forstner*.

7.2 Filtraggio preliminare

Per eliminare eventuali rumori presenti nell'immagine è necessario eseguire un filtraggio passa basso con una maschera gaussiana. Anche in questo caso occorre prestare attenzione per poter rappresentare in modo corretto i risultati in uscita da questo livello.

Il filtraggio è stato eseguito utilizzando una maschera del tipo mostrato in figura (7.3):

$1/52 *$

1	1	2	1	1
1	2	4	2	1
2	4	8	4	2
1	2	4	2	1
1	1	2	1	1

Fig. 7.3: maschera usata per il pre-filtraggio.

Come si vede dalla figura (7.3), i coefficienti appartengono all'insieme:

$$\{ 1, 2, 4, 8 \}$$

questo significa che avendo dati in ingresso *uint*(8) i prodotti risulteranno come rappresentati in tabella (7.4):

Dati in ingresso	**Coefficienti**	**Dati in uscita**
Uint(8)	1	Uint(8)
Uint(8)	2	Uint(9)
Uint(8)	4	Uint(10)
Uint(8)	8	Uint(11)

Tabella 7.4

Vediamo in figura (7.5) come inserire tali dati nei relativi blocchi moltiplicatori di *Simulink* :

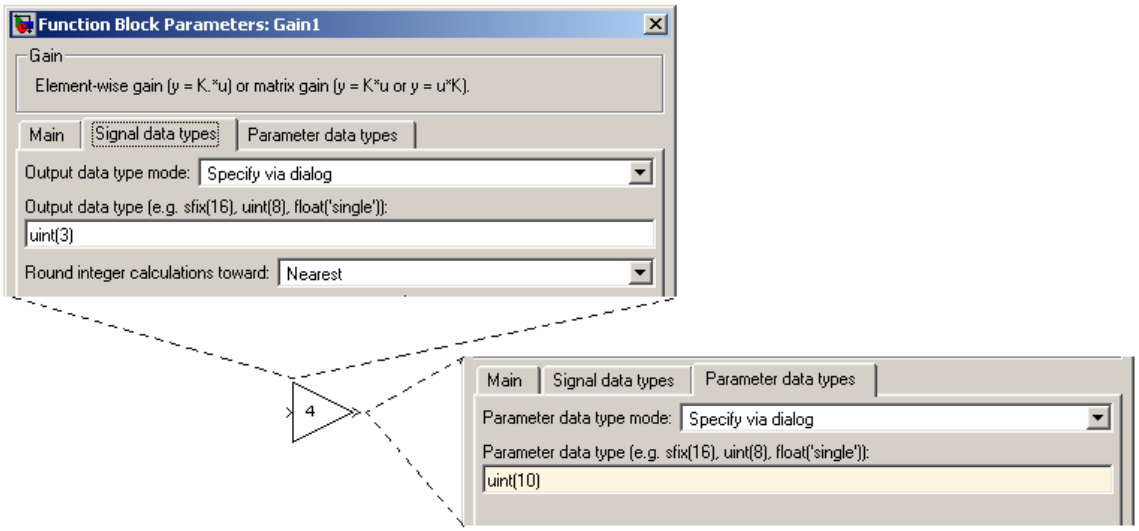


Fig. 7.5: esempio di assegnamento parametri per blocco moltiplicatore con coefficiente 4.

A valle di queste considerazioni occorre sistemare il coefficiente di normalizzazione 1/52.
Per rappresentare questo coefficiente sono necessari numerosi bit a destra del *radix point* : 1/52 = 0.0192, quindi scegliendo una rappresentazione binaria a 16 bit otteniamo 0.0000010011101010 = 0.01916. L'errore commesso con questo troncamento è pari a:

$$\varepsilon = 0.0192 - 0.01916 = 0.00004 \tag{7.1}$$

che è un valore sufficientemente piccolo tale da soddisfare le nostre richieste.

Per ottenere questo risultato si deve, dopo aver aperto il *dialog box* (doppio *click*), inserire i dati come evidenziato in figura (7.6).

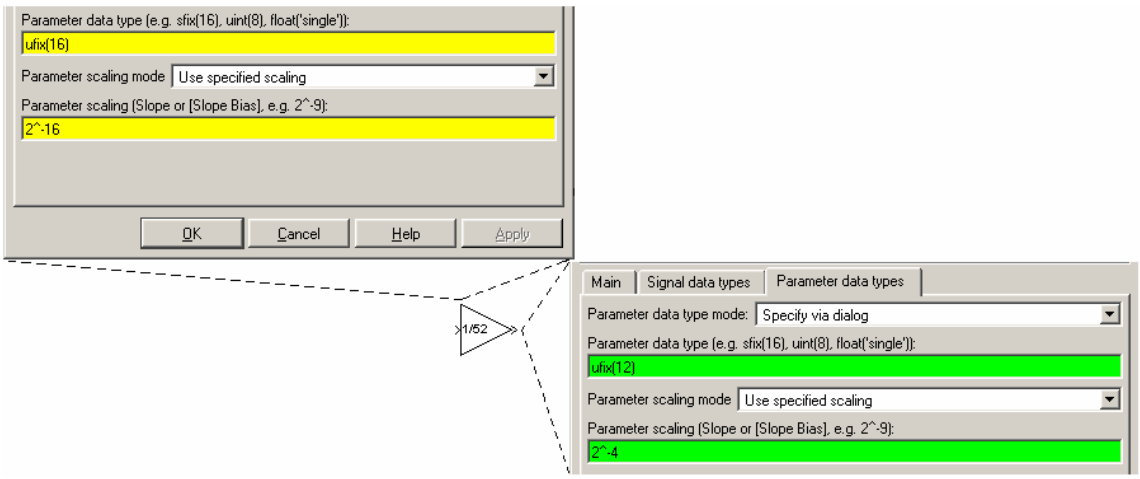


Fig. 7.6: esempio assegnamento parametri con coefficienti frazionari.

Dopo la normalizzazione il risultato ricade nel *range* $[0 \div 255]$ quindi, non avendo più solo valori interi, dovremo aggiungere delle cifre a destra della virgola. Teoricamente le cifre da inserire sono 16, quante quelle utilizzate per il coefficiente di normalizzazione. Tuttavia, operativamente tronchiamo a 4 cifre dopo la virgola e ciò si nota dai campi di colore verde di figura (7.6). L'errore che si commette con questa scelta è pari a:

$$-\left(2^{-4}-2^{-16}\right)=-0.0625\le E_T\le 0 \qquad (7.2)$$

7.3 Calcolo delle componenti del gradiente

Il successivo passo di elaborazione consiste nell'eseguire il calcolo del gradiente con il filtro di *Sobel*. I coefficienti del filtro appartengono all'insieme:

$$\{ -2, -1, 0, 1, 2\}$$

quindi occorre introdurre la rappresentazione dei numeri negativi. Per ulteriori informazioni rimandiamo all'Appendice B.2.

Per trattare numeri negativi occorre inserire *sfix* nel campo riferito al *data type* del *dialog box*. In questo stadio entrano dati di tipo *ufix*(12) con scalamento 2^{-4} (8 bit di parte intera e 4 di parte frazionaria) e in uscita avremo s*fix*(13), in cui il bit aggiuntivo è dovuto al fatto di dover trattare numeri negativi rappresentati in complemento a due.

7.4 Prodotti delle componenti del gradiente

Lo *step* composto dai blocchi color magenta nello schema (7.1) si occupa dell'elevamento a potenza e del prodotto fra le componenti.

Anche queste operazioni richiedono l'aggiustamento dei dati in uscita, che avviene nello stesso modo che é stato spiegato nei paragrafi precedenti.

I valori di uscita per i blocchi che eseguono il quadrato delle componenti devono essere di tipo *ufix*(24), con scalamento 2^{-8}, ed anche il blocco che esegue il prodotto fra le due componenti I_x *e* I_y, che sono di tipo sfix(25), ha scolamento 2^{-8}.

.

7.5 Filtraggio gaussiano

La successiva operazione utilizza i coefficienti visti in figura (7.3), che sono già stati impiegati nel filtraggio preliminare come descritto nel paragrafo 7.2.
I due filtraggi si differenziano per il tipo di dati: in questa fase trattiamo dati di tipo *ufix*(24) e *sfix*(25) con scolamento 2^{-8}.
Per le dimensioni dei dati in uscita notiamo che il valore minimo consentito in ingresso è pari a 1/256 (con scalamento 2^{-8}) moltiplicato per il coefficiente di normalizzazione (1/52):

$$\frac{1}{256} * \frac{1}{52} = 75.12 * 10^{-6} \tag{7.3}$$

Optiamo quindi per l'utilizzo di 16 cifre alla destra della virgola e approssimiamo il minimo valore come $0.0000000000000100 = 2^{-14}$, che in decimale corrisponde a $6.103 * 10^{-5}$.
Ciò porta ad un errore assoluto pari a:

$$E = \left(7.512 * 10^{-5}\right) - \left(6.103 * 10^{-5}\right) = 1.409 * 10^{-5} \tag{7.4}$$

I valori in uscita devono essere ufix(32) con scalamento 2^{-16} per le componenti $\left\langle I_x^2 \right\rangle$ e $\left\langle I_y^2 \right\rangle$, mentre sfix(33) con scalamento 2^{-16} per il prodotto $\left\langle I_x I_y \right\rangle$.

7.6 Calcolo della *corneress*

Una volta ottenute le componenti per il calcolo dell'algoritmo ($\langle I_x^2 \rangle$, $\langle I_y^2 \rangle$ e $\langle I_x I_y \rangle$), è possibile, utilizzando lo schema di figura (7.7), simulare il comportamento dell'algoritmo di *Forstner*.

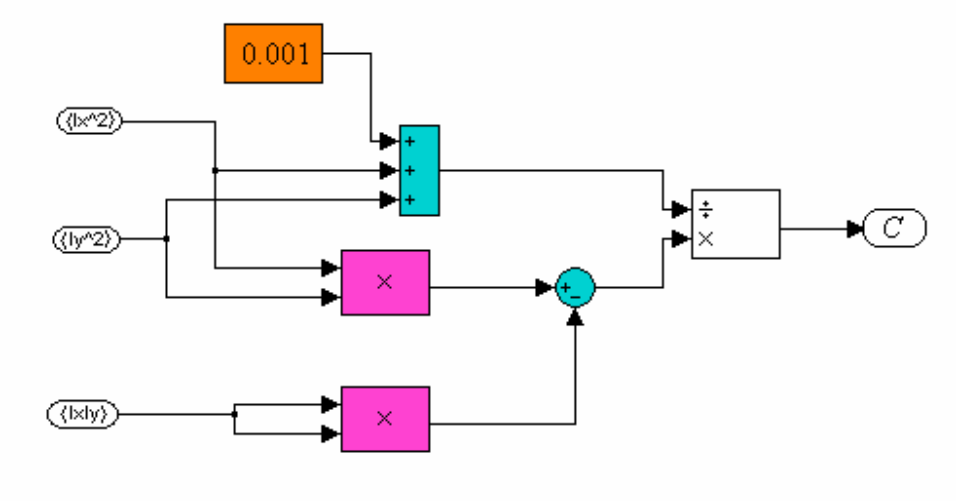


Fig. 7.7: schema di calcolo per l'algoritmo di Forstner.

Per i due blocchi di colore magenta il rigoroso calcolo della dimensione dell'uscita porterebbe a fissarla a *ufix*(64) con scalamento 2^{-32}; questo impone di utilizzare molti bit a destra della virgola che in hardware causerebbero un elevato dispendio di risorse, per cui scegliamo di troncare i valori a dieci bit.

Per il blocco sommatore a tre ingressi l'uscita deve essere posta a *ufix*(33) con scalamento 2^{-16}. Come per i blocchi moltiplicatori decidiamo di troncare a dieci bit.

Un addendo del sommatore è dato dalla costante inserita nel blocco color arancio (0.001), che serve ad evitare di avere in ingresso al divisore (blocco bianco) un dividendo nullo, cosa che

porterebbe l'uscita a valori infiniti. Il sottrattore (blocco tondo di color ciano) deve avere l'uscita *sfix*(43) con scalamento 2^{-10}.

L'ultimo blocco è il divisore, che avendo al divisore valori *ufix*(27) con scalamento 2^{-10} ed al dividendo valori *sfix*(43) con scalamento 2^{-10}, prevede in uscita valori *sfix*(63) con scalamento 2^{-27}.

Vediamo come si ottiene tale risultato, sia dato U come

$$U = \frac{\text{Numeratore}}{\text{Denominatore}}$$

quindi

$$\frac{Num_{\min}}{Den_{\max}} \leq U \leq \frac{Num_{\max}}{Den_{\min}}$$

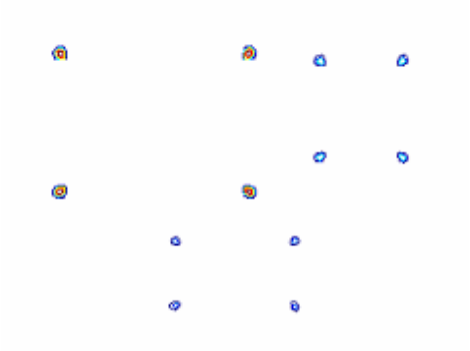
da cui

$$\frac{2^{-10}}{2^{17}} \leq U \leq \frac{2^{32}}{2^{-3}} \tag{7.5}$$

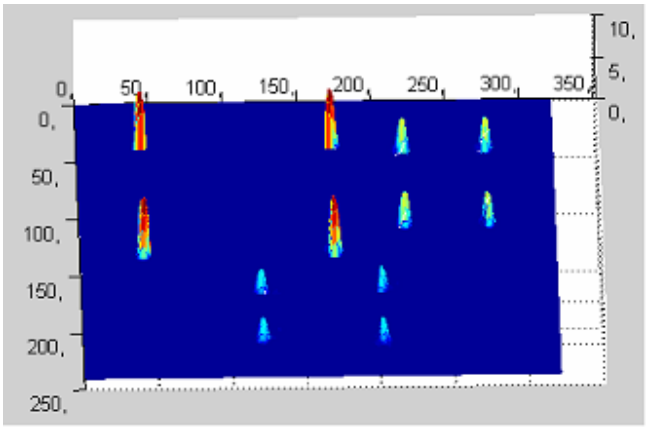
7.7 Simulazioni su immagini preparate ad-hoc

Lo schema è stato simulato impiegando l'immagine sintetica mostrata in figura (7.8).

(a)



(b)



(c)

Fig.7.8: (a) immagine di test; (b) localizzazione dei *corner*; (c) visualizzazione dell'ampiezza dei *corner* su scala logaritmica.

L'immagine riporta figure geometriche di colore diverso su sfondo bianco, questo è stato fatto per studiare il comportamento dell'algoritmo in presenza di livelli di contrasto di diversa ampiezza.
Da questo ne deriva che all'aumentare del contrasto, l'ampiezza degli impulsi risulta maggiore.

7.7.1 Risultati simulativi

Le prove sono state eseguite con l'immagine di figura (7.9) che è la stessa utilizzata per le simulazioni viste nel capitolo 6.

Fig. 7.9: immagine da elaborare.

Il risultato ottenuto è mostrato in figura (7.10):

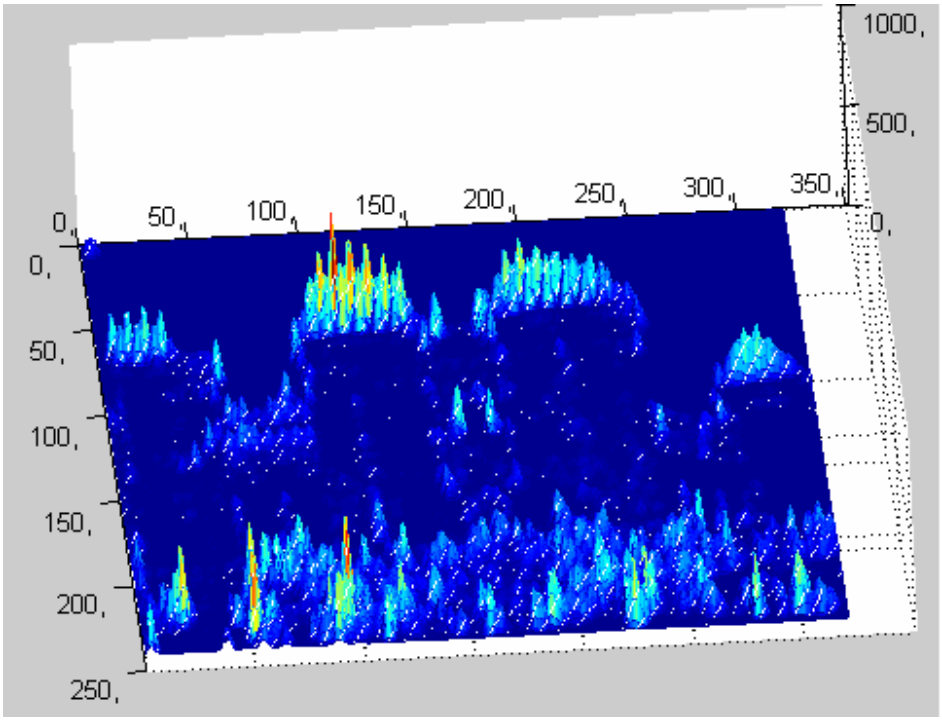


Fig. 7.10: risultato immagine elaborata.

Questo è stato poi confrontato con quello ottenuto nel capitolo 6 tramite dati di tipo *double*. Qui, avendo apportato troncamenti dei dati in più punti dello schema, è inevitabile attenderci delle differenze sul risultato finale.

Questo si nota dalla figura (7.11) dove le differenze più accentuate sono riscontrabili sui *corner*.

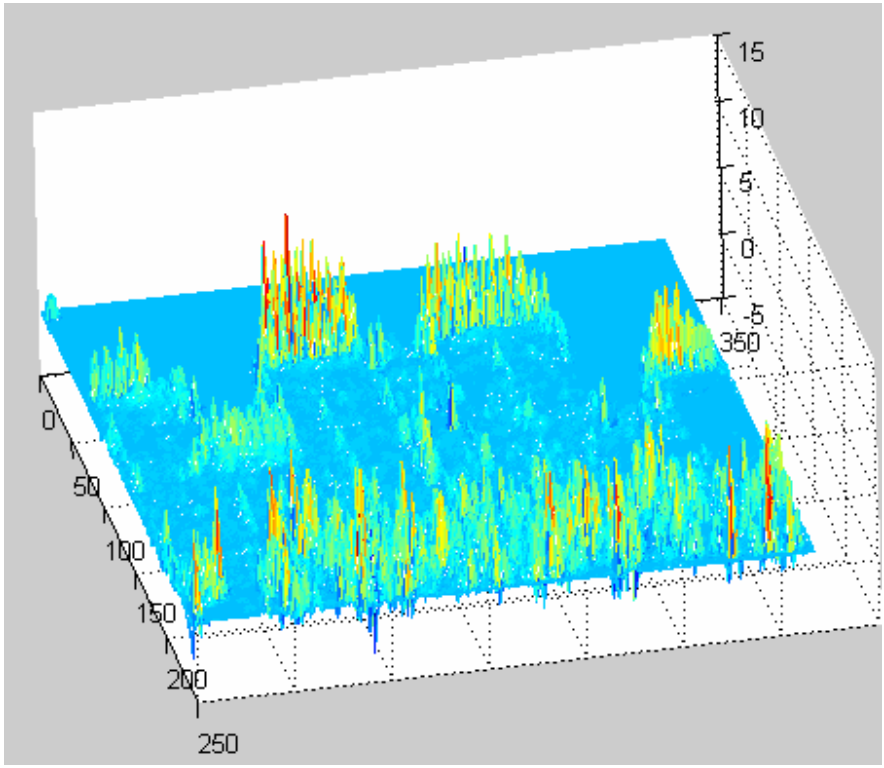


Fig.7.11: differenza tra algoritmo di tipo *double* e *fixed-point*.

7.8 Conclusioni

Nella realtà è difficile trovare immagini nettamente contrastate, avremo a che fare con variazioni di colore abbastanza graduali e quindi picchi di ampiezza limitata.
Vediamo, in figura (7.12), l'ampiezza dei picchi relativa all'immagine di figura (7.8):

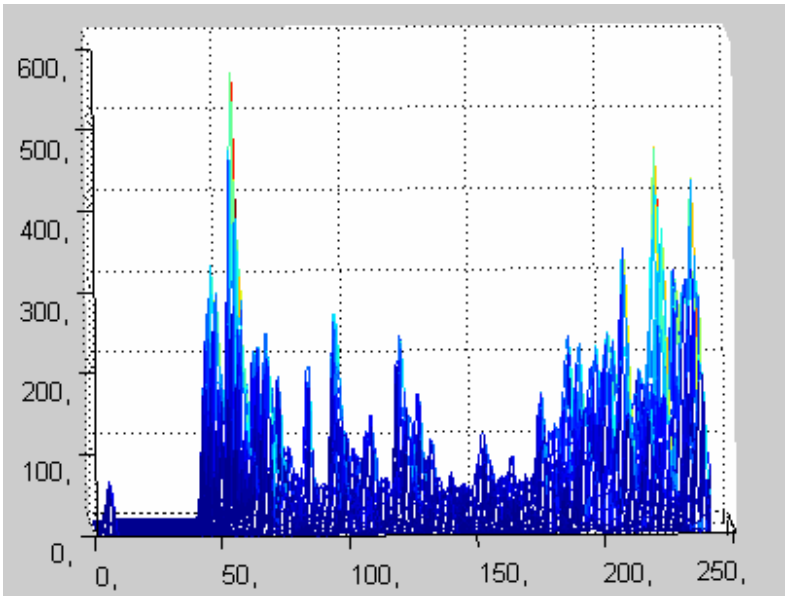


Fig.7.12: visualizzazione ampiezza dei *corner*.

Da questa si nota che i picchi sono di altezza limitata (valore massimo circa 600), sulla base di altre prove abbiamo constatato che il valore massimo dell'ampiezza è di circa 800.

Sulla base di queste considerazioni possiamo ridurre il numero di cifre binarie dei dati *fixed-point* riducendo la dimensione di *bus*, registri e memorie.

Capitolo 8

Simulazione con Xilinx System Generator

8.1 Introduzione

Xilinx System Generator è un estensione di *Simulink* che fornisce la possibilità di implementare progetti, simularne il funzionamento, generare il codice VHDL e stimare le risorse *hardware* impiegate sulle singole famiglie di FPGA *Xilinx*.
Utilizzando i blocchi di libreria, definendo i parametri del formato *fixed-point* (dimensione della parola, posizione della virgola binaria, arrotondamento e saturazione) e collegandoli tra loro come accade per i blocchi *Simulink standard* si ottiene lo schema desiderato.
Per ulteriori informazioni rimandiamo all'Appendice C.

8.2 Primo approccio simulativo

Lo schema studiato nel capitolo precedente è stato ora tradotto rispettando i vincoli riportati in Appendice C, utilizzando *Xilinx System Generator*.
La Figura (8.1) mostra il modello ottenuto:

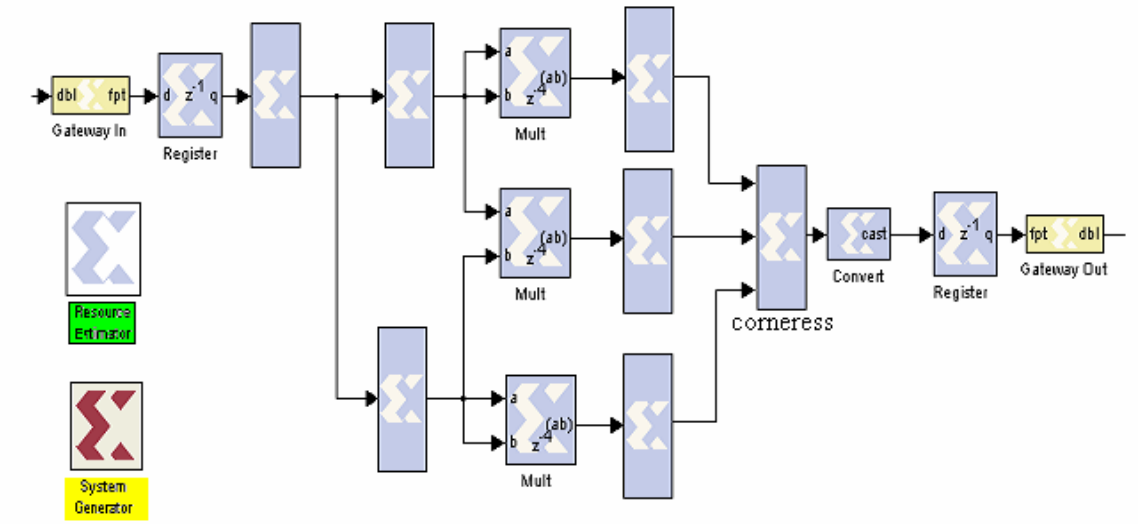


Fig. 8.1: schema realizzato con *System Generator*.

Nella figura (8.1) si noti che il blocco evidenziato in giallo deve essere necessariamente inserito, in quanto è l'interfaccia tra il *System Generator* e la generazione del codice HDL. Inoltre, rispetto alla struttura tradizionale in *Simulink*, possiamo introdurre il blocco evidenziato in verde, il quale permette di stimare le risorse impiegate per l'implementazione *hardware*. La figura (8.2) riporta il *function block* di un blocco moltiplicatore tramite il quale è possibile fissare la precisione dei dati *fixed-point*.

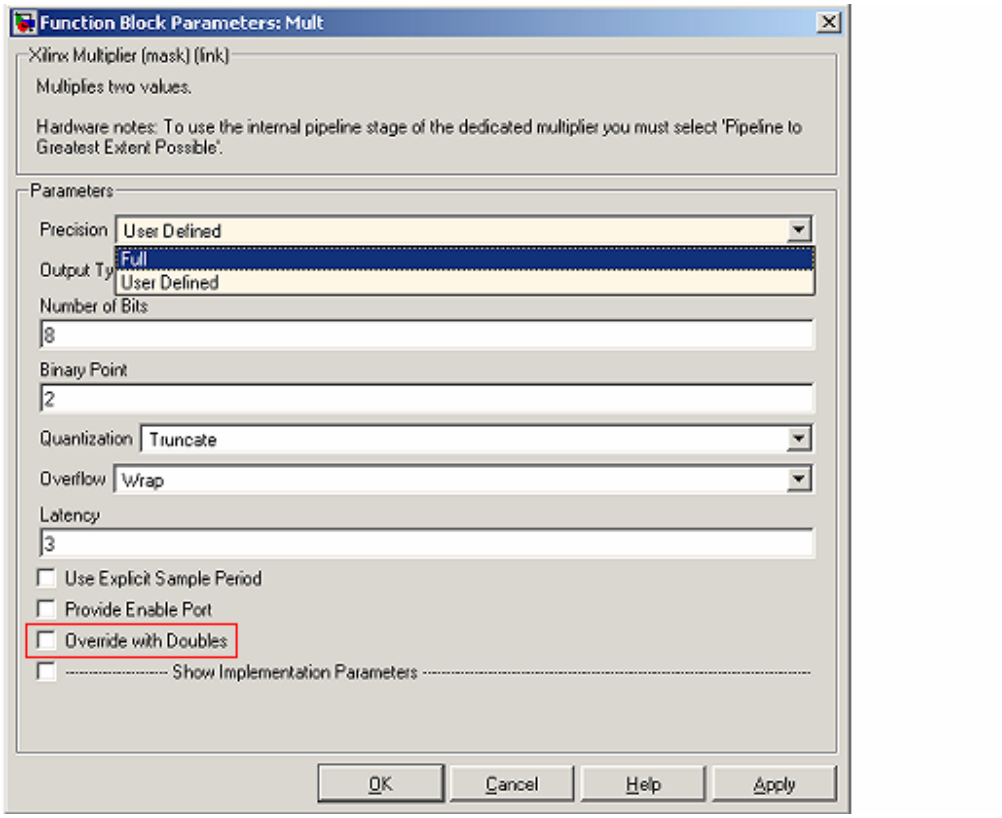


Fig. 8.2: *function block* del blocco moltiplicatore.

La *function block* esiste per ognuno dei blocchi della libreria di *System Generator*.

Il primo approccio è stato la simulazione dell'algoritmo con dati di tipo *double* (selezionare la voce nel rettangolo rosso di figura (8.2)); i risultati coincidono esattamente con quelli trovati

usando le librerie *standard Simulink* (quantunque il tempo di simulazione è risultato ovviamente molto superiore).

Il successivo passo è stato l'utilizzo dell'aritmetica *fixed-point* ed in particolare l'uso della precisione "*full"*. Selezionando tale opzione nei *function block* di ogni blocco, il *software* calcola automaticamente il numero di bit e la posizione della virgola in tutti i punti dello schema, in modo da poter rappresentare tutti i possibili valori richiesti. Si deve poi considerare un nuovo parametro che è la *latenza*, la quale indica il ritardo necessario ad un blocco per eseguire l'elaborazione richiesta. La *latenza* dipende dalla dimensione dei dati in ingresso, dalla complessità dell'elaborazione del blocco e dal minimo periodo di *clock* imposto.

Il parametro può essere settato dal programmatore nel campo *latency* di figura (8.2); se in fase di simulazione il programma non riesce a rispettare il vincolo, interrompe l'esecuzione ed apre una finestra di dialogo dove viene suggerito il minimo valore supportato. Una volta fissati i valori di latenza per ogni blocco, si può avviare la simulazione.

Al termine della simulazione abbiamo eseguito l'operazione "*Resource Estimator*" (Appendice C) per renderci conto delle risorse necessarie ad implementare l'algoritmo con precisione "*full*".

Come si ci poteva attendere, con la precisione scelta e il tipo di algoritmo implementato i dati raggiungono dimensioni tali da occupare grandi quantità di memoria e richiedere blocchi matematici (sommatori, moltiplicatori) di elevata complessità.

8.3 Prove su immagini ad-hoc per la definizione dei dati

Per ottimizzare il consumo di risorse abbiamo simulato i primi stadi dell'algoritmo su un'immagine con massimo contrasto possibile (bianco-nero) figura (8.3).

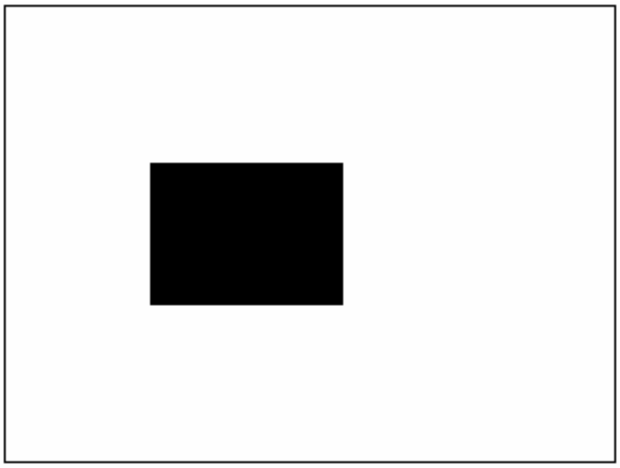


Fig 8.3: immagine a massimo contrasto possibile.

L'immagine di prova viene elaborata tramite il filtro di *smoothing* gaussiano, e successivamente viene calcolato il gradiente lungo l'asse x con la formula:

$$\frac{\partial f}{\partial x} = f(x+1) - f(x) \tag{8.1}$$

Il risultato ottenuto è riportato in figura (8.4):

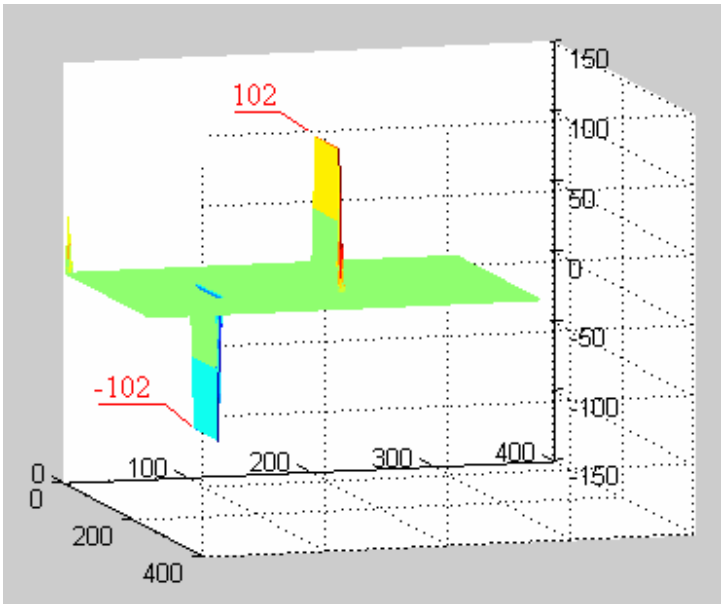


Fig. 8.4: valori limite del gradiente in uscita dal filtro di gradiente.

Come si può vedere otteniamo due fronti: quello negativo (ampiezza massima -102) indica il passaggio bianco-nero, mentre il positivo (valore massimo 102) il passaggio nero-bianco.
Il *range* di escursione da -102 a 102 con codifica "complemento a due" è rappresentabile attraverso otto bit. Fissiamo questo vincolo nei blocchi di calcolo delle componenti del gradiente figura (8.5).

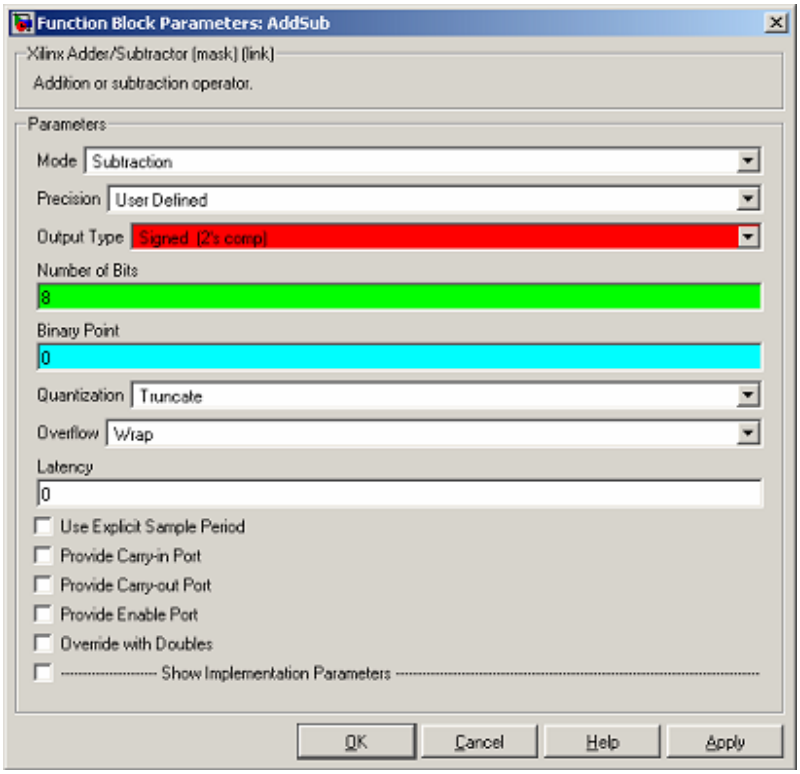


Fig. 8.5: configurazione *function block* del filtro del gradiente.

Nel campo evidenziato in rosso di figura (8.5) imponiamo l'uscita dei dati con codifica "complemento a due", nel campo verde la lunghezza della parola binaria, infine nel campo azzurro fissiamo di lavorare con numeri interi.

Operativamente abbiamo imposto tali condizioni nei blocchi iniziali; nei successivi, per ottimizzare i risultati, tramite i *dialog box* imponiamo la precisione "*full*".

Un'ulteriore minimizzazione delle risorse si ottiene sostituendo i filtri di *Sobel* per il calcolo delle componenti del gradiente con il calcolo approssimato nel discreto, in cui la derivata prima parziale nelle due direzioni diventa:

$$\begin{cases} \dfrac{\partial f}{\partial x} = f(x+1) - f(x) \\ \dfrac{\partial f}{\partial y} = f(y+1) - f(y) \end{cases} \tag{8.2}$$

In figura (8.6) mostriamo l'implementazione in *Simulink* delle formule (8.2):

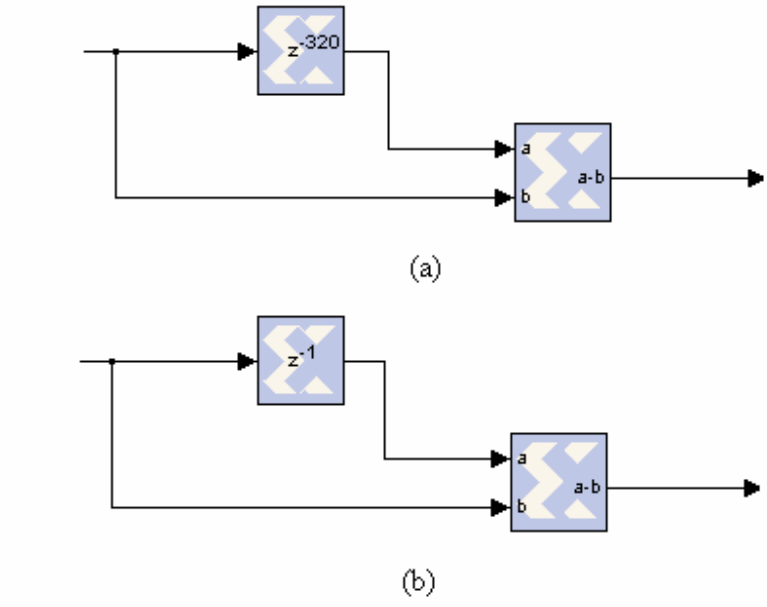


Fig.8.6: (a) approssimazione della derivata parziale lungo la direzione y,
(b) approssimazione della derivata parziale lungo la direzione x.

Così facendo eliminiamo i moltiplicatori nel calcolo del gradiente e riduciamo la dimensione delle linee di ritardo; con i filtri direzionali di *Sobel* si utilizzano due linee di ritardo di dimensione 320 per ogni filtro, per un totale di: (320*2)*2=1280. Utilizzando invece questa approssimazione si arriva a (320+1)=321.

8.4 Analisi del blocco di calcolo della *corneress*

La parte fondamentale dell'algoritmo è racchiusa nel blocco denominato *corneress* in figura (8.1). Vediamo in figura (8.7) il comportamento nel dettaglio:

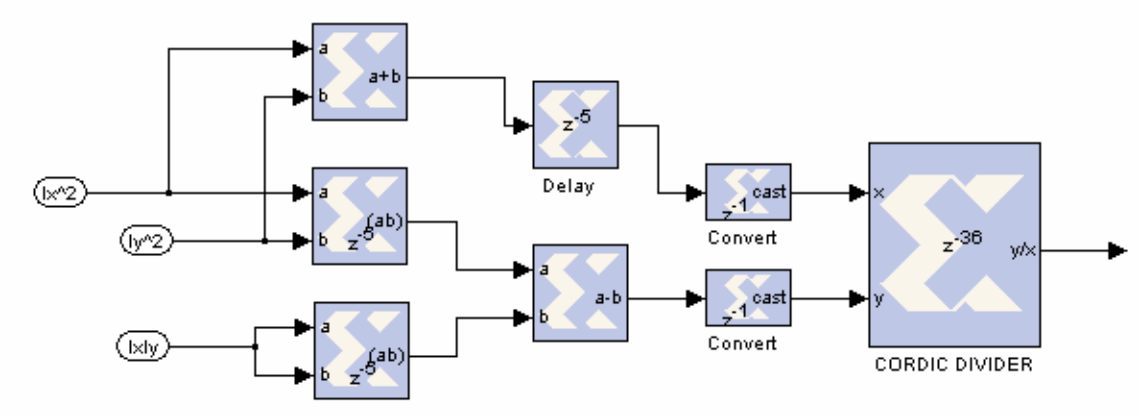


Fig.8.7: blocco che esegue il calcolo della *corneress*.

Il blocco "*delay*" è stato introdotto per compensare le latenze introdotte dai due moltiplicatori e mantenere quindi il parallelismo dei dati in ingresso al divisore. I due blocchi "*convert*" sono introdotti per assicurare che in ingresso al divisore i dati abbiano le stesse dimensioni, poiché i blocchi di somma e prodotto danno come risultato parole di lunghezze diverse.
Il "*cordic divider*" è un blocco di libreria *Xilinx Blockset* che esegue la divisione sfruttando la tecnica *COrdinate Rotation DIgital Computer*, la quale prevede un controllo all'interno per evitare la divisione per zero.

8.5 Latenza totale del circuito

La latenza è il tempo necessario al circuito per elaborare un dato.
Ogni stadio ha un valore di latenza dovuta ai blocchi di libreria che lo compongono e alla loro disposizione; vediamo il calcolo della latenza totale:

- Prefiltraggio di smoothinh 657
- Gradiente 0
- Prodotti 4
- Gaussiane 657
- Calcolo della *corneress* 42
- Registri di input/output 2

Latenza totale = 657+0+4+657+42+2= 1362

Le latenze parziali sono state misurate simulando singolarmente gli stadi di elaborazione ai quali sono stati inviati valori prestabiliti. Ad esempio, per il calcolo della latenza dei filtri gaussiani (media pesata) abbiamo inviato ripetutamente lo stesso valore fino ad averlo in uscita. Il valore si è presentato in uscita 657 colpi di *clock* dopo il primo invio.

8.6 Sogliatura dell'uscita

L'algoritmo di *corner detection* così implementato rileva tutti i *corner* presenti nell'immagine, dai più evidenti ai meno pronunciati. Per mantenere solo i *corner* di un certo interesse eseguiamo un'operazione di sogliatura in modo da eliminare eventuali disturbi dovuti a sfumature di colore, ombre ed altro. Ampliamo lo schema come in figura (8.8) per selezionare i picchi di *corneress* di ampiezza maggiore:

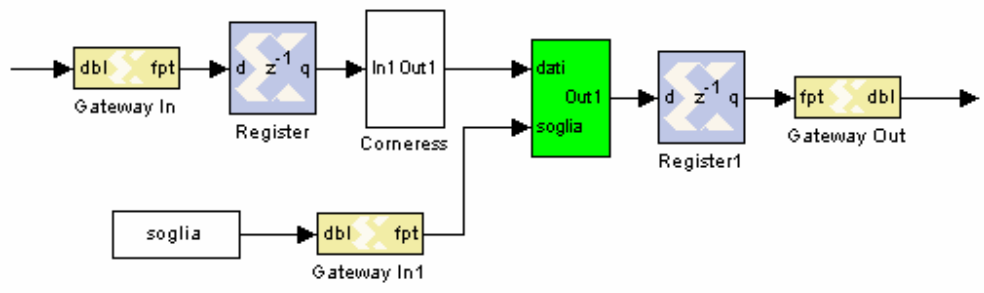


Fig. 8.8: schema completo con selezione dei *corner*.

Il blocco color verde di figura (8.8) esegue il confronto fra i valori della *corneress* e la soglia: se i dati risultano maggiori della soglia vengono riportati in uscita, altrimenti questa viene posta a zero. Analizziamo nel dettaglio il funzionamento del blocco illustrato in figura (8.9):

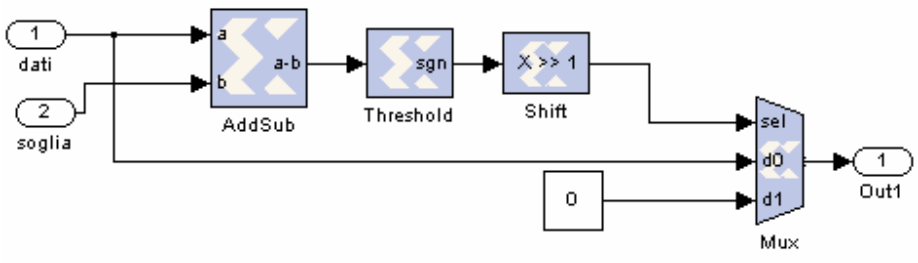


Fig. 8.9: schema del blocco di sogliatura.

In uscita dal blocco "*threshold*" abbiamo il valore 1 (01 in complemento a due) se il risultato della differenza è positivo, -1 (11 in complemento a due) se il risultato è negativo. Il successivo blocco "*shift*" è stato introdotto per poter pilotare il *multiplexer*: shiftando a sinistra di una posizione otteniamo 0 per l'uscita positiva e 1 per quella negativa. Sfruttando tali segnali l'uscita del *multiplexer* permette il passaggio del dato in ingresso se è abilitata la selezione 0, mentre forza l'uscita a zero se è attiva la selezione 1.

8.6.1 Risultati simulativi con soglia

Nelle figure (8.10) (8.11) sono visualizzate rispettivamente l'uscita con tutti i picchi evidenziati, e la stessa applicando un valore di soglia pari a 100.

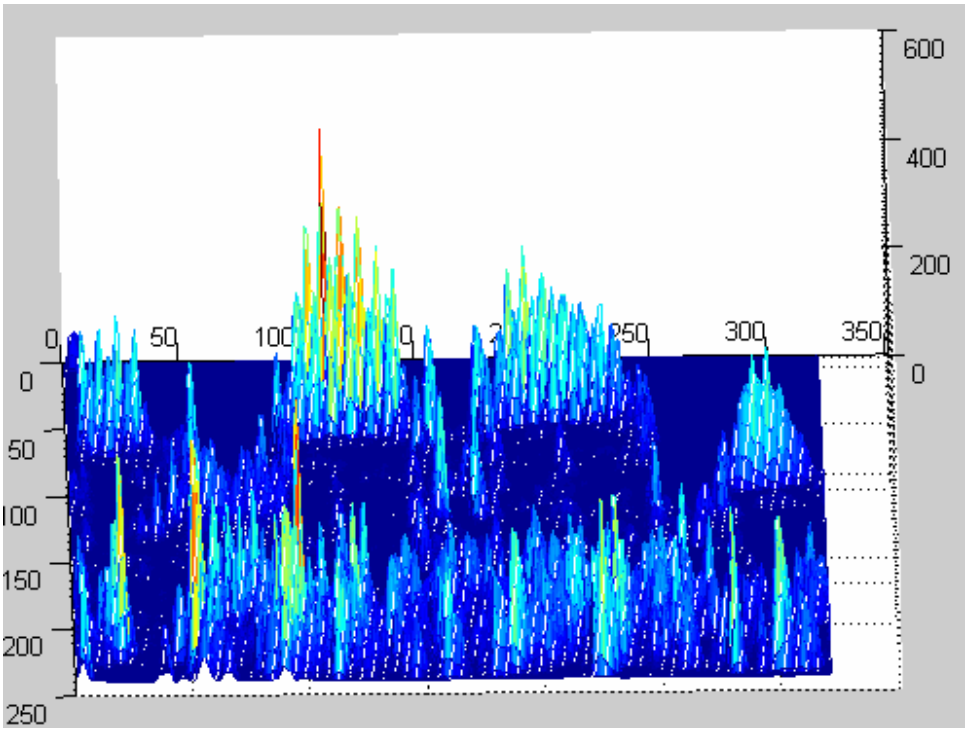


Fig 8.10: risultato dell'algoritmo di *corner detection*.

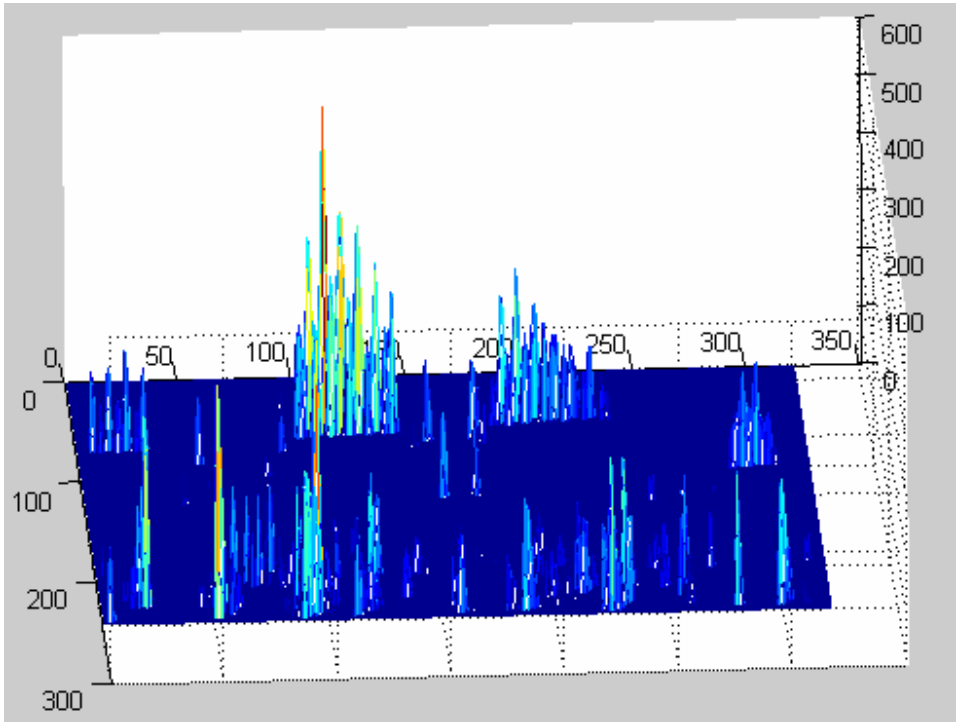


Fig 8.11: risultato dell'algoritmo con soglia 100.

Come è intuitivo osservare solo i valori superiori alla soglia vengono visualizzati mentre i restanti sono annullati.

La scelta del valore di soglia dipende fondamentalmente da due aspetti: per prima cosa dal tipo di immagini trattate, secondariamente dal grado di qualità dell'individuazione ricercata.

Dovendo trattare immagini fortemente contrastate e con forme geometriche ben definite, i valori dei picchi saranno più accentuati rispetto ad immagini "pittoriche" dove si hanno contrasti meno accentuati. Il secondo parametro permette di impostare una maggiore o minore accuratezza nell' individuazione dei *corners*: al diminuire del valore di soglia verranno localizzati i *corners* via via meno evidenti.

Capitolo 9

Interfaccia per videocamera OV7640

9.1 Interfaccia per formato YUV 4:2:2

Qualora fosse necessario lavorare su immagini in bianco e nero è necessario predisporre la videocamera per trasmettere nel formato "YUV 4:2:2" e filtrare la sola componente "Y".
In figura (9.1) è riportata la codifica di un *pixel*:

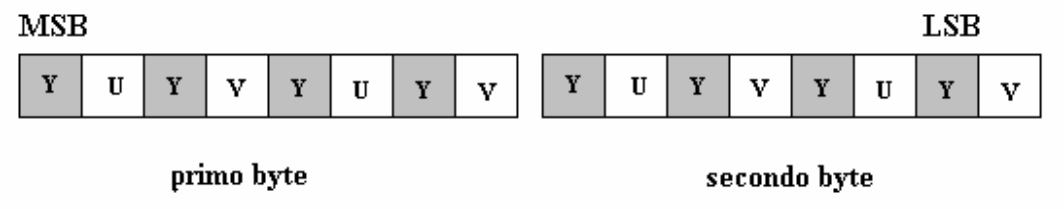


Fig. 9.1: formato YUV della videocamera OV7640.

Per poterla elaborare occorre estrarre la "Y" e per questo si usa il circuito di figura (9.2):

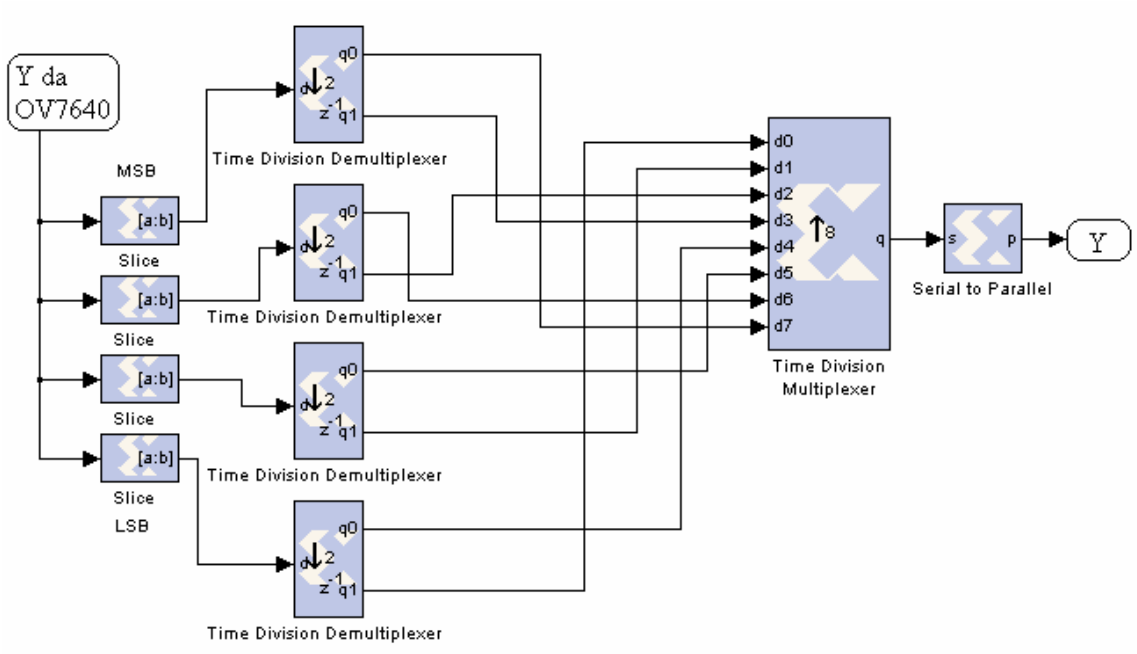


Fig. 9.2: circuito per la selezione della componente "Y".

Il primo stadio, composto da blocchi *"slice"*, ha il compito di selezionare i *bit b1, b3, b5, b7* dei due *byte* del *pix*el, il secondo separa le componenti "y0-y4", "y1-y5", "y2-y6", "y3-y7" che vengono poi ordinate dagli ultimi due blocchi per formare la parola

Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0

Al termine del filtraggio voluto è possibile tornare alla codifica di partenza attraverso il circuito di figura (9.3):

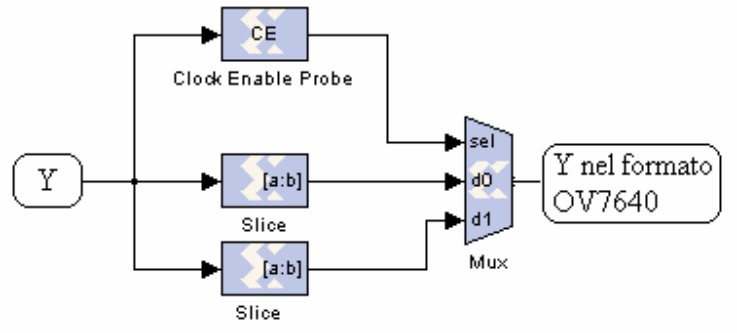


Fig. 9.3: circuito che riporta alla configurazione originale.

Quest'ultimo circuito non è necessario nel caso in cui si scelga di elaborare l'immagine con algoritmi complessi quale la *corner detection*, in quanto in uscita non si avrà l'immagine di partenza modificata, ma dati che descrivono alcune caratteristiche salienti dell'immagine.

9.2 Interfaccia per formato RGB 555

Per poter collegare la videocamera al chip di elaborazione è necessario avere lo stesso formato dei dati da entrambi i lati. Impostiamo la videocamera per lavorare con il formato di uscita RGB 555 mostrato in figura (9.4):

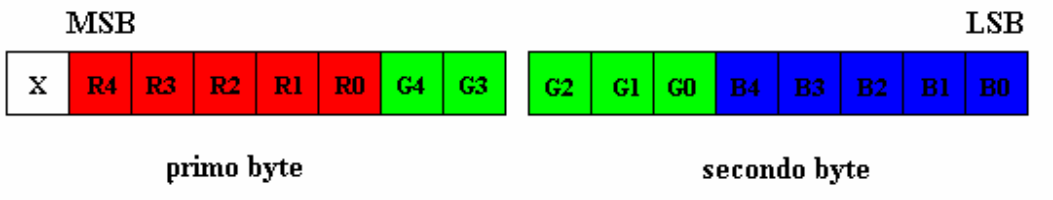


Fig.9.4: formato RGB 555 di OV7640.

Le componenti tricromatiche di un *pixel* sono disposte su due *byte* consecutivi; questo non crea problemi per R e B in quanto hanno tutti i *bit* sullo stesso *byte*, ma ne crea per G è frazionato in due parti: G4 e G3 sul primo *byte* trasmesso, i restanti sul secondo.
Questo è risultato un passo delicato da affrontare nel mantenere il sincronismo dei dati, in quanto si può iniziare l'elaborazione di G solo all'arrivo del secondo *byte*.
Il problema è stato risolto tramite lo schema di figura (9.5):

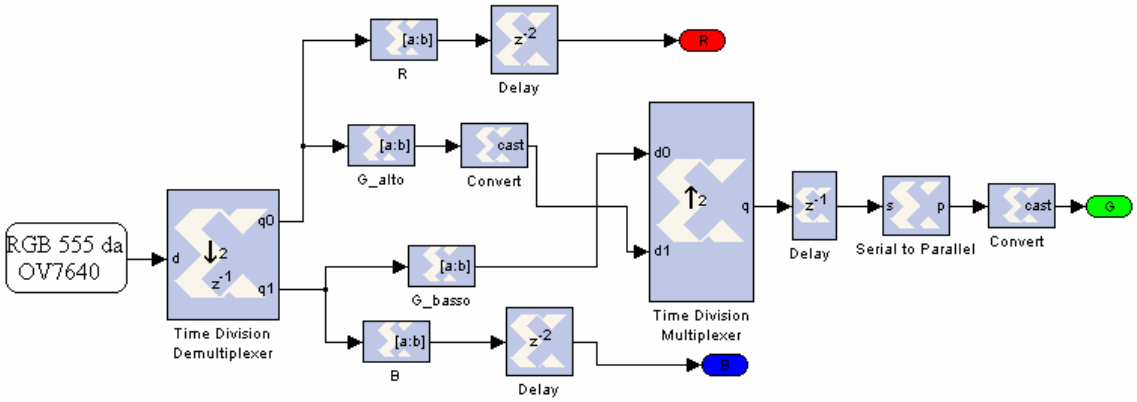


Fig.9.5: circuito di interfaccia fra OV7640 e filtro generico.

Il primo blocco *"Time Division Demultiplexer"* separa i due *byte* del *pixel*: su *"q0"* va il primo e su *"q1"* il secondo. Il blocco *"R"* seleziona i *bit b2…b6* (componente cromatica Red) del primo *byte*, *"B"* seleziona *b0…b4* (componente cromatica Blue) del secondo *byte*.
"G_alto" isola G4 e G3 (*b1, b0* del primo *byte*), successivamente *"convert"* crea la parola "0 G4 G3"; analogamente*"G_basso"* isola G2, G1 e G0 (*b7, b6,* b5 del secondo *byte*).
Il secondo blocco *"Time Division Demultiplexer"* crea la parola seriale

0	G4	G3	G2	G1	G0

poi si trasforma in parallelo e si elimina lo "0" in testa. I blocchi *"delay"* sono introdotti per mantenere il sincronismo.

In uscita da questo circuito si hanno le tre componenti cromatiche indipendenti, quindi si possono filtrare singolarmente utilizzando la stessa struttura di filtro in figura (9.6):

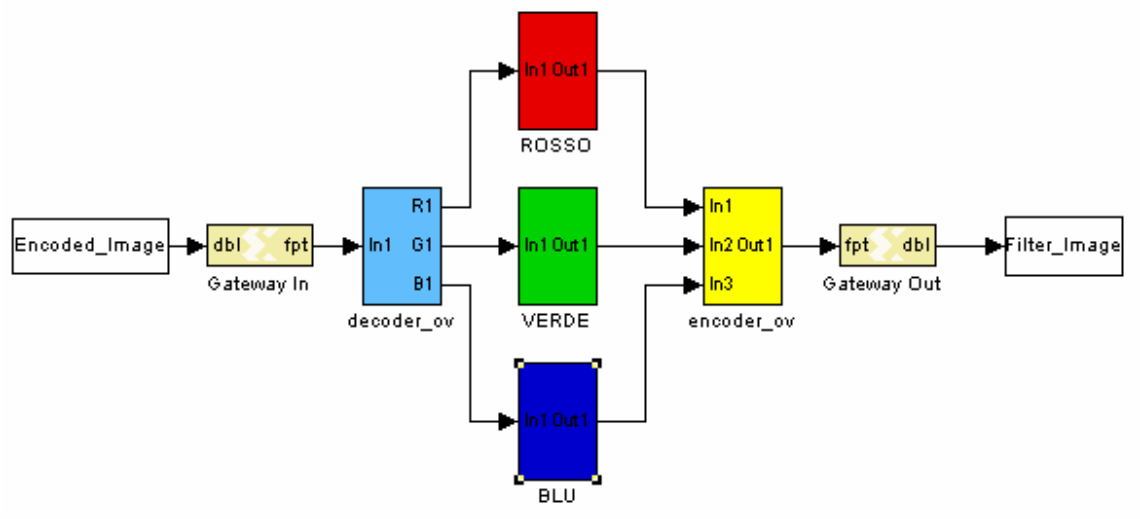


Fig 9.6: schema di filtro generico.

Eseguita l'operazione di filtraggio sulle tre componenti separate occorre riportare il formato come mostrato in figura (9.3). Utilizziamo per questo il circuito di figura (9.7):

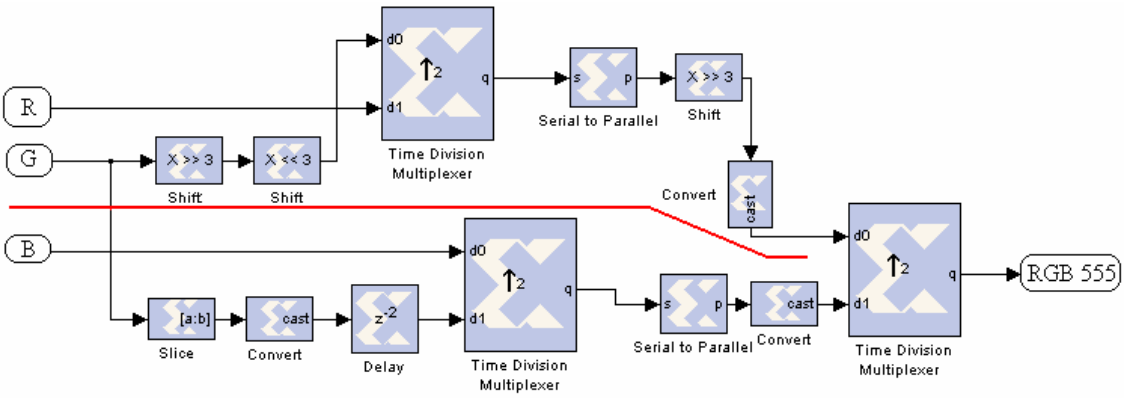


Fig.9.7: circuito che riporta alla codifica originale.

La parte di circuito al di sopra della linea rossa elabora il primo *byte*: i due *"shift"* sul canale *"G"* formano la parola

G4	G3	0	0	0

che entra nella porta *"d0"* del "*Time Division Demultiplexer"* ; quindi sulla porta *"d1"* abbiamo "R", ed in uscita si avrà

R4	R3	R2	R1	R0	G4	G3	0	0	0

I successivi blocchi eliminano i tre "0" in coda alla parola.

Il circuito al di sotto della linea rossa elabora il secondo *byte*: il blocco *"slice*" seleziona

G2	G1	G0

"convert" forma la parola:

0	0	G2	G1	G0

mentre *"delay"* è inserito per rispettare il sincronismo con la componente *"B"* del *pixel.* A questo punto il blocco "*Time Division Demultiplexer"* forma la parola:

0	0	G2	G1	G0	B4	B3	B2	B1	B0

ed i successivi eliminano gli *"0"* di testa. Il "*Time Division Demultiplexer"* finale permette l' alternarsi dei due *byte* che costituiscono i *pixel* elaborati.

Capitolo 10

Conclusioni e sviluppi futuri

Gli algoritmi di filtraggio digitale sono solitamente implementati su processori *DSP* oppure, quando i requisiti impongono elevate frequenze di campionamento, su *ASIC* appositamente progettati.

L'implementazione di questi filtri su FPGA permette di raggiungere prestazioni significativamente superiori a quelle basate su *DSP* tradizionali e, per volumi di produzione moderati, costi molto più bassi, essendo del tutto assenti i costi NRE (*non-recurring engineering costs*) che sono molto forti nel caso ASIC.

Vi sono poi dei vantaggi secondari dato che le FPGA possono essere riprogrammate: ciò rende possibile il miglioramento delle funzionalità offerte dal prodotto, l'adattamento a nuove condizioni operative o a nuovi standard approvati nel frattempo, allungando sostanzialmente la vita utile del prodotto. Questa cosa sarebbe impossibile per *DSP* e *ASIC*.

Una delle caratteristiche più vantaggiose delle FPGA è la possibilità di confrontare le prestazioni di diverse architetture senza alcun costo di produzione e senza tempi di attesa.

Una FPGA non possiede la velocità che può avere un circuito integrato *ASIC* di pari tecnologia, infatti paga la completa riconfigurabilità con la presenza di un numero elevato di risorse di interconnessione, ovvero alcune net che attraversano tutta l'FPGA con i relativi *driver*, ed altre risorse di interconnessione locali. Poichè ogni net deve fisicamente raggiungere un numero elevato di blocchi, il suo *fanout* e le sue capacità parassite introducono nelle FPGA ritardi superiori rispetto a quelle di un *chip ASIC*. Questo però non toglie al costruttore la possibilità di realizzare particolari linee interne dedicate con *driver* rinforzati e capacità parassite più basse per assolvere ad alcune funzioni, come la distribuzione del *clock* o, appunto, il calcolo e la propagazione del *carry* nelle addizioni.

I *DSP*, nonostante la loro struttura adatta all'elaborazione dei segnali, rimangono comunque macchine di tipo sequenziale in cui le unità aritmetiche sono riutilizzate in modo ciclico per eseguire operazioni complesse.

Nelle FPGA è possibile rendere più rapidi i tempi di elaborazione aumentando l'occupazione di risorse *hardware*. Sebbene in linea di principio questo sia possibile in misura arbitraria, non lo è nella pratica, poichè le risorse sono limitate. Anche nelle FPGA si deve poter trovare un'architettura che permetta un compromesso tra tempo di elaborazione e occupazione di area.

Il trattamento digitale dei segnali su FPGA è diventata una soluzione di riferimento anche in ambito video; il consolidamento di questo tipo di soluzione è così elevato che nel 2002 *Xilinx* ha reso disponibile un prodotto *software* denominato *System Generator for DSP*, che permette di completare tutte le fasi di progettazione, simulazione, implementazione e verifica di un sistema *DSP* basato su FPGA all'interno di un unico ambiente di lavoro di tipo *MATLAB/Simulink*.

Il progetto della tesi è stato concepito per rispondere pienamente alle specifiche di spazio e funzionalità derivanti dal suo futuro utilizzo (applicazioni robotiche).

Il nostro studio ci ha portato ad un'analisi di alcuni algoritmi di elaborazione di immagini con l'intento di integrare il circuito di elaborazione all'interno di un occhio robotico (ad esempio MAC-EYE).

Partendo da un algoritmo *software* di *corner detection* sviluppato all'interno del laboratorio in una precedente tesi, ci siamo proposti di analizzare la possibilità di integrare il tutto in *hardware*. Le necessità di ottenere spazi ridotti, pesi contenuti, complessità di calcolo elevate ci hanno indirizzato ad utilizzare un unico *chip* basato su dispositivi a logica programmata ed in particolare su FPGA.

Lo studio e l'analisi è stato realizzato per operare su dispositivi commerciali prodotti da *Xilinx* tramite appositi programmi *software*.

Alla fine dello studio si è deciso di utilizzare un dispositivo della famiglia Spartan3, ed in particolare "XC3S400", generando il codice HDL per la programmazione dell'*hardware* stesso.

I prossimi sviluppi porteranno alla programmazione del dispositivo, al suo test di funzionamento ed all'integrazione con la telecamera OV7640.
L'ultima fase comprenderà l'integrazione *embedded* nell'occhio robotico e lo studio del controllo tramite i segnali elaborati.

Appendice A

I formati colore

Possiamo suddividere i sistemi colore in due classi, seguendo la distinzione di *Helmholtz* (1852): additivi e sottrattivi. I primi utilizzano i colori primari della luce, definiti come rosso, verde e blu, per produrre ogni altro colore tramite addizione figura (A.1); la somma delle tre componenti monocromatiche dà il bianco, l'assenza delle tre il nero. I secondi vengono realizzati tramite l'utilizzo di pigmenti di varia natura, stesi o incorporati in opportuni supporti, che assorbono dalla luce che li irradia tutte le frequenze dello spazio tranne quelle della propria tinta; così, utilizzando giallo, magenta e ciano, è possibile ottenere tutte le altre tinte per sintesi sottrattiva figura(A.1): la sovrapposizione dei tre primari dà un grigio più o meno scuro in funzione della saturazione dei tre elementi, l'assenza dei tre dà il bianco. Tipici sistemi additivi sono le televisioni a colori, sottrattive le tecniche pittoriche, fotografiche e stampa a colori.

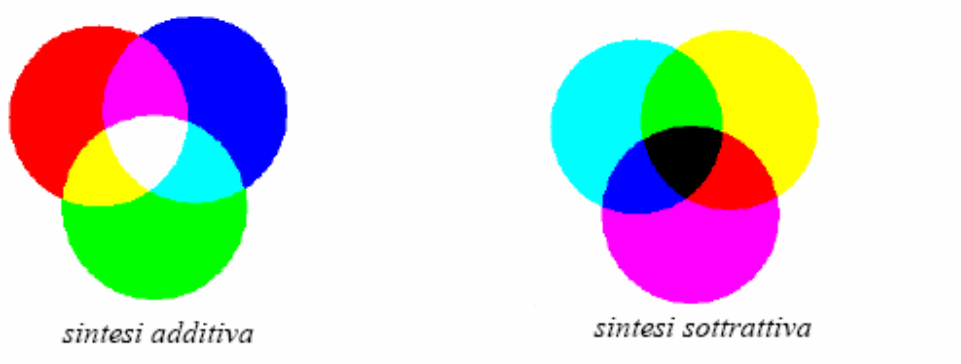


Fig. A.1: sintesi additiva e sottrattiva.

La gamma dei colori nel continuo è dunque funzione dell'assorbimento di certe lunghezze d'onda, più specificatamente di quelle comprese nella gamma del visibile (tra 360 nm e nello spettro della radiazione elettromagnetica); ogni oggetto illuminato assorbe certe radiazioni: le bande riflesse determinano il "colore" assunto dall'oggetto stesso.

L'ingegneria ha elaborato vari modelli digitali adatti alla rappresentazione e alla trasmissione elettronica del colore: per esempio in base alla tinta, alla luminosità o alle componenti primarie di colore. Si parla dunque di spazi di colore e in particolare è interessante, per l'elaborazione delle immagini, studiare quali variabili dello spazio colore siano trascurabili, quando possibile, al fine di analizzare le proprietà delle immagini, per diminuire i tempi computazionali (operare in uno spazio mono o bidimensionale è certamente più semplice rispetto ad un tridimensionale).

Gli spazi colore più importanti sono: *RGB* , *CMY*, *HSV*, *YUV e XYZ.*

Secondo il formato RGB ogni colore è ricavato tramite una sommatoria opportuna dei tre colori primari della luce: rosso, verde e blu; questa rappresentazione si è consolidata nel tempo con la diffusione degli schermi televisivi, le cui immagini sono appunto prodotte dalla combinazione di fosfori rossi, verdi e blu. La struttura nervosa dell'occhio umano, inoltre, è tale da prediligere il riconoscimento di tre particolari lunghezze d'onda, che sono proprio il rosso, il verde e il blu, e ogni altra tinta è percepita attraverso l'interazione delle tre sensazioni elementari.

Dunque lo spazio colore RGB è creato mappando i colori fondamentali in un sistema di coordinate cartesiane tridimensionale; il risultato, normalizzato tra 0 e 1, è un cubo tridimensionale come in figura (A.2).

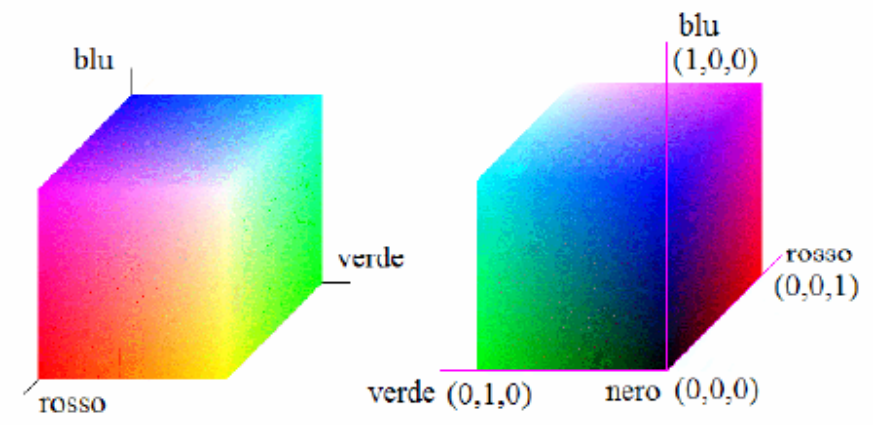


Fig.A.2: Lo spazio colore RGB. Si nota che il nero è nell'origine (0,0,0), dove le componenti sono nulle; il bianco di conseguenza è in (1,1,1), dove le tre componenti si sommano; le tre componenti "pure" si trovano negli spigoli, dove è massimo uno dei tre valori negli assi coordinati e nulli gli altri due.

Notiamo che la tricromatica del formato è additiva: ossia la somma delle tre componenti primarie della luce dà il bianco (come nelle frequenze luminose), e non il nero (come nei colori a tempera o a olio).

Il formato RGB riesce a coprire una vasta gamma di colori, seppur finita; il numero di colori è proporzionale al numero di bit associati per ogni pixel: a 8 bit per pixel avremo $2^8 = 256$ colori, a 16 bit ne avremo $2^{16} = 65536$; aumentando arriviamo poi alle moderne bitmap a 24 bit, con milioni di colori ($2^{24} = 16777216$), le cui minime differenze sono quasi impercettibili.

Tuttavia, sebbene un colore sia completamente specificato dalle quantità di rosso, verde e blu che lo compongono, esistono dei colori che il formato RGB non è in grado di rappresentare: il diagramma di cromaticità della *CIE* (*Commission Internationale de l'Elcairage*) in figura (A.3) mostra chiaramente i limiti della gamma offerta da questo formato. Per questo motivo è stato introdotto il formato XYZ, che seppur di scarso utilizzo permette di rappresentare tutta la gamma del visibile: la scelta ha dato origine allo standard CIE.

Ricordiamo la formula di conversione tra formato RGB e XYZ:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{bmatrix} 0.490 & 0.310 & 0.200 \\ 0.177 & 0.813 & 0.011 \\ 0.000 & 0.010 & 0.990 \end{bmatrix} \begin{pmatrix} r \\ g \\ b \end{pmatrix}$$

A titolo di informazione per i sistemi sottrattivi ricordiamo il formato CMY, che si basa sul mescolamento dei coloranti primari, ciano, magenta e giallo, ed è il modello utilizzato per le stampe e fotocopie a colori. La formula di conversione da RGB a CMY è molto semplice:

$$\begin{pmatrix} c \\ m \\ y \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} - \begin{pmatrix} r \\ g \\ b \end{pmatrix}$$

Una variante utilizza esplicitamente il nero, indicato con K, come quarto colore: tale modello, il CMYK, è impiegato per i processi di stampa a quattro colori, poiché il nero prodotto dalla sovrapposizione dei tre primari non sempre dà un buon risultato. Il difetto principale di questi sistemi di rappresentazione è la mancanza di uniformità percettiva; dati due colori, considerata la distanza ΔC., tale che:

$$C_3 = C_1 + \Delta C$$
$$C_4 = C_2 + \Delta C$$

se ΔC è costante sarebbe desiderabile che i colori C3 e C4 fossero percepiti come "ugualmente distanti" da C1 e C2 ; purtroppo questo non avviene, e le distanze in generale vengono percepite come differenti. In altre parole le variazioni di colore non sono lineari con la distanza, e due colori "distanti" possono sembrare in certi casi più simili di due colori più "vicini" figura (A.3).

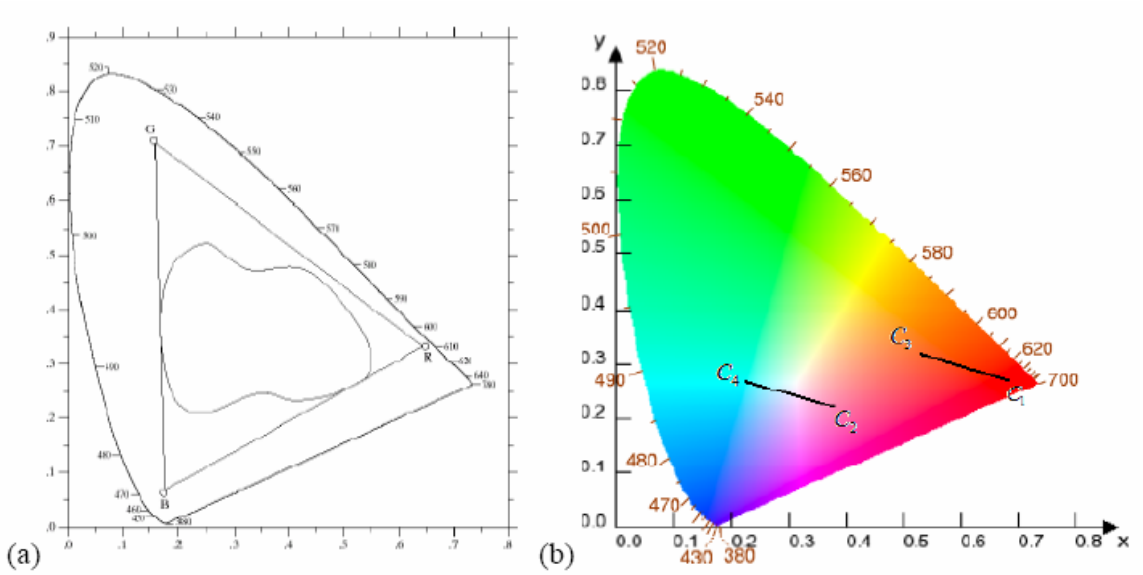


Fig.A.3: diagramma CIE di cromaticità: (a) mostra il range di colori, detto "gamma" prodotto dai monitor RGB (la forma triangolare), e quello dei dispositivi di stampa (la forma irregolare); (b) mostra la mancanza di uniformità percettiva.

Il formato HSV si basa sulla convenzione che un colore possa essere espresso dalla combinazione di tonalità(*hue*), saturazione(*saturation*) e intensità(*value*). La tinta o tonalità è ciò che normalmente chiamiamo "colore", ossia la lunghezza d'onda dominante; la saturazione è una misura di "purezza", poiché indica la quantità di nero, bianco o grigio che è mischiata col colore: ad esempio una saturazione nulla indica assenza di tinta, cioè una scala di grigio; infine la terza componente, l'intensità, è una misura della luminosità del colore. Un'utile rappresentazione dello spazio colore HSV è una piramide la cui base è un esagono, come in figura (A.4).

.

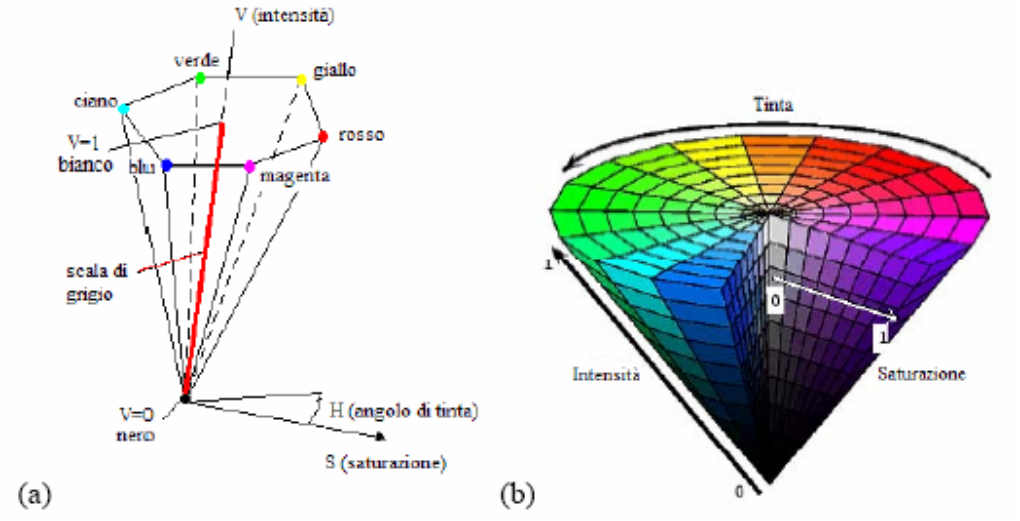


Fig.A.4: (a) e (b) lo spazio colore HSV: la tonalità è l'angolo di escursione tra le tinte, la saturazione è la distanza di un colore (punto) dall'asse piramidale, l'intensità è la distanza dalla punta della piramide . (a) In rosso è segnato l'asse, in corrispondenza della saturazione nulla, che indica la scala di grigio.

E' importante ricordare che non esiste una formula, ma un algoritmo di conversione tra il formato RGB e l'HSV [16]:

```
% algoritmo di conversione RGB-HSV
% R, G, B definite [0, 1]
% H definito [0, 360)
% S, V definite [0, 1]

MAX=max(R, G, B);
MIN=min(R, G, B);

V=MAX;          valore di V

if (MAX<>0)
    S=(MAX-MIN)/MAX;
    else S=0;          } valore di S

if  (S=0)
    H="indefinito";
    else
    delta=MAX-MIN;

    if (R=MAX)
        H=(G-B)/delta;
    else if (G=MAX)
            H=2+(B-R)/delta;
    else if (B=MAX)                    } valore di H
            H=60*[4+(R-G)/delta];
            if (H<0)
                H=H+360;
```

Il formato YUV è basato su un flusso di luminanza (Y), che è la componente di luminosità di ogni colore, e infatti costituisce l'unica sorgente per il formato in scala di grigi; corretto da U e V, che sono due valori di crominanza, funzioni delle componenti tricromatiche del colore, secondo la formula:

$$\begin{pmatrix} y \\ u \\ v \end{pmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.147 & -0.289 & 0.437 \\ 0.498 & -0.417 & -0.081 \end{bmatrix} \begin{pmatrix} r \\ g \\ b \end{pmatrix}$$

E' il formato adottato dal segnale televisivo (dal PAL , e nella variante YDbDr dal SECAM), dalle videocamere digitali, e si presta a originare varie forme "compresse" (YUV422,411ecc.) che sono la base di alcuni noti formati video, come l'MPEG, e immagine, come il JPEG.

Appendice B

Rappresentazione Fixed-Point su Matlab

I calcolatori *general-purpose* utilizzano tecniche di calcolo in virgola mobile per poter definire numeri variabili in un campo di valori molto grande senza dover utilizzare un elevato numero di bit. Un'altra rappresentazione è quella in virgola fissa, nella quale i numeri sono caratterizzati dalla dimensione della parola di bit, dalla posizione della virgola e dalla presenza del segno.

Questo secondo approccio permette l'utilizzo di hardware meno complesso rispetto al *floating-point*; implica che il *chip fixed-point* ha dimensioni più piccole con circuiti che richiedono minor potenza che sono caratteristiche fondamentali quando si opera su applicazioni mobili alimentate a batterie.

Lo sviluppo di *firmware* in virgola fissa può essere un processo delicato.

Per sviluppare il nostro sistema abbiamo utilizzato il *Toolbox Fixed-Point Blockset* (FPB) che include blocchi che estendono le librerie standard di *Simulink* per il calcolo in virgola fissa.

B.1 Generalità

Questa libreria permette di creare sistemi a tempo discreto che usano aritmetica *fixed-point*, inoltre tramite *Simulink* si possono simulare gli effetti comunemente incontrati utilizzando questa aritmetica.

Vediamo quali sono le caratteristiche di rappresentazione di questi numeri.

Una comune rappresentazione di un numero binario è visualizzata in figura (B.1):

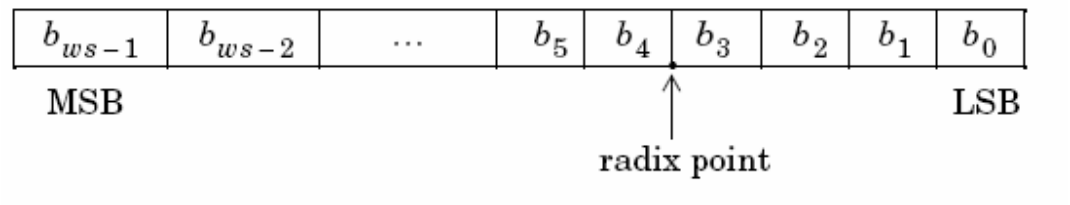


Fig.B.1: rappresentazione numero binario.

dove:

- b_i sono i bit;
- ws è la dimensione della parola in bit;
- *radix-point* indica la posizione della virgola rispetto a LSB (è responsabile dello scalamento).

La posizione della virgola determina di quanto va scalato il valore *fixed-point*, possiamo scegliere tra tre opzioni: *integers, fractionals, generalized fixed-point numbers*. La differenza sostanziale è legata alla posizione della virgola.
Valutiamo in dettaglio i tre tipi di dati supportati dal FPB:

1. *Integers*: *radix-point* è alla destra di LSB quindi lavoriamo con numeri interi, si identificano con **uint** numeri senza segno e **sint** con segno.
2. *Fractionals*: con unsigned il radix-point è alla sinistra di MSB, con *signed* è alla destra di MSB. Si identificano con **ufrac** e **sfrac** .
3. *Generalized fixed-point numbers*: in questo tipo di dati non vi è una posizione di *default* per *radix-point*. Sono rappresentati da **ufix** e **sfix**.

Una caratteristica di tale rappresentazione consiste nell'avere un *range* limitato di possibili valori , dovuto alla lunghezza finita della parola.

Per evitare condizioni di *overflow* (numeri che cadono al di fuori del *range*) e minimizzare l'errore di quantizzazione (dovuto alla conversione in binario) i numeri *fixed-point* devono essere scalati. Si può scegliere tra uno scalamento di *default* (che è fisso e non si può cambiare) oppure sceglierlo arbitrariamente attraverso il tipo 3 (*Generalized fixed-point numbers)*.

Un numero *fixed-point* può essere rappresentato attraverso uno schema di codifica generale [*slope bias*] :

$$V \approx V^{\circ} = SQ + B$$

dove V valore reale*;* V° valore reale approssimato; $S = F * 2^{E}$ *è lo slope (* F normalizzata t.c. $1 \leq F < 2$ *) ; B=polarizzazione;* Q= intero che rappresenta V.

Esistono due metodi di scalamento: *radix point-only* (o potenza di 2) il cui vantaggio è quello di minimizzare il numero di operazioni aritmetiche. In questo caso la formula generale assume i seguenti valori:

$$F=1,\ S=2^{E},\ B=0$$

da cui segue

$$V^{\circ} = 2^{E} * Q$$

cioè lo scalamento della quantizzazione di un numero reale è definito solo da una potenza di due.

L'altro metodo permette di definire sia lo *slope* che il *bias* attraverso la sintassi [*slope bias*].

Un'altra caratteristica importante riguarda il *range* e la precisione: il *range* di un numero evidenzia il limite di rappresentazione mentre la precisione mostra la distanza tra numeri successivi (è lo *slope* che determina tale valore). Il *range* e la precisione in un numero *fixed-point* dipendono dalla lunghezza della parola e dallo scalamento.

Lavorando con operazioni aritmetiche che coinvolgono numeri *fixed-point* occorre prestare attenzione che non si verifichi *overflow*, cioè che il risultato di una operazione cada al di

fuori del *range* di rappresentabilità. Un ulteriore problema è dato dalla quantizzazione in quanto il valore di un *fixed-point* è arrotondato e può non corrispondere al valore vero.
Infatti il risultato di una operazione in *fixed-point* tipicamente è caricato in un registro che ha una lunghezza pari al valore del dato originale. Se il risultato ha un numero di bit che eccede la dimensione del registro è necessario effettuare una operazione di arrotondamento o troncamento. Questo è molto importante in quanto tali operazioni producono errori di quantizzazione e rumori computazionali.
Abbiamo quattro modi per attuare l'arrotondamento, noi utilizziamo *round toward nearest* che consiste nell'approssimare il numero calcolato al più vicino valore rappresentabile.

B.2 Tool Fixed-Point Blockset

Il *tool Fixed-point blockset* consiste di blocchi base *fixed-point Simulink*, i quali sono utilizzati per progettare e simulare sistemi dinamici usando aritmetica in *fixed-point*.
Le caratteristiche salienti sono:

- usare aritmetica *fixed-point* per lo sviluppo e la simulazione di modelli *Simulink* in *fixed-point*;
- cambiare il tipo di dato permettendo di osservare immediatamente la differente dimensione della parola, la posizione della virgola, etc.;
- generazione di un modello *fixed-point* attraverso un processore *floating-point*, per permettere di emulare gli effetti dell'aritmetica *fixed-point* in un sistema *floating*.

Lo scopo di questo *blockset* è colmare il *gap* che esiste tra il progetto di un sistema dinamico e la sua implementazione in *hardware fixed-point*.
Coloro che hanno sperimentato l'implementazione di un sistema su *hardware fixed-point* ben conoscono la difficoltà e il tempo necessario che questo processo richiede.

La maggior parte dei blocchi fornisce un *dialog box* (doppio *click*), i cui elementi presenti all'interno sono elencati sotto:

- il nome e il tipo di blocco appare in alto sul *dialog box*;
- sotto il titolo una breve descrizione sulla funzione svolta dal blocco;
- sotto la descrizione una serie di campi, uno per ogni parametro che può essere settato per il blocco considerato, in cui è possibile inserire anche espressioni *Matlab*;
- nella parte inferiore sono presenti tre bottoni etichettati ***Apply***, ***Cancel*** e ***Help***. Premendo ***Apply*** si chiude il *dialog box* settato ai valori correnti del campo; premendo ***Cancel*** si ritorna ai valori di *default* perdendo qualsiasi cambiamento fatto nel *dialog box*; premendo ***Help*** si visualizza una spiegazione dettagliata del comportamento del blocco.

Simulink legge i valori inseriti in questi campi e li passa a *Matlab* che li utilizzerà durante la simulazione.

In figura (B.2) mostriamo come si presenta un *dialog box*:

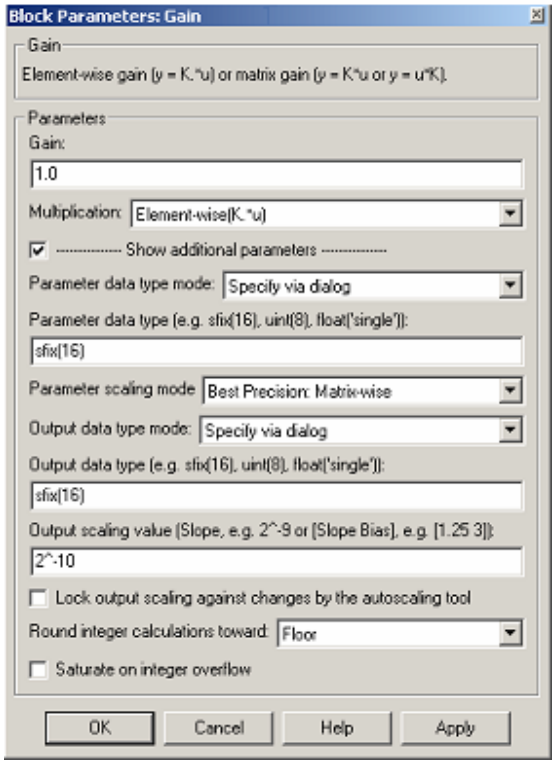


Fig.B.2: *dialog box* del blocco *fixed-point gain*.

Infine tutti i blocchi messi a disposizione dalla libreria sono mostrati in figura (B.3):

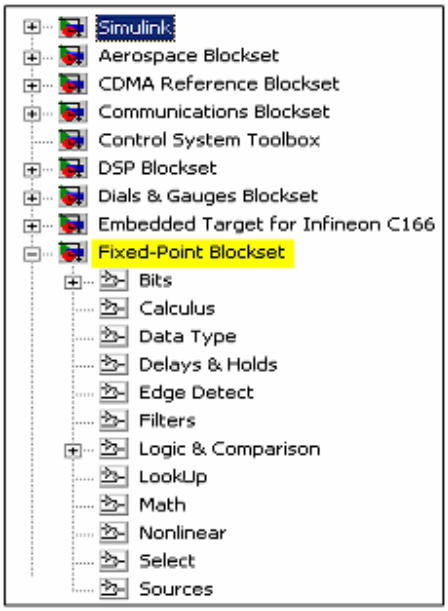


Fig.B.3: elenco blocchi *fixed-point*.

B.3 Rappresentazione numeri con segno

Spesso negli algoritmi numerici è necessario trattare numeri con segno. Tipicamente la rappresentazione di numeri negativi fixed-point è effettuata in tre differenti modi:

- modulo e segno,
- complemento a uno,
- complemento a due.

Ciascuna rappresentazione ha caratteristiche diverse e la scelta di utilizzo dipende dal tipo di operazioni aritmetiche richieste.

Il range di rappresentabilità di un numero senza segno e con complemento a 2 di un fixed-point di dimensione *ws*, scalamento S, base B è mostrato in figura(B.4):

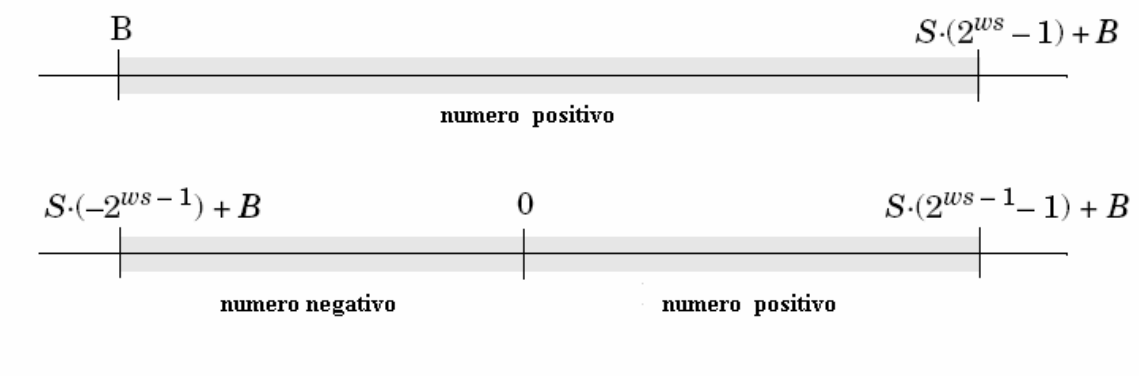


Fig.B.4: *range* di rappresentabilità.

La tecnica "modulo e segno" rappresenta il modulo con n-1 cifre e il segno è determinato dalla cifra binaria dio testa: "0" numero positivo, "1" numero negativo.

Il range di rappresentabilità si estende da $-2^{n-1}+1$ a $2^{n-1}-1$, in questo caso si ha una doppia rappresentazione per lo zero (+0 , -0).

Il "complemento a uno" rappresenta i numeri negativi come 2^{n-1}-(modulo numero da rappresentare). In pratica è sufficiente negare ciascun bit del modulo. Anche in questo caso si ha la doppia rappresentazione dello zero.

Per eliminare il doppio zero si usa il "complemento a due" che spazia da -2^{n-1} a $2^{n-1}-1$.

Partendo dal complemento a uno aggiungendo +1 si ottiene tale rappresentazione:

dato n si ha:

$$C_2(n) = C_1(n) + 1$$

Appendice C

Guida a Xilinx System Generator

Per la costruzione di un modello mediante questo *toolbox* occorre avviare *Matlab* ed aprire *Simulink Library Browser*, figura (C.1), tramite l'icona presente nel pannello di controllo.

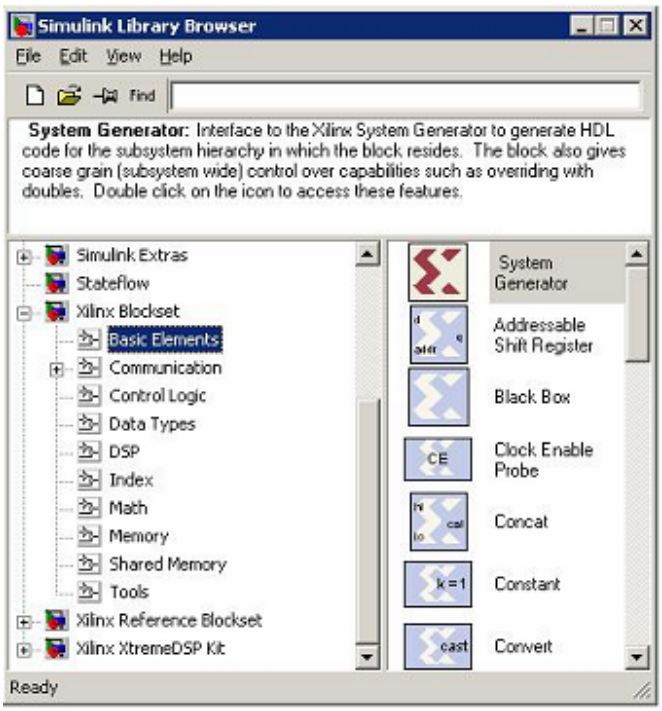


Fig.C.1: *Simulink Library Browser*.

Gli *step* fondamentali necessari per la creazione dell'algoritmo attraverso *System Generator* sono i seguenti:

1. descrizione dell'algoritmo in termini matematici;
2. realizzazione dell'algoritmo usando dati rappresentati in doppia precisione;
3. passaggio dall'aritmetica in doppia precisione al *fixed-point*;
4. conversione del progetto in *hardware* in modo efficiente.

Lo *step* 4 è critico e può essere fonte di errori in quanto è difficile garantire che l'implementazione *hardware* rispecchi fedelmente l'algoritmo creato. *System Generator* elimina questo inconveniente generando automaticamente il codice per l'implementazione *hardware*.

Lo *step* 3 è anch'esso fondamentale, poichè un'efficiente implementazione *hardware* usa precisione *fixed-point*. *System Generator* non esegue automaticamente questo *step*, il quale coinvolge un delicato *trade off* di analisi Purtroppo questo *step* non può essere trascurato, ossia non si possono semplicemente usare in *hardware* operazioni *floating point*, poiché il *range* dinamico richiesto dalla maggior parte delle operazioni è piccolo abbastanza da poter essere trattato da una rappresentazione *fixed point*, che risulta anche a buon mercato.

Simulink fornisce un ambiente grafico per creare e modellare sistemi dinamici. *System Generator* consiste di una liberia chiamata *Xilinx Blockset* e di un *software* necessario a tradurre un modello *Simulink* nella realizzazione *hardware*.

System Generator rileva i parametri definiti in *Simulink* attraverso il *dialog box* dei blocchi *Xilinx Blockset*, e li riporta nella realizzazione *hardware* attraverso *entity*, *architectures*, porte, segnali ed attributi del linguaggio VHDL. Inoltre automaticamente produce *files* di comando per la sintesi di FPGA, la simulazione HDL e l'implementazione, così che l'utente possa lavorare interamente in un ambiente grafico, dalla specifica di progetto alla realizzazione *hardware*.

Xilinx Blockset è accessibile dalla libreria *Simulink* ed i suoi elementi possono essere combinati con quelli degli altri *blockset* già utilizzati. Solo i blocchi e i sottosistemi costituiti dai blocchi *Xilinx Blockset* sono tradotti in *hardware* dal *System Generator*. Il processo di generazione è controllato dal *System Generator block* che si trova nella libreria *Xilinx Blockset Basic Elements*.

Da questo blocco l'utente può scegliere la famiglia di FPGA da utilizzare, il periodo di *clock* del sistema ed altre opzioni di implementazione attraverso il *dialog box* mostrato in figura (C.2).

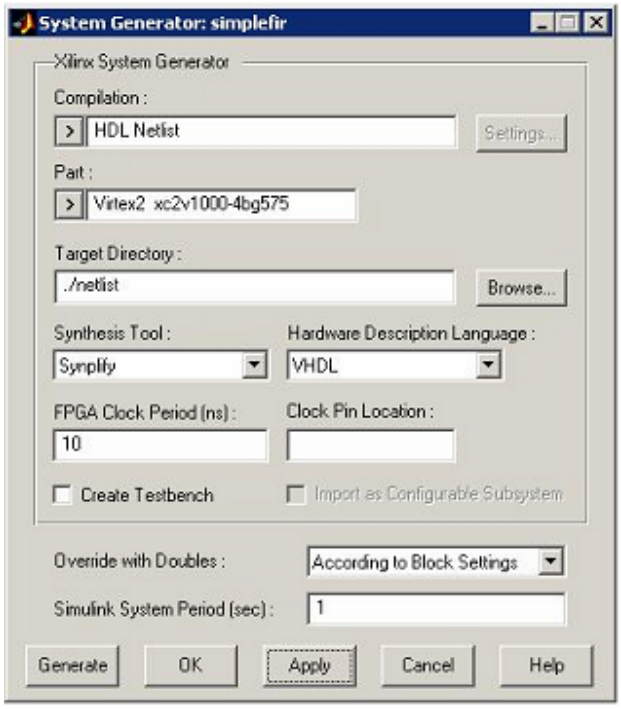


Fig. C.2: *System Generator dialog box.*

System Generator fornisce due librerie *Simulink*, "*Xilinx Blockset"* e "*Xilinx Reference Blockset"*, ognuna contenente blocchi utilizzabili per costruire modelli.

Questi infatti forniscono elementi aritmetici, logici e memorie che permettono l'implementazione del modello in FPGA.

Tipicamente i modelli utilizzano, oltre ai blocchi *Xilinx*, le normali librerie *Simulink*, che permettono di analizzare e visualizzare il comportamento del modello; uno dei principali vantaggi dell'utilizzo di *System Generator* è dato dall'ampia possibilità di simulazioni fornite dall'ambiente *Simulink*.

Ogni segnale presente nel modello, per poter essere tradotto in *hardware*, deve essere campionato e convertito in tipo *fixed-point* o booleano. *System Generator*, a differenza di *Simulink*, non permette di trattare segnali a tempo continuo poiché la sua funzione è la realizzazione di sistemi digitali. Il tipo di dati e il *sample rate* sono propagati attraverso il modello *System Generator* secondo le normali regole di *Simulink*, quindi qualsiasi sorgente deve essere campionata e quantizzata.

La conversione dei segnali dai blocchi *Simulink* ai blocchi *Xilinx* avviene attraverso *Xilinx gateways*, il quale impone il rispetto di tali vincoli di temporizzazione. Con un doppio *click* sul blocco si visualizzano i controlli che permettono di definire il tipo di segnale (figura C.3).

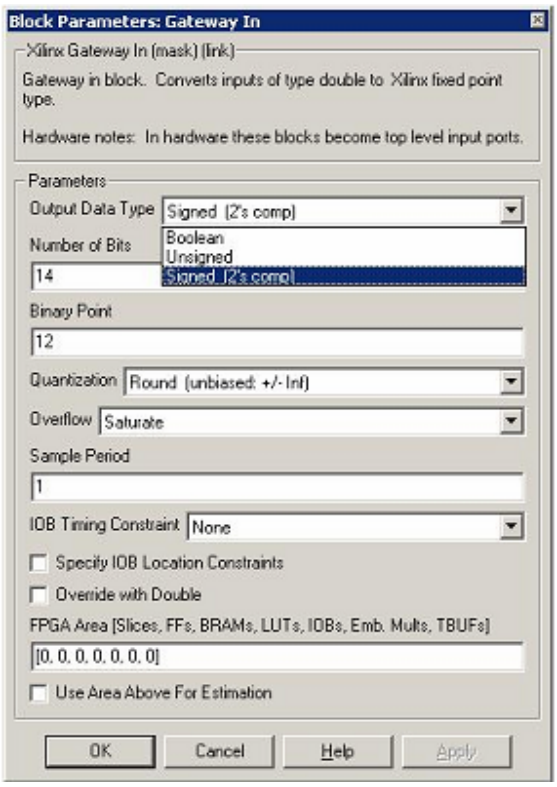


Fig. C.3: *Gateway In dialog box.*

In uscita il tipo di dati può essere "*boolean*"," *unsigned*", "*signed two's complement*" con una precisione numerica definita dall'utente.

La figura (C.3) mostra un *gateway* che impone l'ingresso a 14 bit, *signed two's complement* con 12 *bit* di parte frazionaria.

Un FPGA fornisce una considerevole flessibilità sulla definizione della precisione aritmetica: questo permette di costruire dati con una precisione arbitraria.

La quantità di FPGA usata è strettamente dipendente dall'ampiezza dei dati, così che una variazione della precisione richiesta dall'applicazione può permettere miglioramenti in termini di costo, velocità circuitale e potenza dissipata.

Terminato lo schema del progetto, è possibile ottenere una stima delle risorse impiegate inserendo nel foglio di lavoro il blocco "*Resource Estimator*" illustrato in figura (C.4).

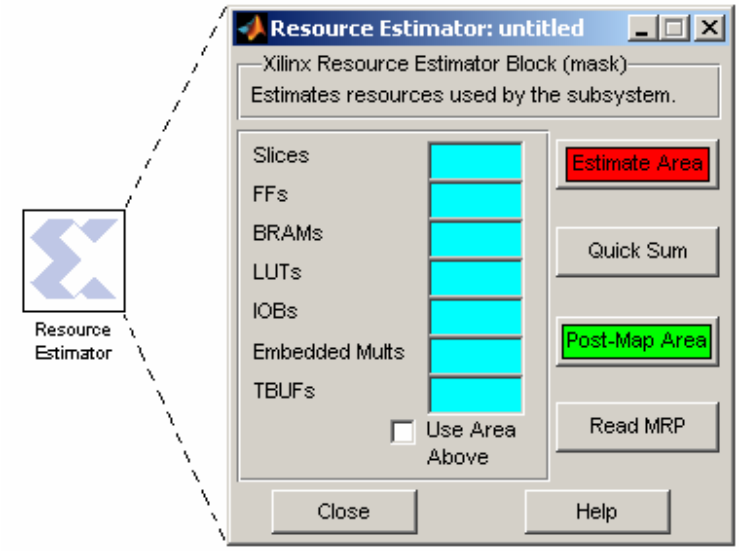


Fig. C.4: dialog box dell'icona "*Resource Estimator*".

Il blocco "*Resource Estimator*" esegue una rapida stima delle risorse che il modello *System generator* richiede all' FPGA. Questa stima è calcolata sommando le risorse impiegate dai singoli blocchi della libreria presenti nel progetto. Ogni blocco ha un vettore nel quale sono memorizzate le quantità di *lookup table* (LUTs), *flip flop* (FFs), blocchi di memoria (*BRAM*), moltiplicatori 18x18, *buffer tristate* e segnali di I/O richiesti. Dalla somma di tutti i vettori otteniamo la stima delle risorse totali necessarie.

Per procedere con la stima delle risorse utilizzate bisogna premere il tasto "*Estimate Area*" evidenziato in rosso in figura (C.4): nei campi azzurri compariranno le quantità delle risorse complessivamente stimate.

Quindi si deve premere "*Post-Map Area*" (evidenziato in verde in figura(C.4)): con questo comando il *software* calcola la percentuale delle risorse occupate nel componente da noi scelto in precedenza (figura (C.2)).

Al termine tutti i dati calcolati saranno riportati in un file *report* con estensione ".mrp". Questo formato è leggibile attraverso il programma X*ilinx Project Navigator.*

Nel *file* sono riportate le percentuali di utilizzo dei singoli componenti; qualora il circuito richiedesse risorse superiori a quelle disponibili compariranno valori maggiori del 100%.

Appendice D

Software utilizzati

- MATLAB Version 7.0.1.24704 (R14) Service Pack 1;
- Simulink Version 6.1 (R14SP1);
- Project Navigator ISE 6.3I;
- Xilinx System Generator for DSP;

Famiglie Xilinx

Mostriamo nelle figure (D.1 e D.2) seguenti le famiglie Xilinx che sono state analizzate per la scelta del dispositivo finale:

Spartan-3

Device	System Gates	Logic Cells	CLB Array (One CLB = Four Slices)			Distributed RAM (bits[1])	Block RAM (bits[1])	Dedicated Multipliers	DCMs	Maximum User I/O	Maximum Differential I/O Pairs
			Rows	Columns	Total CLBs						
XC3S50	50K	1,728	16	12	192	12K	72K	4	2	124	56
XC3S200	200K	4,320	24	20	480	30K	216K	12	4	173	76
XC3S400	400K	8,064	32	28	896	56K	288K	16	4	264	116
XC3S1000	1M	17,280	48	40	1,920	120K	432K	24	4	391	175
XC3S1500	1.5M	29,952	64	52	3,328	208K	576K	32	4	487	221
XC3S2000	2M	46,080	80	64	5,120	320K	720K	40	4	565	270
XC3S4000	4M	62,208	96	72	6,912	432K	1,728K	96	4	712	312
XC3S5000	5M	74,880	104	80	8,320	520K	1,872K	104	4	784	344

(a)

Device	Available User I/Os and Differential (Diff) I/O Pairs																	
	VQ100		TQ144		PQ208		FT256		FG320		FG456		FG676		FG900		FG1156	
	User	Diff	User	Diff	User	Diff	User	Diff	User	Diff	User	Diff	User	Diff	User	Diff	User	Diff
XC3S50	63	29	97	46	124	56	-	-	-	-	-	-	-	-	-	-	-	-
XC3S200	63	29	97	46	141	62	173	76	-	-	-	-	-	-	-	-	-	-
XC3S400	-	-	97	46	141	62	173	76	221	100	264	116	-	-	-	-	-	-
XC3S1000	-	-	-	-	-	-	173	76	221	100	333	149	391	175	-	-	-	-
XC3S1500	-	-	-	-	-	-	-	-	221	100	333	149	487	221	-	-	-	-
XC3S2000	-	-	-	-	-	-	-	-	-	-	-	-	489	221	565	270	-	-
XC3S4000	-	-	-	-	-	-	-	-	-	-	-	-	-	-	633	300	712	312
XC3S5000	-	-	-	-	-	-	-	-	-	-	-	-	-	-	633	300	784	344

(b)

Fig. D.1: (a) caratteristiche dispositivi famiglia Spartan3; (b) tabella package.

Virtex-2

Device	System Gates	CLB (1 CLB = 4 slices = Max 128 bits)			Multiplier Blocks	SelectRAM Blocks		DCMs	Max I/O Pads(1)
		Array Row x Col.	Slices	Maximum Distributed RAM Kbits		18 Kbit Blocks	Max RAM (Kbits)		
XC2V40	40K	8 x 8	256	8	4	4	72	4	88
XC2V80	80K	16 x 8	512	16	8	8	144	4	120
XC2V250	250K	24 x 16	1,536	48	24	24	432	8	200
XC2V500	500K	32 x 24	3,072	96	32	32	576	8	264
XC2V1000	1M	40 x 32	5,120	160	40	40	720	8	432
XC2V1500	1.5M	48 x 40	7,680	240	48	48	864	8	528
XC2V2000	2M	56 x 48	10,752	336	56	56	1,008	8	624
XC2V3000	3M	64 x 56	14,336	448	96	96	1,728	12	720
XC2V4000	4M	80 x 72	23,040	720	120	120	2,160	12	912
XC2V6000	6M	96 x 88	33,792	1,056	144	144	2,592	12	1,104
XC2V8000	8M	112 x 104	46,592	1,456	168	168	3,024	12	1,108

(a)

Package	Available I/Os										
	XC2V 40	XC2V 80	XC2V 250	XC2V 500	XC2V 1000	XC2V 1500	XC2V 2000	XC2V 3000	XC2V 4000	XC2V 6000	XC2V 8000
CS144	88	92	92	-	-	-	-	-	-	-	-
FG256	88	120	172	172	172	-	-	-	-	-	-
FG456	-	-	200	264	324	-	-	-	-	-	-
FG676	-	-	-	-	-	392	456	484	-	-	-
FF896	-	-	-	-	432	528	624	-	-	-	-
FF1152	-	-	-	-	-	-	-	720	824	824	824
FF1517	-	-	-	-	-	-	-	-	912	1,104	1,108
BG575	-	-	-	-	328	392	408	-	-	-	-
BG728	-	-	-	-	-	-	-	516	-	-	-
BF957	-	-	-	-	-	-	624	684	684	684	-

(b)

Fig. D.2: (a) caratteristiche Virtex-2; (b) tabella package

Bibliografia

[1] Vijay K. Madisetti and Douglas B. Williams, *The Digital Signal Processing Handbook*, CRC Press and IEEE Press, 1998.

[2] Wayne Niblack, An Introduction to Digital Image Processing, Prentice/Hall International, 1986.

[3] Bernard Bosi and Guy Bois, "Reconfigurable Pipelined 2-D Convolvers for fast Digital Signal Processing", *IEEE Transaction on Very Large Scale (VLSI) Systems*, Vol. 7, No.3, p 299-308, Sep. 1999.

[4] Cheng-The Hsieh and Seung P. Kim, "A Highly-Modular Pipelined VLSI Architecture for 2-D FIR Digital Filter," *Proceedings of the 1996 IEEE 39th Midwest Symposium on Circuits and Systems*, Part 1, p 137-140, Aug. 1996.

[5] D. D. Haule and A. S. Malowany, "High-speed 2-D Hardware Convolution Architecture Based on VLSI Systolic Arrays", *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, p 52-55, Jun 1989.

[6] K. Hsu, L. J. D'Luna, H. Yeh, W. A. Cook and G. W. Brown, "A Pipelined ASIC for Color Matrixing and Convolution", *Proceedings of the 3rd Annual IEEE ASIC Seminar and Exhibit*, Sep. 1990.

[7] V. Hecht, K. Ronner and P. Pirsch, "An Advanced Programmable 2D-Convolution Chip for Real Time Image Processing", *Proceedings of IEEE International Symposium on Circuits and Systems*, p.1897-1900, 1991.

[8] T. Tommasini, A. Fusiello, E. Trucco e V. Roberto, "Making good features track better", In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition –IEEE Computer Society Press, 1998.
URL: http://www.engin.umd.umich.edu/~jwvm/ece581/21_Gblur.pdf

[9] Xilinx Company
URL: http://www.xilinx.com/bvdocs/publications/ds099.pdf

[10] Xilinx Company System Generator
URL: http://www.xilinx.com/products/software/sysgen/app_docs/user_guide.htm

[11] Mathworks
URL: http://www.mathworks.com/access/helpdesk/help/toolbox/simulink

[12] Xilinx Company XST
URL: http://toolbox.xilinx.com/docsan/xilinx7/books/docs/xst/xst.pdf

[13] Omnivision Company
URL: http://www.ovt.com

[14] Datasheet OV7640
URL: http://www.ovt.com/p_cameraChips.html#1m

[15] Datasheet Cypress CY7C68013
URL:
http://www.cypress.com/portal/server.pt?space=CommunityPage&control=SetCommunity&CommunityID=209&PageID=259&fid=14&rpn=CY7C68013

[16] J.Foley, A. Van Dam, S. Feiner, J. Hughes, "Computer Graphics" Wesley 1996, p.592