Relazione sul Progetto di Reti Logiche

Stefano Pelletti, codice persona 10672854

Politecnico di Milano

prof. Gianluca Palermo

consegna di Aprile 2021

Sunto

Il presente documento ha il compito di illustrare quello che è stato il percorso di sintesi di un componente logico in grado di rispettare una determinata specifica fornita dal docente. Precisamente, vengono presentate una rielaborazione della specifica traendone le opportune ipotesi, gli obiettivi che sono stati posti (sia di progetto che personali), una descrizione dell'architettura e del comportamento del componente, le scelte progettuali e implementative adottate, eventuali considerazioni e osservazioni possibilmente di interesse, le analisi sperimentali ed i relativi risultati, e infine delle conclusioni in merito all'esperienza.

La relazione è stata scritta in modo da "introdurre" il lettore alla logica di funzionamento del componente. Pertanto, almeno le parti iniziali, sono volutamente un poco più discorsive ma più chiare.

Note di utilizzo: linguaggio di descrizione hardware utilizzato: VHDL,
tool integrato: VIVADO, board di prova: FPGA xc7a200tfbg484-1.
Nei diagrammi sarà sempre espressa la direzione dei segnali, per facilitare la lettura.

Relazione sul Progetto di Reti Logiche

Introduzione

La specifica del progetto prevede, ad alto livello, di implementare un componente che sia in grado di eseguire una semplice equalizzazione di immagine. Più precisamente, vengono caricati in una memoria RAM, modellizzata come un array di byte, le informazioni di base relative all'immagine: numero di righe nel byte [0], numero di colonne nel byte [1], e i vari livelli di colore dei pixel dell'immagine a seguire.

Il componente deve essere in grado di svolgere il suo compito e caricare in RAM, subito dopo l'immagine originale, l'immagine equalizzata.

Il compito del componente è quello di:

1. Scansionare i dati di ingresso una prima volta per cercare Massimo e Minimo valore. Da questa prima scansione verranno determinati alcuni valori utili, come il DELTA e lo SHIFT_LEVEL, per poter attuare l'equalizzazione.
2. Scansionare una seconda volta i dati, applicando una opportuna trasformazione presente nel documento di specifica, e quindi riscriverli in RAM.

Assunzioni da specifica: RAM[0] (#C, numero di colonne) e RAM[1] (#R, numero di righe) possono contenere valori da 0 a 128 (inclusi). L'immagine originale, costituita da quindi da #R * #C Byte, ha inizio da RAM[2] e ha fine in RAM[#R*#C+1]. Ci si aspetta che l'immagine equalizzata si trovi in posizione da RAM[#R*#C+2] fino a RAM[2+2*(#R*#C)]. Il protocollo stabilisce che l'esecuzione parta quando viene posto alto il segnale "start", e la procedura sia considerata terminata quando viene posto alto il segnale "done". Il componente, dopo una prima equalizzazione a seguito di un segnale di reset, deve poter essere in grado di ripetere l'equalizzazione (eventualmente su una diversa immagine) senza che gli venga fornito un secondo segnale di reset.

Descrizione del comportamento del componente *"top"*

Dalla specifica possiamo ricavare una serie di passi indispensabili:

I. Attesa: finché non viene fornito un segnale di "start", il componente attende.

A. Avvio: il componente recupera dalla RAM i valori #R e #C, per determinare fino a dove deve spingersi nella memoria RAM per fare la prima scansione.

B. Primo Loop: accedendo alla memoria e leggendo valore dopo valore, bisogna determinare il "MAXvalue" e il "MINvalue".

C. Calcolo: una volta ottenuti i due valori, che rappresentano gli estremi del range dei possibili colori, va determinato il DELTA value e lo SHIFT_LEVEL, col quale è possibile applicare l'equalizzazione su ciascun pixel dell'immagine.

D. Secondo Loop: scansionando uno ad uno i valori dei pixel dell'immagine originale, applicare la trasformazione e ricaricarli in RAM in posizione corretta.

E. "End" : il componente si riporta in uno stato dal quale può iniziare nuovamente l'equalizzazione, alza il segnale di "done" e attende che il segnale di "start" venga abbassato.

Questi sei passi corrispondono esattamente ai sei "macro-stati" utilizzati dal componente *top*:

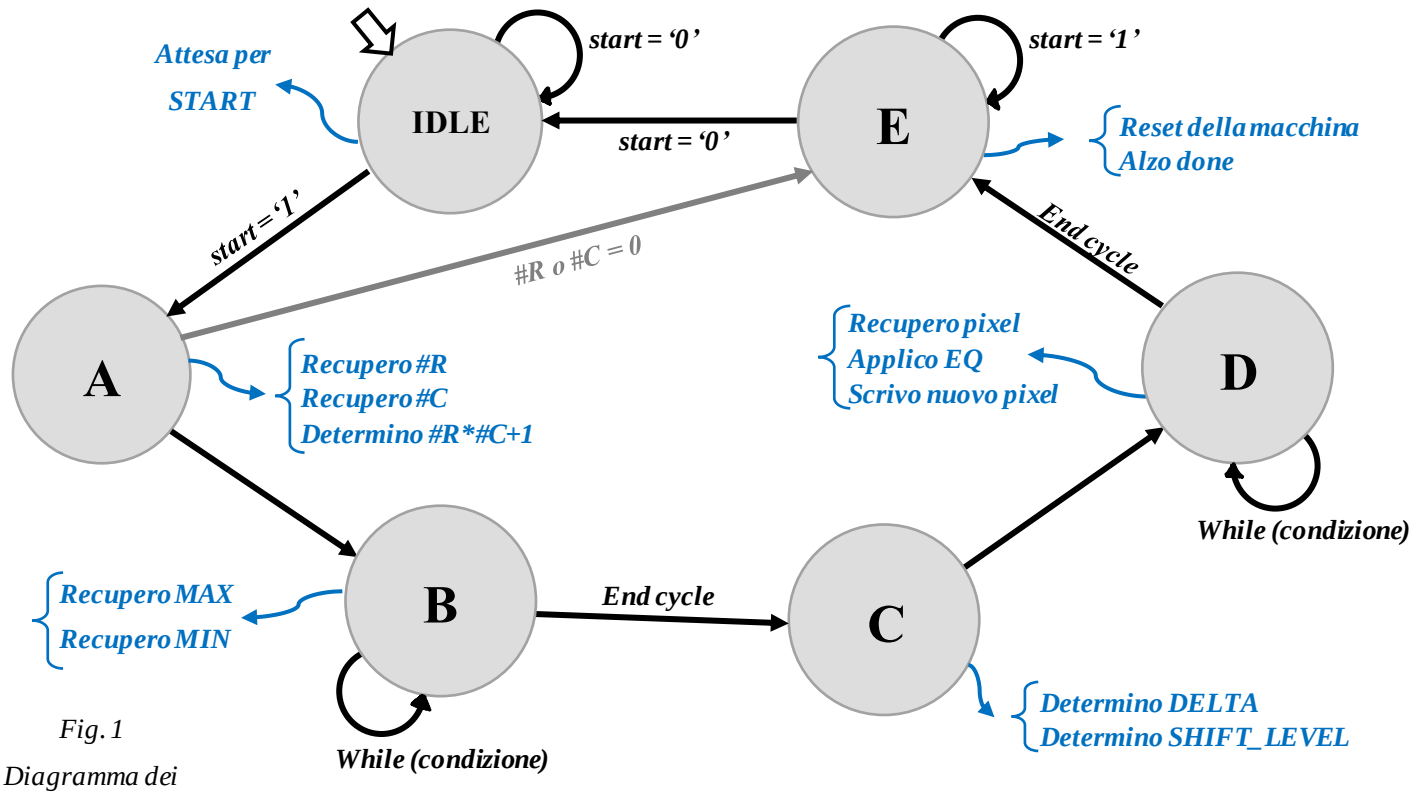


Fig. 1
Diagramma dei macro-stati

Diversi sono gli approcci possibili: leggere ogni pixel e memorizzarlo in una RAM interna per poi non doverli rileggere, usare processi d'alto livello per poi lasciar fare la sintesi al tool, usare funzioni e moduli preconfezionati…

In questo caso è stato scelto di implementare una sorta di "controller" che pilota gli ingressi dei sotto-componenti dai quali sarà composto, senza, di per sé, fare elaborazioni complesse.

Dal diagramma degli stati precedente sorgono diverse esigenze:
Bisogna tenere traccia dei valori di #R, di #C, così come dell'indice dell'ultimo byte, i Massimi e Minimi. Bisogna almeno un contatore per scandire la memoria, e bisogna trovare un modo per fare l'elaborazione dell'immagine.

È stato quindi scelto di introdurre dei sotto-componenti, tra i quali delle **memorie** e un componente che calcola il nuovo valore del pixel, chiamato **ShiftCalculator**.
Usando queste due sotto-componenti, e pilotando i loro ingressi con una **Macchina a stati finiti**, viene implementato completamente il componente *"top"*.

Dopo aver introdotto i moduli della memoria e dello ShiftCalculator, verrà presentato il diagramma architetturale completo, per "riunire i pezzi" illustrati.

a) **Memorie : Generic8/16bitMemory**

Compito delle memorie è di tenere traccia di un numero binario naturale.

In questo caso, tuttavia, si è voluto rendere più "intelligente" il comportamento delle memorie: utilizzando due bit di *controllo*, un modulo di memoria può fare diverse cose:

- **"00" : IDLE :** la memoria resta "ferma" sul valore memorizzato precedentemente, non memorizza nuovi valori, e presenta gli stessi output dell'istante precedente.
- **"01" : ASSIGN :** la memoria memorizzerà il numero fornito in input.
- **"10" : INC :** la memoria funziona come se fosse un contatore: ad ogni ciclo di clock il valore mostrato in output aumenta di uno. Continua finché non raggiunge il valore fornitogli in input (valore target). Quando lo raggiunge, comunica che l'ha raggiunto tramite un apposito segnale ("done"), e si riazzera.
- **"11" : DEC :** in modo simile a INC, la memoria decresce fino a raggiungere il valore 0.

Le memorie usate sono componenti sincrone, e fin dal principio è stato scelto di usare memorie da 8 o 16 bit, anche con eventuali sprechi (poi soppressi da VIVADO).

Nel componente sono stati utilizzate 3 memorie da 8 bit, 2 da 16 bit, e una variante* da 8 bit:

nome	capacità (bit)	ruolo (valore in memoria)
m1memory	8	#R
m2memory	8	#C
m3memory	16	#R*#C + 1
m4memory	16	Counter
MAXmemory	8	MAX value
MINmemory*	8	MIN value

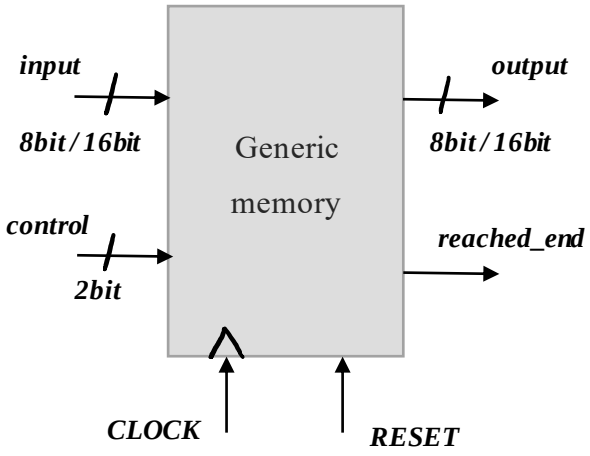


Fig. 2
Modello di GenericMemory

A livello di implementazione VHDL una memoria generica è costituita da un segnale interno "count" che viene aggiornato nell'unico processo "UpdateProcess" presente, a seconda delle modalità di controllo sopra descritte.

È quindi VIVADO che provvede a inserire l'elemento di memoria per implementare il registro vero e proprio.

È presente anche un secondo segnale interno "count_done" che rappresenta la fine del conteggio, nel caso si stia usando la memoria come contatore.

I due segnali interni sono quindi legati alle porte di output:

```
( output <= count,  reached_end <= count_done )
```

Nel file .vhd sono presenti tre architetture per le memorie usate, tutte 'uguali' a meno delle grandezze delle parole registrate (o per il valore di reset nel caso della variante*).

b) **ShiftCalculator**

Per fare le operazioni di equalizzazione sui pixel, si è scelto di implementare una componente ad hoc. Lo ShiftCalculator possiede una memoria interna da 8 bit per lo SHIFT_LEVEL.

Questa componente possiede anch'essa delle modalità di funzionamento:

- **"00" o "11" : IDLE :** il valore in uscita è posto a 0 e la memoria trattiene il valore memorizzato.
- **"01" : ASSIGN :** lo ShiftCalculator riceve in input il DELTA value, e provvede ad eseguire una serie di operazioni per calcolare lo SHIFT_LEVEL.

  Il calcolo dello SHIFT_LEVEL SL è eseguito come segue:

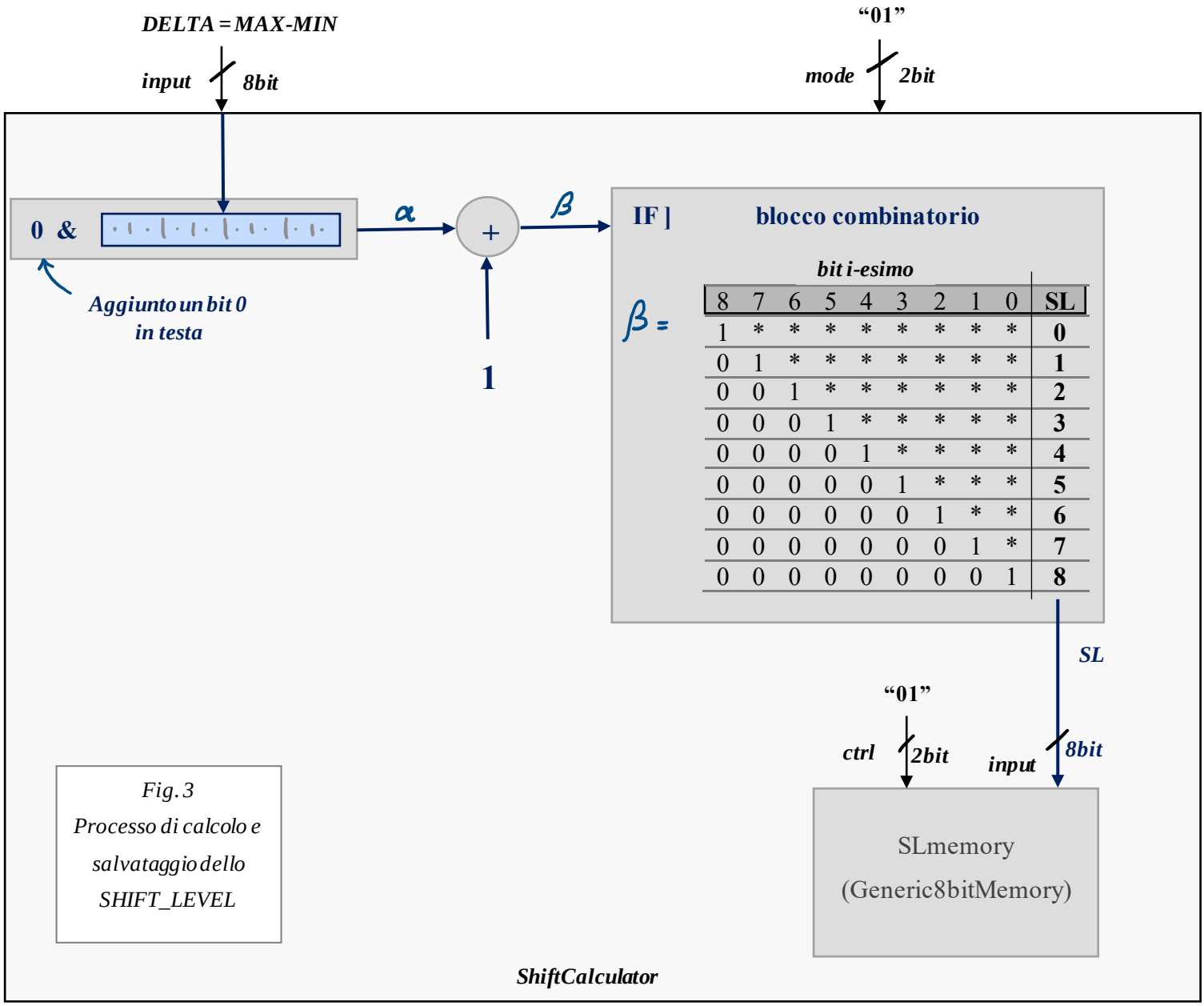


Fig. 3
Processo di calcolo e salvataggio dello SHIFT_LEVEL

- **“10” : ESECUZIONE:** a differenza delle due altre modalità questa è volutamente asincrona. Lo ShiftCalculator riceve in input il MIN_value (memorizzato in MINmemory) e il valore del pixel corrente. Di conseguenza, esegue l’operazione:

  output <= (input – MINinput) << SL ; “istantaneamente”

  L’operazione di Shift è anch’essa codificata in un blocco simile a quello per il calcolo dello Shift Level nella modalità ASSIGN. Il MINinput è volutamente lasciato come input al componente, e viene “cablato in modo fisso” alla memoria MINmemory. (Per brevità, una bozza di diagramma per questa modalità viene inserita nel diagramma di modulo sottostante in Fig. 4)

A livello di implementazione VHDL si hanno quindi una memoria interna e delle variabili che rappresentano collegamenti logici per fare alcuni step intermedi di elaborazione (alpha e beta). Il tutto è gestito da un unico processo che può agire in modalità sincrona (“00”, “11”, “01”) o asincrona (“10” : esecuzione), seguendo una struttura simile a quella usata per le memorie.

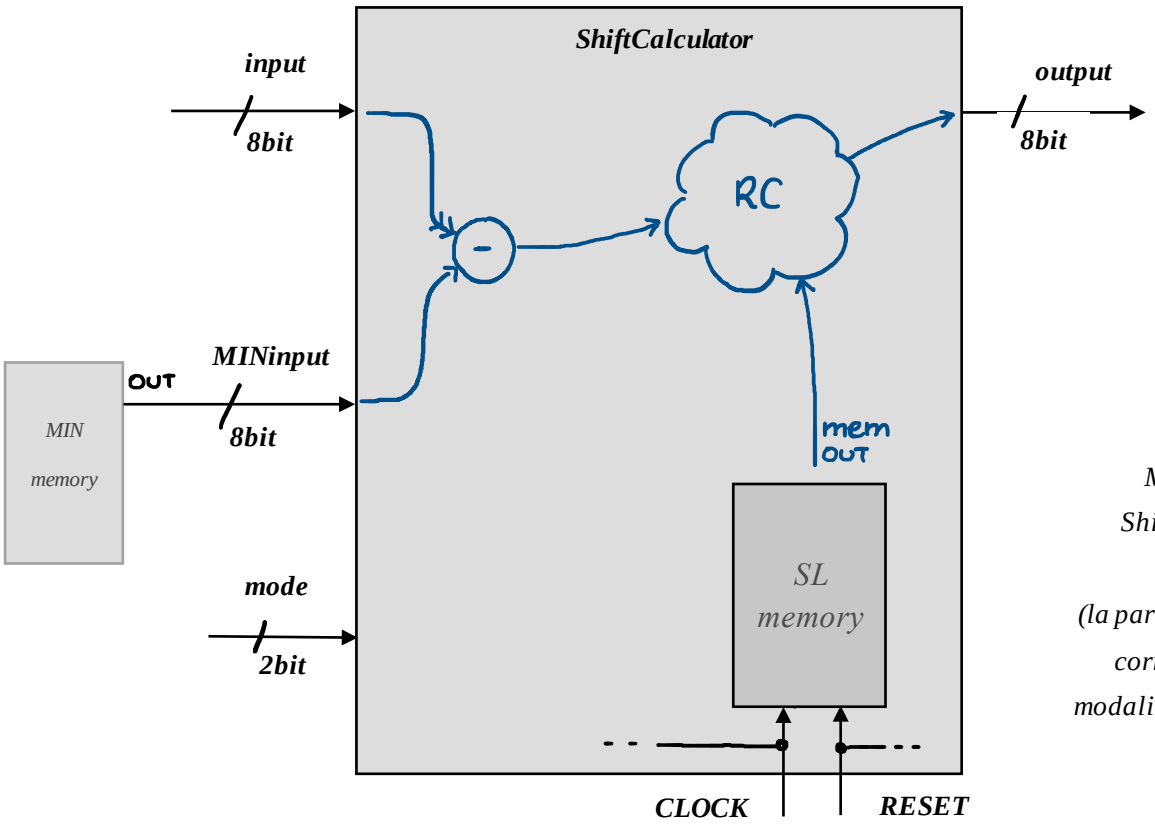
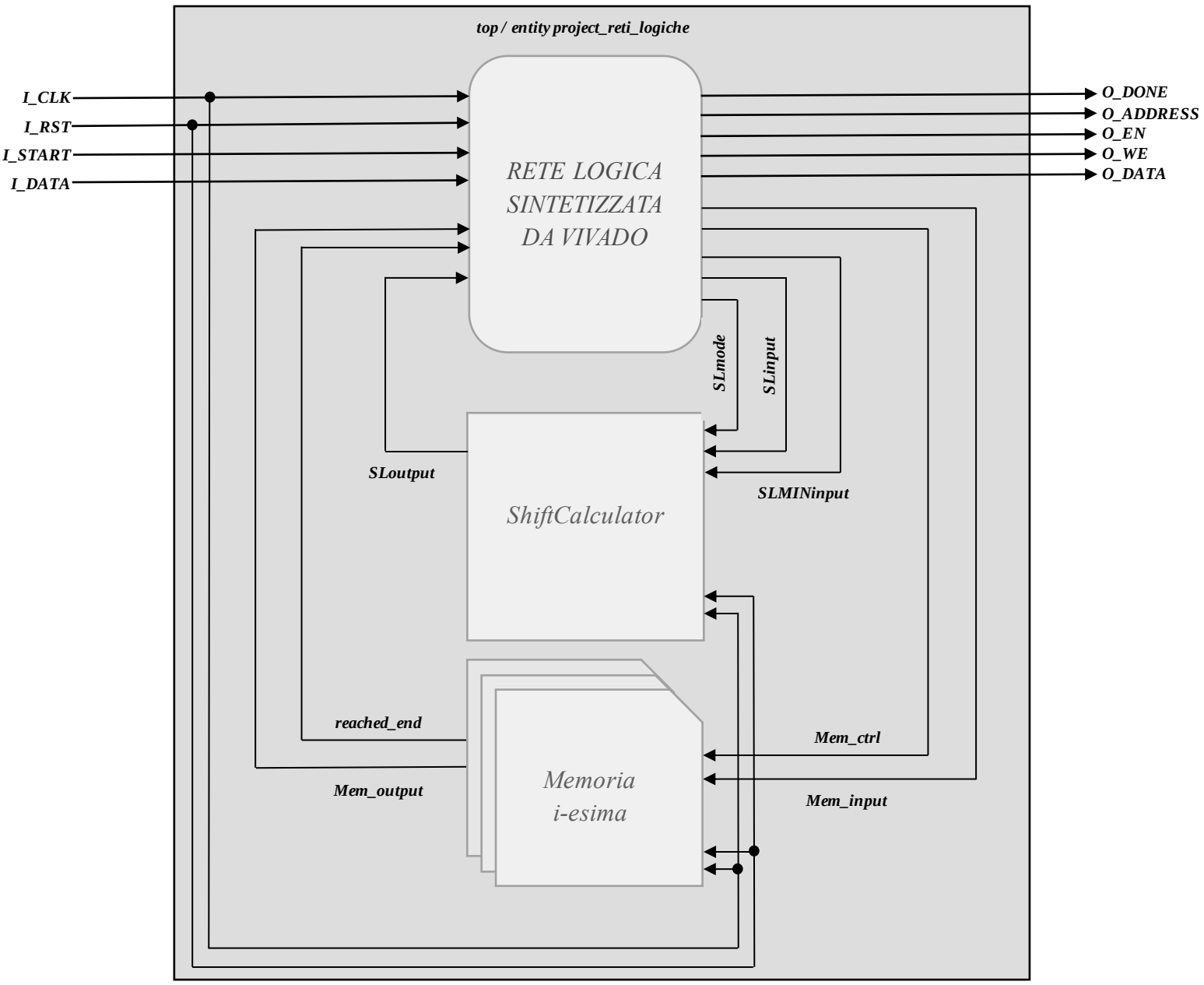


Fig. 4
Modello di ShiftCalculator
-
(la parte segnata in blu corrisponde alla modalità di Esecuzione)

Architettura del componente “top”

Si possono ora riunire i pezzi: il componente “top” è composto dalle 6 memorie descritte prima e dallo ShiftCalculator. È una macchina a stati finiti sincrona, con sei macrostati (I A B C D E). Il componente “top” agisce come un controller: a seconda dello stato interno e degli input, pilota gli input alle sue sotto-componenti e gli output all’esterno.

Nel file .vhd è chiamato *project_reti_logiche*. Viene riportato il suo modello (Fig. 5).



Per semplicità di lettura, le 6 memorie sono condensate in un unico blocco, e i segnali ad esse (almeno 12 direttamente dalla rete di controllo e tutti distinti) sono riportati una unica volta.

Fig. 5
Modello di project_reti_logiche

Comportamento nello specifico

Lo stato interno è implementato usando due segnali enumerati current_state e next_state. Per aiutare nella comprensione del circuito vengono riportati anche alcuni grafici di simulazione.

Per via del funzionamento sincrono sia della memoria RAM esterna che delle componenti interne, alcune operazioni (come l'assegnamento delle memorie) e la richiesta del byte i-esimo dalla RAM hanno bisogno di più step. Per questo motivo i sei macro-stati sono suddivisi in ulteriori stati (ognuno consecutivo all'altro, a meno che non sia specificato):

- **IDLE:** come prima, la macchina a stati cicla su questo stato fintanto che il segnale i_start non viene posto a 1. Le componenti interne sono anch'esse tutte poste in modalità Idle, e gli output sono tutti posti a 0.
- **A0:** prima azione: viene richiesto alla RAM il byte 0, alzando o_en a 1 e ponendo o_address a 0.
- **A1:** seconda azione: settando m1ctrl a "01" viene imposto a m1memory di memorizzare il byte 0, in arrivo sul bus i_data con qualche nanosecondo di delay. Inoltre, viene richiesto alla RAM il byte 1.
- **A2:** terza azione: il byte 1 in arrivo dalla RAM viene memorizzato in m2memory. Viene chiusa la porta di input di m1memory imponendo m1ctrl a "00" (idle). o_en viene portato a 0.
- **A3:** quarta azione: viene chiusa la porta di input di m2memory.
- **A4:** quinta azione: vengono controllati i valori di m1memory e m2memory. Se uno dei due è zero, si fa un Jump allo stato E0. Altrimenti, in m3memory viene assegnato il valore `(m1output*m2output + 2)`.
    - **[nota:** non viene fatto alcun controllo sul fatto che sia al massimo da 128x128 pixel. È stato compreso come "dimensione massima della foto" da 16kB. È presente un limite massimo della dimensione in kB, che è di poco meno di 32kB. Complessivamente le due immagini in memoria devono stare sotti ai 65kB, altrimenti i 16bit delle memorie non riescono più correttamente a rappresentare gli indirizzi e il componente non funziona più bene.]
- **A5:** sesta azione: il contatore m4memory viene inizializzato a 2, il primo pixel da leggere. La porta di input di m3memory viene chiusa.

- **A6:** settima azione: viene cablato all'input di m4memory il valore (m3output -1), ossia l'ultimo pixel della foto originale. Quando il contatore raggiungerà questo valore porrà m4end a 1.

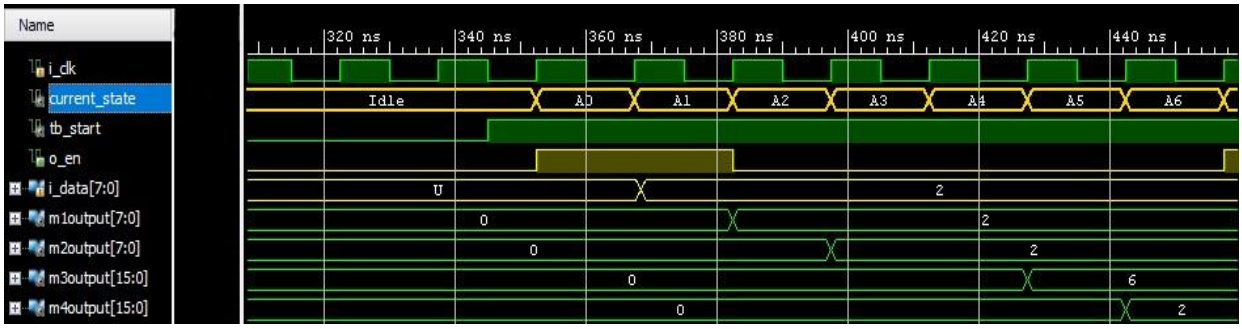


Fig. 6 – step di simulazione per gli stati Idle e A:
Sono stati riportati i valori di uscita delle memorie m1, m2, m3 e m4. L'immagine test è una 2x2.
Da notare che gli assegnamenti hanno effetto dopo un ciclo di clock.

- **B0:** loop ½: è da pensare come un while: se il contatore ha finito di contare, si passa a C0, altrimenti si chiede alla RAM il prossimo byte da ricevere.
- **B1:** loop ½: si riceve in input il pixel sul bus i_data. Esso viene confrontato con le memorie MAX e MIN (inizializzate col RESET a 0 e a 255 rispettivamente). Se i confronti sono positivi, MAXmemory / MINmemory memorizzano il valore, altrimenti rimangono chiuse all'assegnamento. m4memory viene incrementato di uno, e si torna in B0. (gli ingressi di MAXmemory e MINmemory verranno chiusi subito in B0).

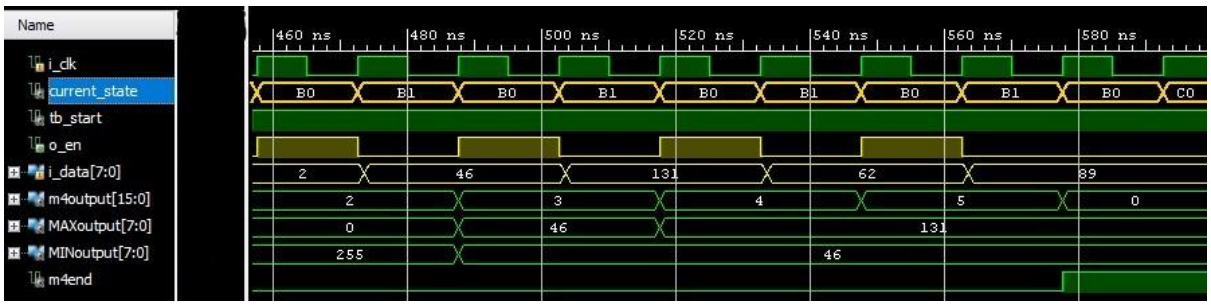


Fig. 7 – step di simulazione per il loop state B:
Sono stati riportati i valori di uscita del contatore m4 e delle memorie MAX e MIN.
Valori in memoria: [46, 131, 62, 89]. *MAX* = 131, *MIN* = 46.

- **C0:** ShiftCalculator viene posto in modalità assegnamento, e gli viene passato in input MAXinput – MINinput (il DELTA). Il contatore m4 viene reinizializzato a 2.
- **C1:** l'input di m4memory viene assegnato nuovamente a (m3memory-1).
- **C2:** lo ShiftCalculator viene posto in modalità esecuzione.

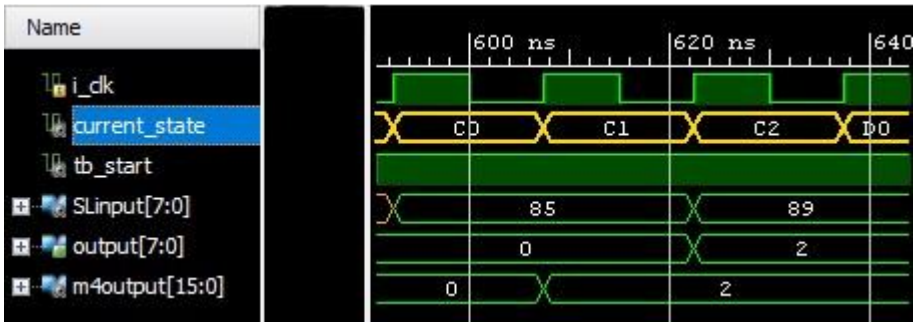


Fig. 7 – simulazione per stato C:
Sono stati riportati: l'output di m4, il DELTA passato in input allo ShiftCalculator e lo ShiftLevel (output) calcolato.

- **D0:** loop ½: in modo analogo a B0, è da pensare come un while: se il conteggio è terminato, si avanza allo stato E0. Altrimenti, viene chiesto alla RAM il prossimo byte da leggere e si passa allo stato D1.
- **D1:** loop ½: il byte ricevuto viene elaborato dallo ShiftCalculator. Viene chiesto alla RAM di scrivere all'indirizzo (m4output+m3output-2) il risultato dello ShiftCalculator. Questo è possibile poiché i pixel delle due immagini sono sempre distanti tra loro (m3output-2), che è di fatti un offset. Infine, si fa avanzare il contatore.

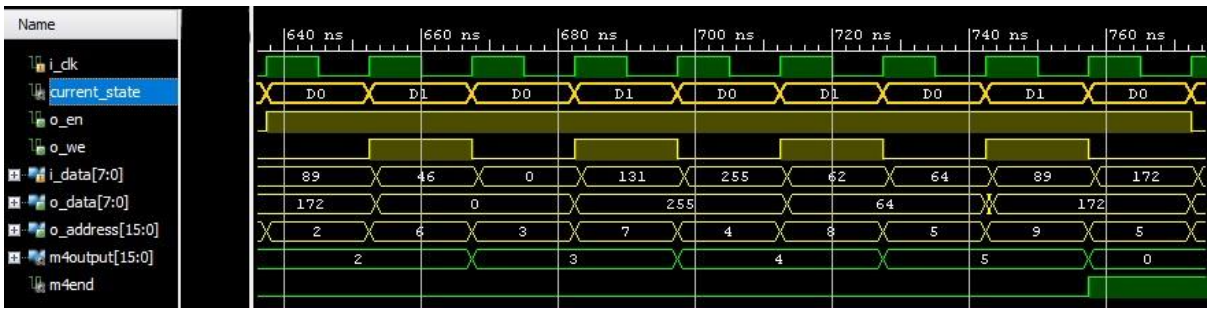


Fig. 8 – simulazione per loop state D:
Sono stati riportati: i segnali di enable e write_enable, i valori dei pixel in entrata, i valori dei pixel in uscita e gli indirizzi alla RAM.

- **E0:** questo è uno stato di servizio, usato per "resettare" tutte le componenti interne.
- **EndState:** la macchina viene posta in questo stato, dove comunica di aver terminato alzando il segnale o_done. Cicla in attesa che il segnale i_start venga riportato basso. A quel punto la macchina torna nello stato di Idle.

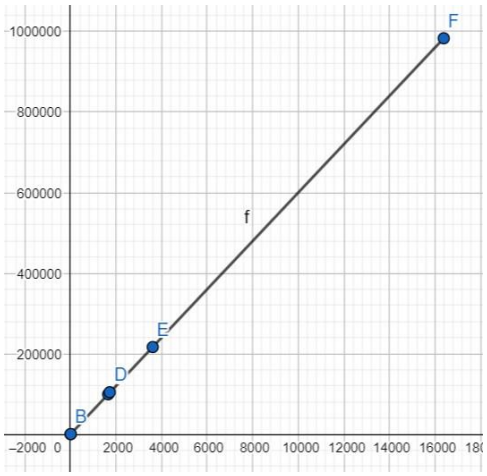
Alcune considerazioni

Il circuito è stato pensato per non essere complesso, senza ottimizzazioni particolari e volutamente senza il riutilizzare alcune risorse, come le memorie m1 e m2 che sono usate solamente nelle fasi iniziali del processo.

In termini di efficienza, il circuito utilizza pienamente le risorse solo in fase di scrittura (state D), mentre, per via di come funzionano le memorie, in lettura (state B) ci impiega lo stesso tempo, che idealmente dovrebbe essere la metà.

Con un clock period di 15 ns, alcuni test opzionali (funzionali post-implementazione) hanno riportato:

Image size (pixel)	Time elapsed from Start to Done
2x2 (4)	442 ns
4x3 (12)	922 ns
69x24 (1656)	99’565 ns
42x41 (1722)	104’662 ns
95x38 (3610)	216’805 ns
128x128 (16384)	983’242 ns



Dopo aver inserito in un grafico i dati, è evidente che sussiste una relazione lineare tra il tempo di elaborazione e il numero di pixel.

$$T(n, c) = 11c + c/2 + 2(2n+1)c$$

*Dove **n** è il numero di pixel dell’immagine,*
*e **c** è il periodo di clock*

È inoltre evidente che ulteriori ottimizzazioni sulle operazioni negli stati sono possibili, tuttavia, per mantenere una certa coerenza e per l’esiguo vantaggio in termini di tempo, si è preferito non modificare ulteriormente il codice già funzionante.

Il tempo ideale, guardando l’equazione appena sopra, è molto vicino a un “tempo costante” + 2nc, dovuto al tempo per leggere (nc) + il tempo per scrivere (nc).

Risultati sperimentali

A) Sintesi

Il componente è sintetizzabile (e implementabile) correttamente con VIVADO.

Il report di sintesi presenta 34 warnings, dovuti per metà a segnali non utilizzati (e soppressi da vivado) e per il resto per l'assegnamento dei segnali, che non vengono ripetuti in tutti gli stati ma solo quando devono essere cambiati, e per questo sintetizzati come dei LATCH.

Altre informazioni d'interesse: 96 celle utilizzate, 38 porte I/O, e 476 reti, una TJ di 45.6 gradi, una potenza assorbita di circa 8W.

B) Simulazioni e testbench

Tutte le simulazioni di seguito riportate sono state eseguite come "post-implementation functional simulation". Si riporta una sola volta l'intera forma d'onda che ci si aspetta.

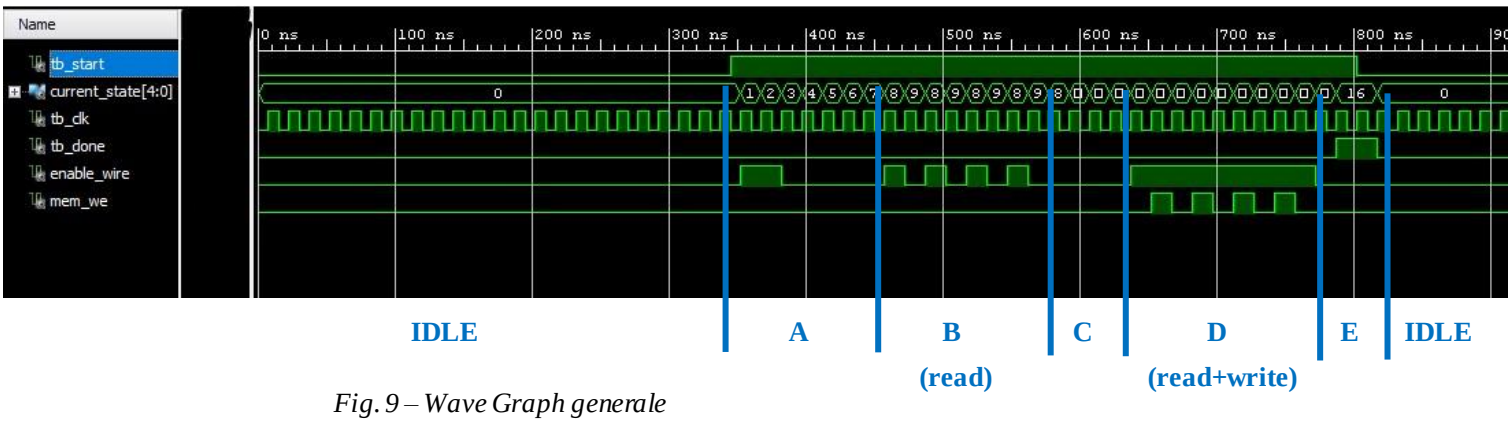


Fig. 9 – Wave Graph generale

Sono indicate con una barra le varie fasi/macrostati, che risultano evidenti a vista d'occhio semplicemente guardando i segnali alla RAM.

Nella tabella seguente vengono riportati i vari test effettuati, con un nome autodescrittivo, il clock period e la dimensione immagine, e le relative motivazioni.

Per generale funzionamento s'intende: che la simulazione termina correttamente, che l'equalizzazione sia applicata corretta, e che la macchina si comporti nel modo previsto.

I casi particolari vertono su condizioni limiti, come: assegnare un massimo alla fine del ciclo, o un ciclo che termina immediatamente.

Nome test	**dimensione**	**clock period**	**motivazione**
TB di esempio	2x2	15ns	Test generale funzionamento
TB memoria 0 delay	4x3	15ns	Test dei segnali alla memoria sui fronti del clock
TB con 1 pixel	1x1	15ns	Test 'caso particolare'
TB con immagine monocromatiche	4x3	15ns	Test 'caso particolare'
TB con primo, secondo, o entrambi byte a 0	0x0, *x0, 0x*	15ns	Test 'caso limite'
TB con reset asincrono	2x2	15ns	Test generale funzionamento
TB iniziale con clock aumentato o diminuito	4x3	5/150ns	Test generale funzionamento
TB con memoria che agisce nella parte di clock "bassa"	2x2	10ns	Test generale funzionamento
TB con gli esempi sul documento di specifica	4x3	15ns	Test generale funzionamento/controllo corretta calibration dell'EQ.
TB con reset a metà esecuzione	4x3	15ns	Test generale funzionamento
TB con massima occupazione	128x128	15ns	Test 'caso limite'
TB con pixel tutti a 0, 1, o tutti a 255	2x4	15ns	Test corretto calcolo dello SL
TB con numero minimo per ultimo e numero massimo per primo o viceversa, o entrambi	2x4	15ns	Test 'caso particolare'
TB con 3 immagini in sequenza	5x7, 9x4, 2x6	5ns	Test generale funzionamento

TB immagine molto grande e nessuna equalizzazione necessaria	60x30	10ns	Test generale funzionamento
TB con 16 immagini	4x2	10ns	Test generale funzionamento
TB con multiple immagini e reset intermedi	15x10, 10x5, 4x4	15ns	Test generale funzionamento
TB intervallo di colori ridotto	4x3	15ns	Test generale funzionamento
TB sulla stessa immagine per 10 volte	4x3	15ns	Test generale funzionamento
TB con stesse dimensioni ma diversi byte iniziali	6x2, 2x6, 4x3, 1x12	15ns	Test generale funzionamento
TB con memoria con delay 10ps e altissimo clock	2x2	500ps	Test generale funzionamento (a 200ps inizia a non funzionare più)

I TestBench si sono concentrati sul corretto funzionamento del componente, in tutte le situazioni possibili ‘lecite’.

TestBench puramente di prova come immagini 130x2 funzionano, ma, in alcune situazioni davvero particolari, potrebbe non funzionare a dovere.

Conclusioni

L’esperienza del progetto si è rivelata assolutamente d’interesse.
Se da una parte ci si è dovuti scontrare con l’utilizzo di un nuovo software, dall’altra quest’ultimo fornisce un aiuto non da poco, anche per una utenza “inesperta”.
Ancora sono molti gli aspetti non totalmente chiari, come la gestione del timing e l’integrazione tra multiple componenti distinte tra loro.

Per quanto riguarda il progetto in sé, alla fine si è riusciti ad ottenere un risultato non male, senza la necessità di dover fare particolari compromessi, e potendo anche generare componenti simili tra loro in modi quasi totalmente diversi.

Infine, si vorrebbe ribadire come questa relazione, insieme al file .vhd, sono stati pensati per rimanere comprensibili anche dopo un po’ di tempo senza maneggiare la materia.

Grazie per l’attenzione

Stefano Pelletti, codice persona 10672854

Politecnico di Milano, 2021