PROGETTO DI RETI LOGICHE

Politecnico di Milano | Losavio - Jin

PREFAZIONE

Questo progetto è stato svolto dai seguenti studenti:

Andrea Losavio: **10522569** – 844670

Songle Jin: **10483883** – 825575

Documentazione utilizzata:

VHDL.pdf (Ferrandi),
11_introduzione_al_vhdl.pdf (Palermo),
VHDL-Cookbook.pdf (Palermo),
First-Start-with-Vivado.pdf (Palermo);

I test sono stati effettuati su 2 macchine:

Desktop PC: Windows 10 for Education x64 / CPU: i7 4790k – 4 GHz

Laptop PC: Windows 10 Home Edition x64 / CPU: i7 6500U – 2.6 GHz

La realizzazione del progetto ha seguito questi steps: Studio del progetto dalla specifica e dai testbench, stesura in linea teorica della FSM e dell'algoritmo di ricerca, stesura del codice.

INTRODUZIONE

Lo scopo del progetto è quello di realizzare un componente hardware (HW) descritto in VHDL che, fornita un'immagine in scala di grigi, calcoli l'area del rettangolo minimo che circoscrive totalmente una figura di interesse.

L'immagine è distribuita in una memoria di dimensione 2^{16} byte in sequenza ed è rappresentata in forma di matrice (con massima dimensione 255 x 255). La comunicazione con la memoria avviene nel momento in cui il segnale 'enable wire' è alto.

Il modo in cui è distribuita la memoria è il seguente:

Address: 0 – 1 → Area del rettangolo: bit meno significativi – bit più significativi.
Address: 2 – 4 → Numero colonne matrice – numero righe matrice – soglia.
Address: 5 – 65535 → Matrice dell'immagine in scala di grigi (0 – 255).

Per semplicità si immagina che l'elemento (n, m) della matrice sia un pixel.
La figura di interesse si basa su un semplice principio di 'soglia' (threshold). Ogni pixel che supera codesta soglia viene preso in considerazione.
Applicando un determinato algoritmo (spiegato successivamente) il componente sarà in grado di decidere qual è la 'base' e qual è la 'altezza' del rettangolo e con una semplice moltiplicazione si otterrà la 'area'.
Questa 'area' verrà salvata nella memoria nell'address 0 e 1, abilitando il segnale di 'write enable' e disabilitandolo al momento opportuno (a fine scrittura).

SCELTE PROGETTUALI

Entità:

Il progetto si basa sull'entità base:

```
entity project_reti_logiche is
    port (
            i_clk          : in std_logic;
            i_start        : in std_logic;
            i_rst          : in std_logic;
            i_data         : in std_logic_vector(7 downto 0);
            o_address      : out std_logic_vector(15 downto 0);
            o_done         : out std_logic;
            o_en           : out std_logic;
            o_we           : out std_logic;
            o_data         : out std_logic_vector(7 downto 0)
        );
end project_reti_logiche;
```

Composto da quattro segnali input:

i_clk è il segnale di CLOCK in ingresso generato dal TestBench;

i_start è il segnale di START generato dal TestBench;

i_rst è il segnale di RESET che inizializza la macchina;

i_data è il segnale che arriva dalla memoria in seguito ad una richiesta di lettura;

e cinque segnali output:

o_address è il segnale che manda l'indirizzo alla memoria;

o_done è il segnale che comunica la fine dell'elaborazione e il dato di uscita scritto in memoria;

o_en è il segnale di ENABLE da dover mandare alla memoria per poter comunicarci;

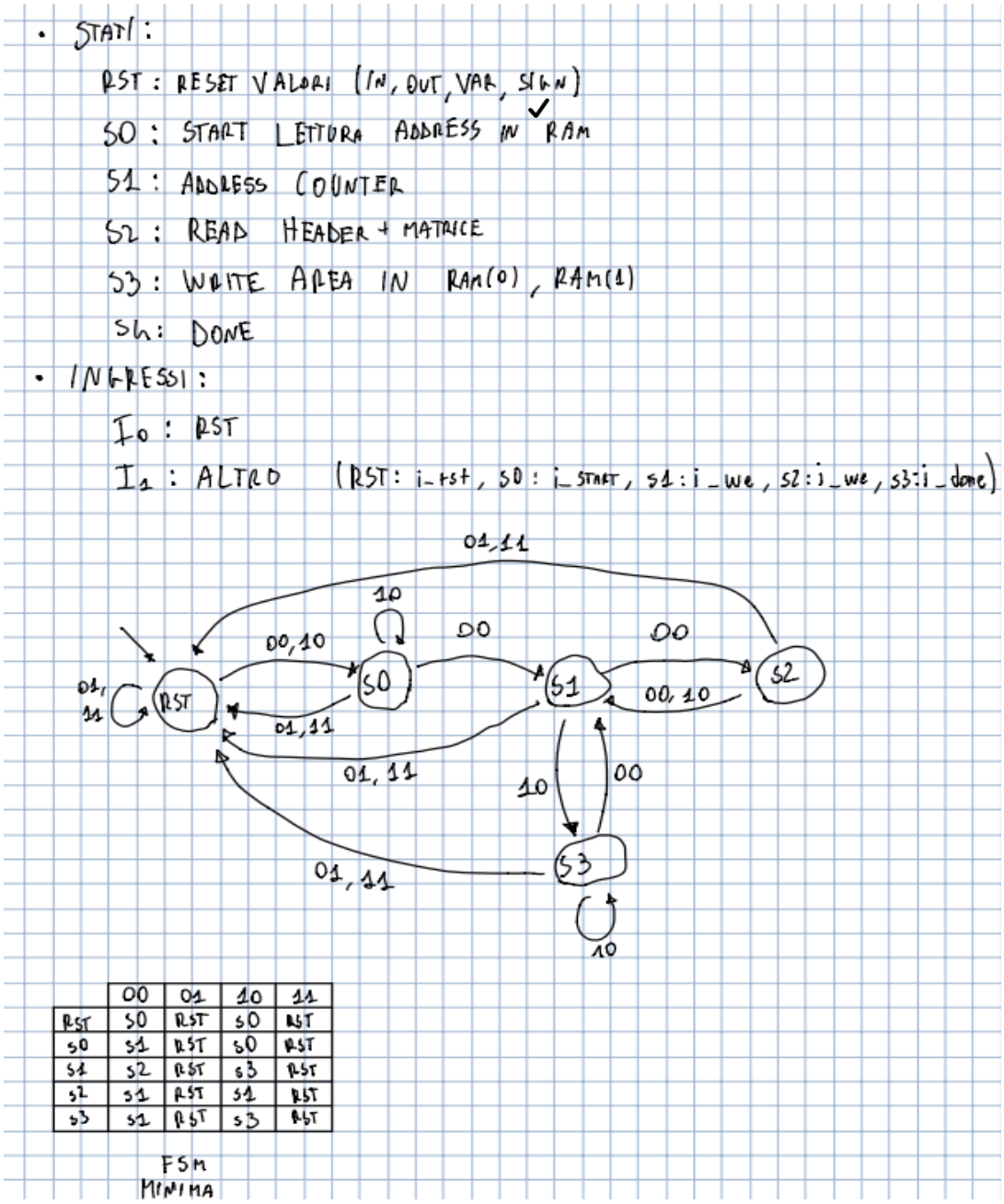
o_we è il segnale di WRITE ENABLE da dover mandare alla memoria (=1) per poter scriverci. Per leggere da memoria esso deve essere 0;

o_data è il segnale di uscita dal componente verso la memoria (utile per la scrittura in memoria).

Idee:

L'idea iniziale era quella di scrivere un singolo processo, per cercare di capire il funzionamento dei testbench e della memoria. Successivamente si è passato a una fase teorica per comprendere come strutturare il componente. Alla fine è stata scelta l'idea di creare una macchina a stati finiti (FSM).

Soluzione iniziale abbiamo pensato a una FSM di questo genere:



	00	01	10	11
RST	S0	RST	S0	RST
S0	S1	RST	S0	RST
S1	S2	RST	S3	RST
S2	S1	RST	S1	RST
S3	S2	RST	S3	RST

FSM MINIMA

L'idea era un po' affrettata e abbastanza abbozzata per dare un assaggio a ciò che doveva essere. Anche se idealmente sembrava abbastanza logico, nella pratica si rischiava di avere tanti ritardi inutili per passare da uno stato all'altro (fino a 3 cicli di clock per fare un'azione che con la nuova FSM è possibile fare in 1).

Soluzione finale:

L'idea finale della macchina a stati finiti è composta dai seguenti stati:

- **IDLE**: stato in cui la macchina non fa assolutamente nulla, come succede anche nei processori dei computer. La scelta è scaturita nel momento in cui, come primo stato, si aveva il Reset, il che non era corretto, in quanto il reset deve avvenire solo quando il segnale 'i_rst' viene impostato a 1.
- **RST**: da come dice il nome, lo stato di reset. La macchina entra in questo stato quando in ingresso si riceve un segnale alto di 'i_rst'. Esso fa tornare il componente allo stato di "fabbrica" (tutti i valori settati a 0).
- **S0**: stato in cui la macchina lavora sull'header dell'immagine in memoria. Vengono salvati temporaneamente n. colonne, n. righe e soglia in tre variabili diverse.
- **S1**: stato in cui la macchina lavora sulla matrice (reading) dell'immagine in memoria. Qui viene applicato l'algoritmo di ricerca dell'area.
- **S2**: stato in cui la macchina effettua i calcoli principali per l'area, tra cui addizione, sottrazione e moltiplicazione.
- **S3**: stato in cui la macchina scrive sulla memoria (writing) i bit meno significativi e i bit più significativi della 'area' del rettangolo che circoscrive la figura di interesse.
- **S4**: stato finale in cui arriva la macchina e serve solo per dare conferma che tutto è avvenuto con successo.

ALGORITMO DI RICERCA

L'algoritmo di ricerca consiste in **tre** variabili:

```
type coordinate_type is array (1 downto 0) of std_logic_vector(7 downto 0);
variable pixel_counter : coordinate_type := (others => "00000001");
variable top_left_pixel : coordinate_type := (others => "00000000");
variable bottom_right_pixel : coordinate_type := (others => "00000000");
```

Per questione di ordine è stato scelto di usare un array di dimensione 2, di 8 bit ciascuno.

- **pixel_counter** : sostanzialmente è un punto che si muove sulla matrice in ordine, partendo dalla posizione (1,1). Possiamo dedurre quindi che, a causa di ciò, la complessità temporale è almeno **O(n.ROW*n.COLUMN)**.
- **top_left_pixel:** è il vertice del rettangolo risultante posizionato in alto a sinistra della figura di interesse.
- **bottom_right_pixel**: è il vertice del rettangolo risultante posizionato in basso a destra della figura di interesse.

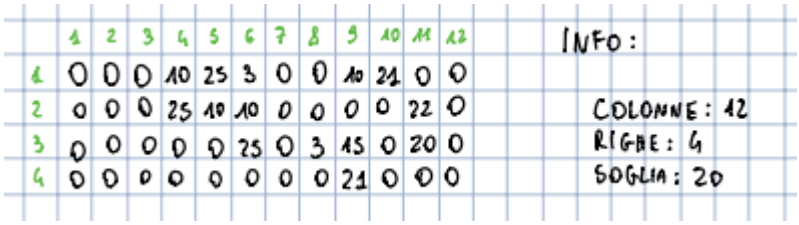
In più si è voluto utilizzare un enum di due elementi:

```
type step_type is (STEP_ONE, STEP_TWO);
variable stepCnt : step_type := STEP_ONE;
```

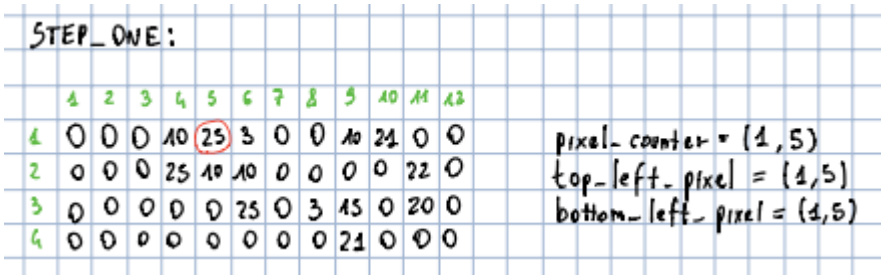
Che è considerabile pari a un booleano. Esso serve solamente per differenziare due step importanti. Il primo quello della definizione del punto iniziale, il secondo è il resto dell'algoritmo fino alla fine del suo percorso.
Il modo in cui sono inizializzate le due variabili di due vertici del rettangolo hanno un loro scopo. Nelle spiegazioni successive verrà reso meglio noto il perché di questa scelta.

Funzionamento:

Come esempio funzionale, per capire il modo in cui lavoro l'algoritmo, prendiamo come esempio codesta matrice, con i corrispettivi dati già forniti:

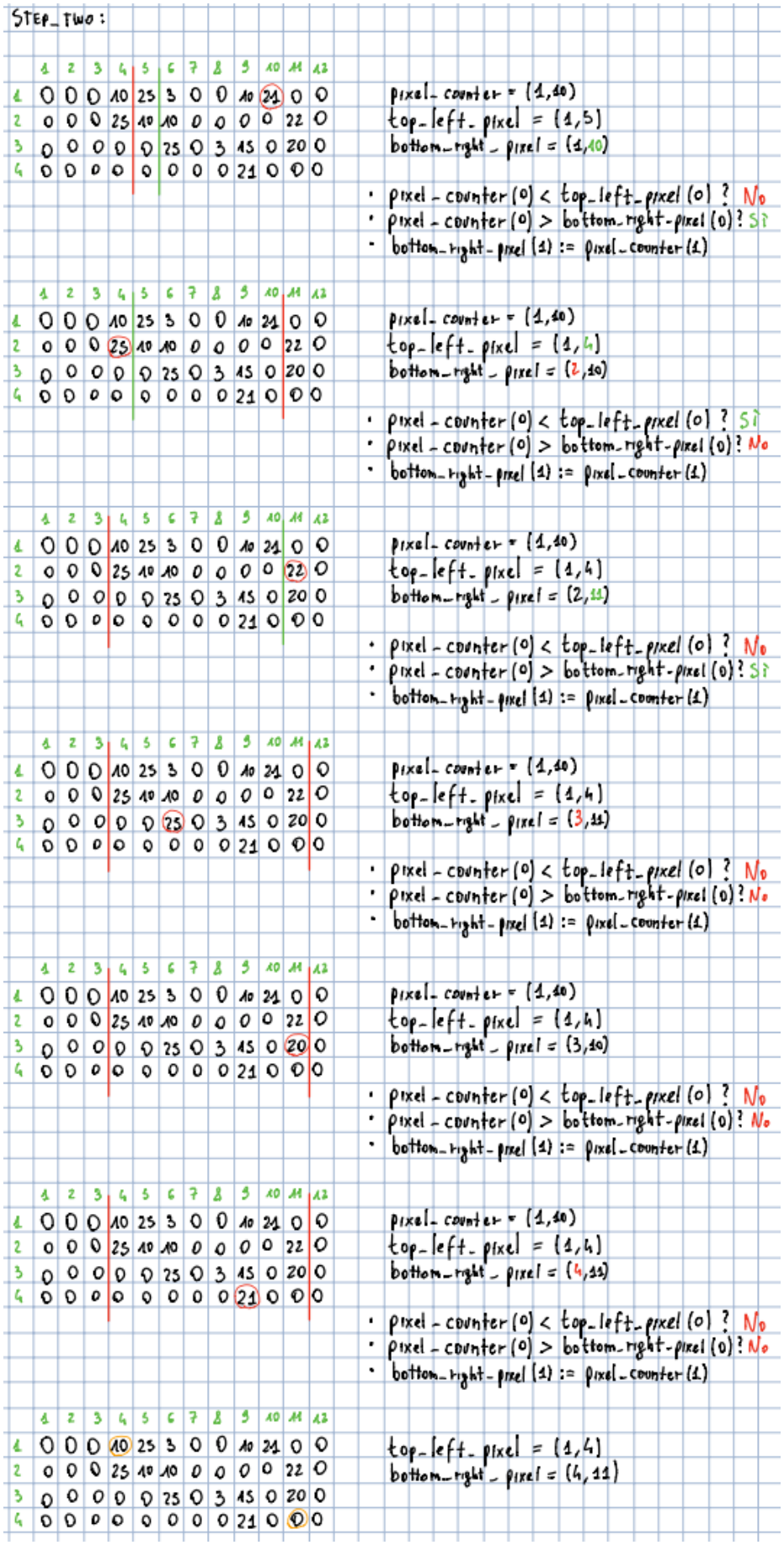


L'algoritmo scandisce pixel per pixel tramite la variabile precedentemente dichiarata, **pixel_counter**, che tiene conto della posizione (posX, posY) in modo che, non appena arrivati a (n.ROW, n.COLUMN), il contatore si ferma e si passa allo stato successivo, quello del calcolo dell'area.
Appena inizia il conteggio c'è un check, pixel per pixel, che controlla se esso supera la soglia. Nel caso sia vero, ci sono degli ulteriori controlli che verificano se il pixel **top_left_pixel** ha la colonna maggiore rispetto al pixel_counter, oppure se **bottom_right_pixel** ha la colonna minore rispetto al pixel_counter. Nel caso uno dei due è vero, si fa un update al valore della colonna del corrispettivo. E viene poi aggiornata la riga del **bottom_right_pixel**.
Prima di ciò, avviene uno "step one" che assegna il valore ad entrambi i pixel del primo pixel sopra la soglia:

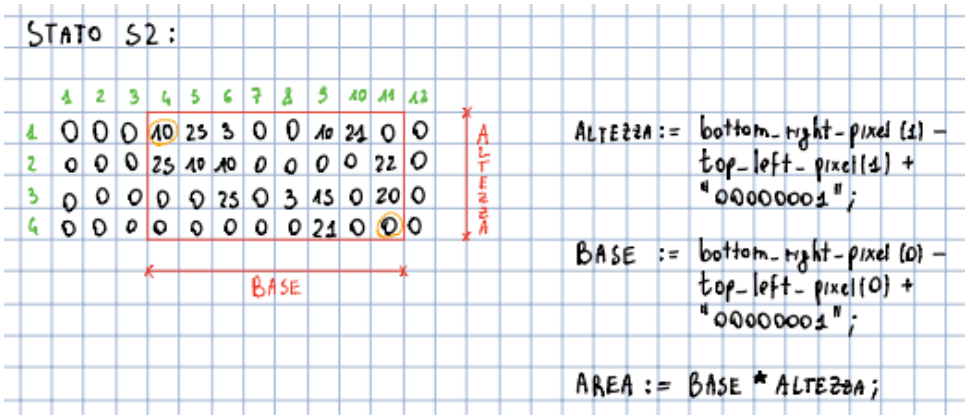


Dopo di ché avviene lo "step two" che lavorerà in una maniera differente:
La riga di **top_left_pixel** rimarrà invariata, quella è la massima altezza dell'area. La colonna però può subire variazioni in base al **pixel_counter** se trova PRIMA rispetto alla colonna di **top_left_pixel**.
Nel caso di **bottom_right_pixel**, lui riceverà variazioni continue sia della riga che della colonna. L'algoritmo per la colonna è a specchio rispetto a quello del **top_left_pixel**, e cioè che se il **pixel_counter** si trova DOPO rispetto alla colonna di **bottom_right_pixel**, allora il valore vien aggiornato, mentre la riga viene aggiornata se il valore è sopra la soglia.

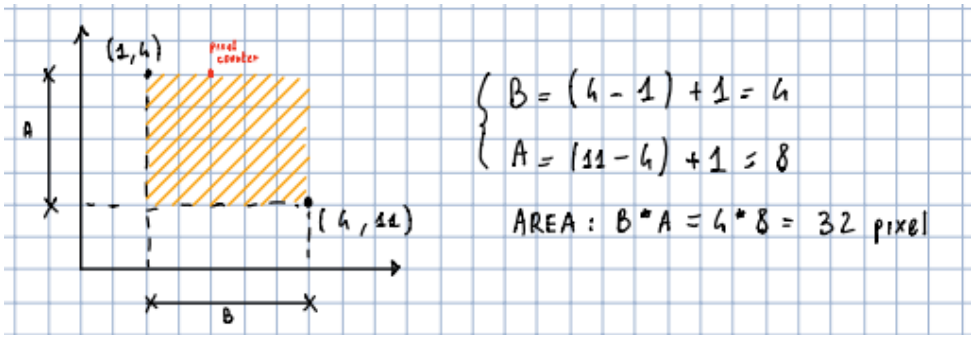
Ecco un esempio completo dello “step two”:



Arrivati all'ultima riga e ultima colonna, il conteggio finisce e si segnala che lo stato attuale ha finito il suo lavoro e che quindi si è pronti a passare, nel prossimo ciclo di clock, nello stato successivo, in questo caso lo stato del calcolo dell'area, che in questo caso fa ciò:

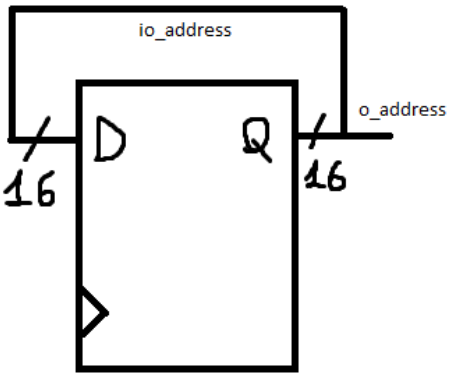


Che poi a sua volta segnalerà di aver concluso il suo lavoro, e il prossimo stato sarà quello di scrittura sul primo e secondo byte della RAM.



UTILIZZO POST-SINTESI

Segnale io_address: La scelta che è stata fatta per ovviare problemi con i ritardi in memoria (siccome l'i/o non è istaneaneo) è stato utilizzato un segnale che idealmente doveva essere un autoanello di questo tipo:



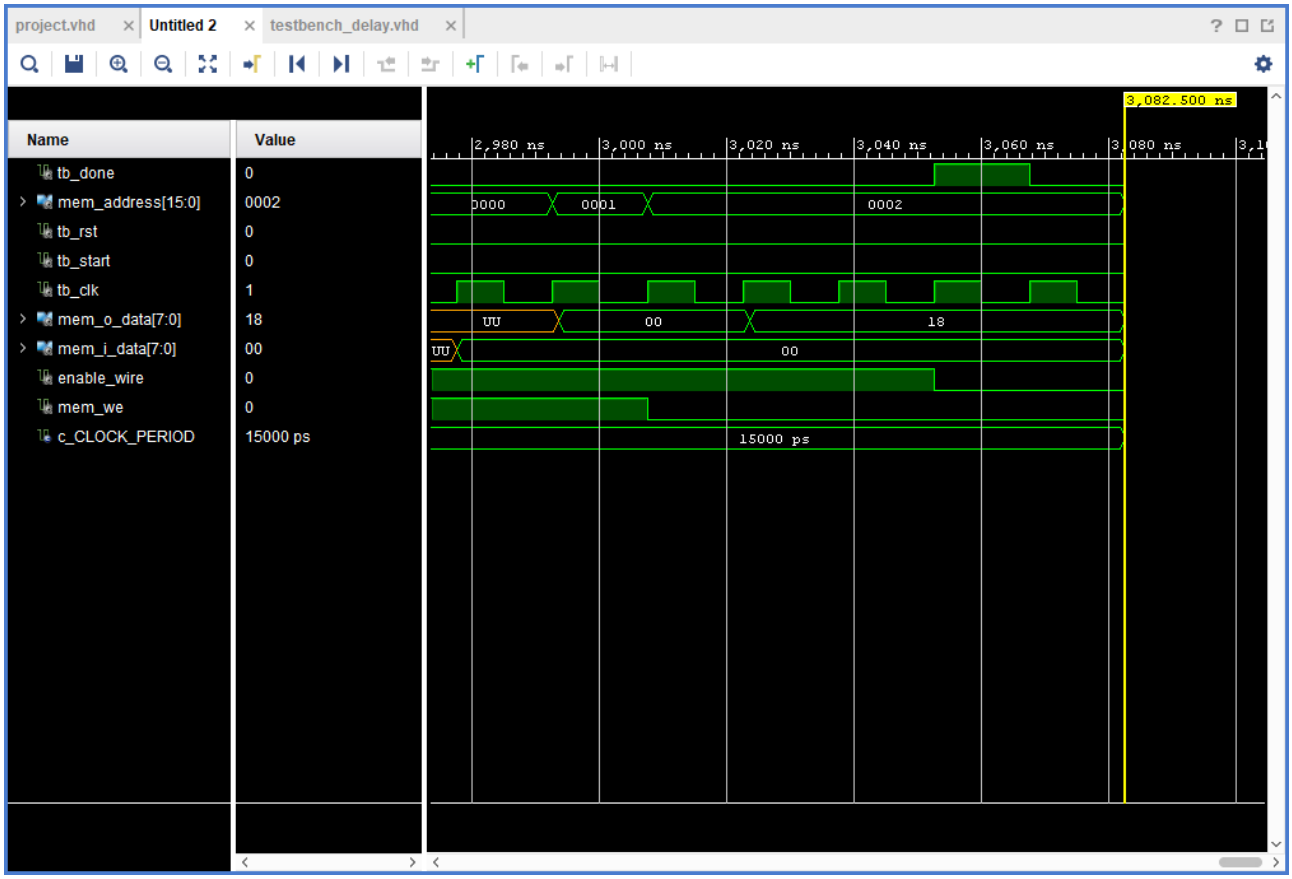
Ma probabilmente, non avendo mappato l'entity come componente, lo vede come un segnale ricevuto dall'esterno, avendo 16 Flip Flop in più.
Utilizzo di variabili: Le variabili che sono state utilizzate nell'intermezzo non danno problemi nella parte finale dello svolgimento. In post-sintesi, più di una variabile, probabilmente è stata riutilizzata, risparmiando qualche flip flop.
Segnali di input: Inizialmente si voleva utilizzare un solo segnale che veniva alzato per un clock e poi riportato a zero, in modo da risparmiare qualche flip flop (invece di utilizzarne tanti quanti sono gli stati). Si verificava però un problema inusuale con i ritardi in memoria e quindi per evitarli si è ritornati all'usare tanti segnali di input per il cambio di stato.
Task multiple: Nel momento in cui si è notato che una task si divideva in più parti ed era difficile organizzarla per un solo stato, si è deciso di dividerla in più stati, rendendo più maneggevole e modulare il codice (anche se alla fine si è utilizzato un solo componente).
Macchina a stati finiti: Basandosi sulla documentazione di 'VHDL', la macchina a stati finiti è divenuta un'architettura con tre processi dove uno regola lo stato corrente e lo stato prossimo, un altro regola lo stato prossimo basandosi sull'input ricevuto, e l'ultimo svolge la funzione degli stati. Essi hanno al loro interno un rising_edge(i_clk) iniziale perché tutti gli stati devono sincronizzarsi sul clock, in modo che il parallelismo sia sincronizzato, tranne il reset che è asincrono, ma comunque lo stato viene ricevuto al clock successivo.

TESTING

I testbench utilizzati sono in parte quelli forniti dal tutore del progetto di Reti Logiche. Inizialmente sono stati forniti soltanto quattro testbench basati su una memoria RAM istantanea, successivamente altri quattro basati su una RAM un po' più realistica che fornisce un ritardo di 1 ns, quindi per clock superiori a frequenze di 1 GHz non ci dovrebbero essere problemi.
Gli altri testbench sono dei testbench personali che verificano probabili casi limite che però verranno visionati nell'ultimo paragrafo.

SINTESI

I test effettuati tramite i testbench forniti, sia in pre-sintesi (behavioral) che in post-sintesi (functional e timing), hanno avuto successo di esecuzione.



La tempistica che ci mettono tutti i test (sia con memoria istantanea che non) sono di:
3082.5 ns

N.B.: Nel caso della post-sintesi timing, i test utilizzati sono quelli con delay, ovviamente.

TIMING (Post-sintesi | Constraint)

Il test per il timing è stato effettuato con un constraint particolare che permette di impostare il ciclo del clock differentemente da 15 ns:

```
create_clock -period 8 -name i_clk [get_ports i_clk]
```

E il risultato ottenuto, con 8 ns è questo:

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0,257 ns	Worst Hold Slack (WHS):	0,163 ns	Worst Pulse Width Slack (WPWS):	3,500 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	306	Total Number of Endpoints:	306	Total Number of Endpoints:	145

All user specified timing constraints are met.

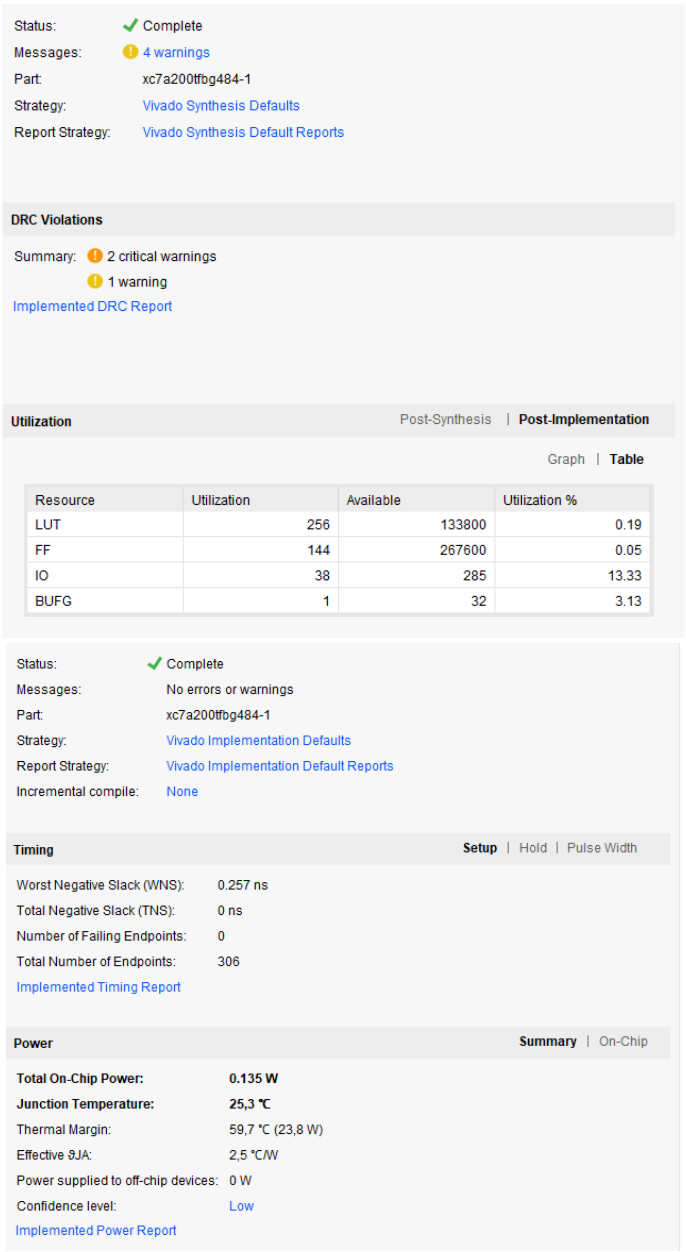
Quindi il massimo che regge è **poco meno di 8 ns** di clock. Non un grande traguardo ma la causa principale è il fatto che la moltiplicazione non è stata implementata ma si è utilizzata

quella del VHDL (a * b). E in più si è utilizzato un solo componente non dichiarato, e cioè l'entity project_reti_logiche. In compenso il numero dei flip flop utilizzati è medio-basso.

N.B.: A causa di un problema verificatosi in matrici alquanto piccole, dopo aver aggiustato il codice, il clock che riesce a reggere non è più poco meno di 7.9 ns ma è un valore compreso tra 7.9 e 8 ns.

PROJECT SUMMARY

Come riepilogo del progetto:



Status: Complete
Messages: 4 warnings
Part: xc7a200tfbg484-1
Strategy: Vivado Synthesis Defaults
Report Strategy: Vivado Synthesis Default Reports

DRC Violations

Summary: 2 critical warnings
1 warning
Implemented DRC Report

Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	256	133800	0.19
FF	144	267600	0.05
IO	38	285	13.33
BUFG	1	32	3.13

Status: Complete
Messages: No errors or warnings
Part: xc7a200tfbg484-1
Strategy: Vivado Implementation Defaults
Report Strategy: Vivado Implementation Default Reports
Incremental compile: None

Timing **Setup** | Hold | Pulse Width

Worst Negative Slack (WNS): 0.257 ns
Total Negative Slack (TNS): 0 ns
Number of Failing Endpoints: 0
Total Number of Endpoints: 306
Implemented Timing Report

Power **Summary** | On-Chip

Total On-Chip Power: 0.135 W
Junction Temperature: 25,3 °C
Thermal Margin: 59,7 °C (23,8 W)
Effective ϑJA: 2,5 °C/W
Power supplied to off-chip devices: 0 W
Confidence level: Low
Implemented Power Report

Il che dimostra principalmente che la post sintesi e l'implementation (col constraint sul clock) sono stati effettuati con successo.

Il numero di **Look Up Table** è 256 (che comunque solitamente variano test per test)
Il numero di **Flip Flop** è 144 (un numero comunque medio-basso, come già detto)
Il numero di **Input/output** è di 38 (potevano essere diminuiti utilizzando components)
Il numero di **Global Buffer** è di 1.

TEST PERSONALI

Sono stati effettuati test personali per verificare alcuni casi limite, modificando i testbench esistenti.

Matrice 255x255 (Overflow):
Il test è stato effettuato con un testbench di questo tipo:
2 => “11111111”, 3 => “11111111”, 4 => “00000000”, others => “00000000”
Inizialmente si è verificato un problema per come è stato scritto l’algoritmo. Il valore della colonna arrivava a “11111111” e dopo aver sommato nuovamente 1 alla variabile il valore della colonna andava a “00000000” senza tener conto dell’overflow. Per tenerne conto è stata aggiunta una variabile ‘overflow’ che ne tiene conto.

Matrice 0x0:
Il test effettuato non ha dato alcun problema, l’unico problema identificato è la tempistica che teoricamente doveva essere ridotta, rispetto a come si presenta.

Reset in un tempo ‘t’:
Il test è stato effettuato aggiungendo ad un testbench le seguenti righe:

```
wait for 1000 ns;
tb_rst <= ‘1’;
wait for c_CLOCK_PERIOD;
tb_rst <= ‘0’;
wait for c_CLOCK_PERIOD;
tb_start <= ‘1’;
wait for c_CLOCK_PERIOD;
tb_start <= ‘0’;
Prima del ‘wait until tb_done = ‘1’ ‘.
```

Inizialmente si ha avuto un problema con gli stati. Sostanzialmente lo stato corrente diventava il RST non appena si riceveva (asincronamente) i_rst = ‘1’, però lo stato successivo, nel clock seguente, prendeva in considerazione lo stato che c’era prima. Ad esempio se lo stato in cui era è S4 e io ho chiamato il reset, rimaneva nello stato S4. Per risolvere questo problema è stato aggiunto un altro if (asincrono) nel secondo processo (DELTA) che assegna allo stato prossimo RST.

Test ciclico:
E’ stato rimosso l’ultimo assert di un qualunque testbench fornito per vedere se il codice fosse in grado di ripetere il tutto senza problemi. Inizialmente il problema è stato che, per un if mancante, modificava anche il terzo address, oltre al primo e secondo, quindi il numero di colonne non era quello corretto, nel secondo ciclo. Dopo aver aggiunto un if il problema è stato risolto.