PROVA FINALE
(PROGETTO RETI LOGICHE)

PROFESSORE: WILLIAM FORNACIARI

POLITECNICO
MILANO 1863

ALESSANDRO BIANCO
SALVATORE BUONO

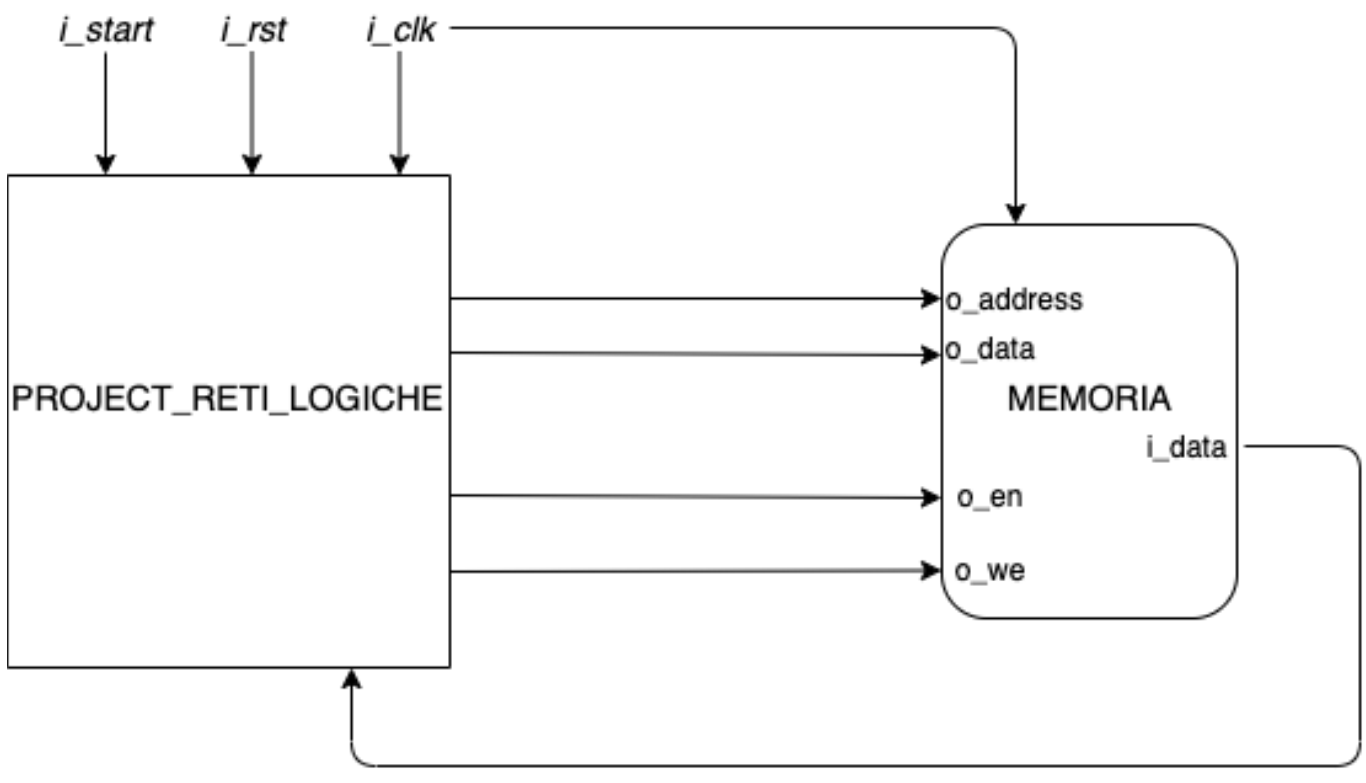
1.INTRODUZIONE

Con la seguente relazione si vuole analizzare la scelta effettuata per la risoluzione al problema proposto dai Docenti del corso di Reti Logiche del Politecnico di Milano nell' AA 2020/2021.
Il progetto è stato svolto in collaborazione tra **Bianco Alessandro [10659523]** e **Buono Salvatore [10600540]**, entrambi studenti iscritti al terzo anno di Ing. Informatica.

Si presenta innanzitutto la specifica del progetto, arricchita di commenti dei Docenti: è richiesto di implementare un modulo hardware (FPGA: xc7a200tfbg484-1),utilizzando VHDL, che permetta di equalizzare i valori dei pixel contenuti in un vettore immagine, spostandosi correttamente tra gli indirizzi contenuti all'interno della memoria, come segue da specifica sottostante. A riguardo è presente la descrizione del componente, modellato come una macchina a stati finiti.

Sono infine analizzati i dati provenienti dall' esecuzione di alcuni banchi di test (tra cui quelli proposti dai Docenti), con relativa descrizione e commento.

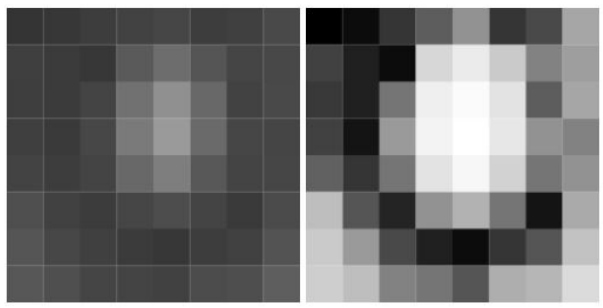
E' qui proposto uno schema di come la nostra entità comunica con la memoria fornita.



Interazione hardware-memoria

2. SPECIFICA

La specifica della Prova finale (Progetto di Reti Logiche) 2020 è ispirata al metodo di equalizzazione dell'istogramma di una immagine.
Il metodo di equalizzazione dell'istogramma di una immagine è un metodo pensato per ricalibrare il contrasto di una immagine quando l'intervallo dei valori di intensità è molto ristretto, effettuandone una distribuzione su tutto l'intervallo di intensità, al fine di incrementare il contrasto.




Una immagine non equalizzata


La stessa immagine dopo l'equalizzazione dell'istogramma

Nella versione da sviluppare non è richiesta l'implementazione dell'algoritmo standard ma di una sua versione semplificata. L'algoritmo di equalizzazione sarà applicato solo a immagini in scala di grigi a 256 livelli e deve trasformare ogni suo pixel nel modo seguente:

DELTA_VALUE = MAX_PIXEL_VALUE – MIN_PIXEL_VALUE
SHIFT_LEVEL = (8 – FLOOR(LOG2(DELTA_VALUE +1)))
TEMP_PIXEL = (CURRENT_PIXEL_VALUE - MIN_PIXEL_VALUE) << SHIFT_LEVEL
NEW_PIXEL_VALUE = MIN(255 , TEMP_PIXEL)

Dove MAX_PIXEL_VALUE e MIN_PIXEL_VALUE , sono il massimo e minimo valore dei pixel dell'immagine, CURRENT_PIXEL_VALUE è il valore del pixel da trasformare, e NEW_PIXEL_VALUE è il valore del nuovo pixel.

Il modulo da implementare dovrà leggere l'immagine da una memoria in cui è memorizzata, sequenzialmente e riga per riga, l'immagine da elaborare. Ogni byte corrisponde a un pixel dell'immagine.
La dimensione della immagine è definita da 2 byte, memorizzati a partire dall'indirizzo 0. Il byte all'indirizzo 0 si riferisce alla dimensione di colonna; il byte nell'indirizzo 1 si riferisce alla dimensione di riga. La dimensione massima dell'immagine è 128x128 pixel.

L'immagine è memorizzata a partire dall'indirizzo 2 e in byte contigui. Quindi il byte all'indirizzo 2 è il primo pixel della prima riga dell'immagine.
L'immagine equalizzata deve essere scritta in memoria immediatamente dopo l'immagine originale.

Le ***dimensioni dell'immagine*** , ciascuna di dimensione di 8 bit, sono memorizzate in una memoria con indirizzamento al Byte partendo dalla posizione 0: il byte in posizione 0 si riferisce al numero di colonne (`N -COL`), il byte in posizione 1 si riferisce al numero di righe (`N-RIG`).
I ***pixel dell'immagine*** , ciascuno di un 8 bit, sono memorizzati in memoria con indirizzamento al Byte partendo dalla posizione 2.
I ***pixel della immagine equalizzata*** , ciascuno di un 8 bit, sono memorizzati in memoria con indirizzamento al Byte partendo dalla posizione `2+(N-COL*N-RIG)` .

Note ulteriori sulla specifica

1. Si noti che nel modulo da implementare, FLOOR(LOG2(DELTA_VALUE +1)) è un numero intero con valori tra 0 e 8 facilmente ricavabile da controlli a soglia.
2. Si faccia attenzione al numero di bit necessari in ogni passaggio.
3. Il modulo deve essere progettato per poter codificare più immagini, ma l'immagine da codificare non verrà mai cambiata all'interno della stessa esecuzione, ossia prima che il modulo abbia segnalato il completamento tramite il segnale DONE. Si veda il prossimo punto per il protocollo di re-start.

4. Il modulo partirà nella elaborazione quando un segnale START in ingresso verrà portato a 1. Il segnale di START rimarrà alto fino a che il segnale di DONE non verrà portato alto; al termine della computazione (e una volta scritto il risultato in memoria), il modulo da progettare deve alzare (portare a 1) il segnale DONE che notifica la fine dell'elaborazione. Il segnale DONE deve rimanere alto fino a che il segnale di START non è riportato a 0. Un nuovo segnale start non può essere dato fintanto che DONE non è stato riportato a zero. Se a questo punto viene rialzato il segnale di START, il modulo dovrà ripartire con la fase di codifica.

5. Il modulo deve essere progettato considerando che prima della prima codifica verrà sempre dato il reset al modulo. Invece, come descritto nel protocollo precedente, una seconda elaborazione non dovrà attendere il reset del modulo.

MEMORIA

o_address	i_data	
0	**3**	numero di colonne
1	**1**	numero di righe
2	**183**	valore primo pixel
3	**76**	valore secondo pixel
4	**250**	valore terzo pixel
5	**214**	valore primo pixel eqz
6	**0**	valore secondo pixel eqz
7	**255**	valore terzo pixel eqz

Esempio con memoria di dimensione 3

2.1 Interfaccia del Componente

Il componente da descrivere deve avere la seguente interfaccia:

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

In particolare:

- il nome del modulo deve essere project_reti_logiche
- i_clk è il segnale di CLOCK in ingresso generato dal TestBench;
- i_rst è il segnale di RESET che inizializza la macchina pronta per ricevere il primo segnale di START;
- i_start è il segnale di START generato dal Test Bench;
- i_data è il segnale (vettore) che arriva dalla memoria in seguito ad una richiesta di lettura;
- o_address è il segnale (vettore) di uscita che manda l'indirizzo alla memoria;
- o_done è il segnale di uscita 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 comunicare (sia in lettura che in scrittura);
- 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 (vettore) di uscita dal componente verso la memoria.

2.2 Descrizione Memoria

```
-- Single-Port Block RAM Write-First Mode (recommended template) --
-- File: rams_02.vhd
--

library ieee;
use ieee.std_logic_1164.all; use ieee.std_logic_unsigned.all; entity rams_sp_wf is
port(

clk : in std_logic;
we : in std_logic;
en : in std_logic;
addr : in std_logic_vector(15 downto 0); di : in std_logic_vector(7 downto 0); do : out
std_logic_vector(7 downto 0)

);
end rams_sp_wf;

architecture syn of rams_sp_wf is
type ram_type is array (65535 downto 0) of std_logic_vector(7 downto 0); signal RAM :
ram_type;
begin

process(clk) begin

if clk'event and clk = '1' then if en = '1' then

if we = '1' then
RAM(conv_integer(addr)) <= di;
do <= di after 2 ns;

else
do <= RAM(conv_integer(addr)) after 2 ns;

        end if;

      end if;

end if; end process;

end syn;
```

3. ANALISI DELLA FSM

Qui è riportato lo schema dettagliato della FSM con relativa descrizione di tutti gli stati. Nel grafo sono inoltre segnati i segnali che permettono una transizione piuttosto che un'altra. La macchina è composta da venti stati.

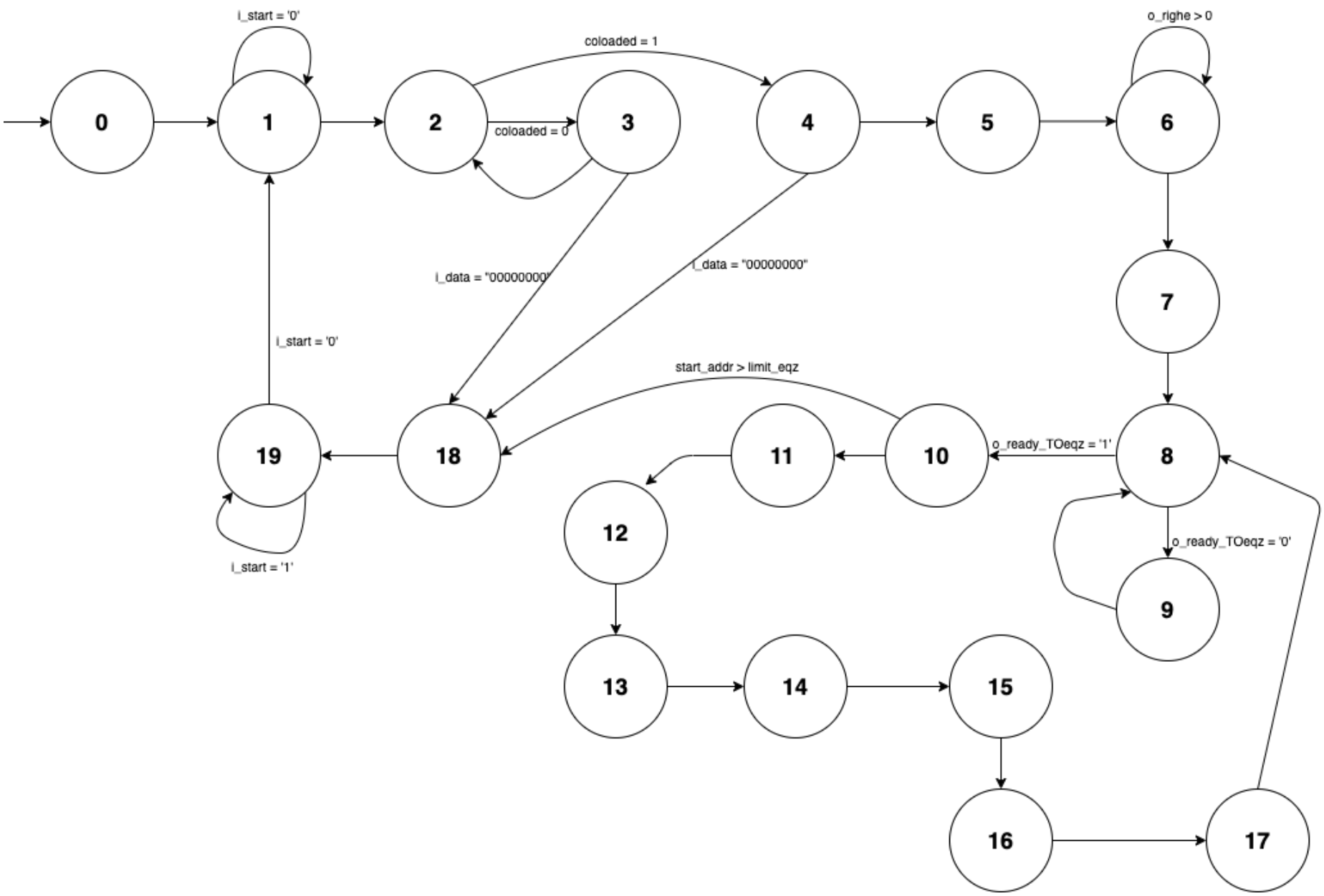


Grafico della FSM

STATI DELLA MACCHINA	NOME STATI	DESCRIZIONE
S0	RESET	Stato di reset della macchina
S1	START	Stato iniziale della macchina
S2	WAIT_STATE	Stato di delay per valori
S3	LOAD_COL	Stato di caricamento numero colonne
S4	LOAD_RIG	Stato di caricamento numero righe
S5	PRE_CALC	Stato di setting per calcolo dimensione
S6	CALC_DIM	Stato di calcolo dimensione immagine
S7	LOAD_FIRST	Stato di caricamento del primo pixel
S8	LOAD_VAL	Stato di delay per caricamento pixels
S9	CALC_MAXMIN	Stato per il calcolo massimo e minimo
S10	START_EQZ	Stato di inizio equalizzazione
S11	TEMPORARY_PX1	Stato di elaborazione temporanea (1)
S12	TEMPORARY_PX2	Stato di elaborazione temporanea (2)
S13	SHIFT_LEVEL	Stato per shiftare il vettore
S14	END_EQZ	Stato di fine equalizzazione pixel
S15	CTRL_PREW	Stato di controllo pre-scrittura
S16	WRITING	Stato di scrittura sulla memoria
S17	NEXT_PX	Stato di predisposizione al prossimo pixel
S18	DONE	Stato dove done è portato alto
S19	DONE_WAIT	Stato prima della codifica di una nuova immagine

3.1 Descrizione FSM

Alla base dell'avanzamento tra gli stati della macchina, ci sono diciannove **segnali ausiliari**; è importante sottolineare come, tra questi, ci siano utili segnali di conteggio (veri e propri contatori) che vengono incrementati in stati significativi del processo di equalizzazione oppure stati di salvataggio informazione : questi ci permettono di tenere traccia di riferimenti quali:

-quanti pixel sono stati già equalizzati (***counter_eqz***)

-a quale indirizzo ci troviamo in lettura (***start_addr***)

-su quale indirizzo dobbiamo scrivere (***addr_out***)

-indirizzo del pixel non ancora equalizzato (***save_addr_NOTelab***)

-ed altri correttamente descritti nel codice *(vedi codice)*

All' inizio della prima computazione viene portato alto il segnale di reset : questo (come gli altri eventuali segnali di reset) è gestito in modo asincrono, come da specifica: ad ogni ciclo di clock la macchina verifica che il segnale *i_rst* sia basso per proseguire e in caso contrario, appunto, si resetta ad S0.

Gli altri diciannove stati sono utilizzati per modellare la logica che permette l'equalizzazione del vettore immagine. Nello specifico :

- AVVIO E CALCOLO DIMENSIONE [S1-S6]

Lo stato S1 cicla su se stesso fintanto che *i_rst* è alto: quando quest'ultimo è portato basso e viene dato il segnale di i_start, la macchina si sposta in uno stato di delay artificiale [S2] che ci predispone alla lettura del numero di colonne e successivamente, tramite il flag ***col_loaded***, a quello di righe.

Vengono infatti letti da memoria e salvati sui segnali ausiliari ***o_col*** [S3] e ***o_righe*** [S4] rispettivamente il numero di colonne e quello di righe che compongono l'immagine da equalizzare. In entrambi gli stati è implementata una logica basata sul fatto che se uno dei due valori è nullo, non c'è niente da equalizzare e si finisce nello stato DONE.

Lo stato S5 basa quindi la sua logica sul fatto che il numero delle righe sia almeno uno: inizia il calcolo della dimensione.
Lo stato S6 cicla su di esso fino a quando il segnale ***o_righe*** non è pari a 0, a questo punto la dimensione è pronta su ***o_dim***.

- CALCOLO MASSIMO E MINIMO [S7-S9]

È facile ipotizzare che il primo pixel sia il massimo e il minimo (temporanei) sui quali iniziare a effettuare dei calcoli! Lo stato S7 si occupa di salvare su segnali ausiliari ***o_massimo*** e ***o_minimo*** il valore contenuto all' indirizzo 2.

Particolare importanza è data allo stato S8 [VAL_LOAD]: ci permette infatti di avere un delay che gestisce la lettura e lo scorrimento della memoria. Ci forniamo del segnale ***value*** per salvare il valore del pixel corrente, che confrontiamo poi in S9 con i valori di massimo e minimo temporanei… Per fermare i calcoli è stato predisposto il segnale ***limit_count:*** quando questo viene raggiunto la macchina porta alto il segnale ***o_ready_TOeqz*** che permette allo stato S8 di spostare la computazione in S10.

- CALCOLO PIXEL EQUALIZZATO [S10-S15]

Vengono correttamente settati i segnali che ci accompagnano in questi stati (quali ***delta_value***, ***start_addr,*** *vedi codice*). In S10 inizia la vera e propria equalizzazione del pixel. Il seguente stato infatti, permette la lettura del valore su *i_data* e salva su ***save_addrr_NOTelab*** l'indirizzo del pixel che non è stato ancora equalizzato.

Ci spostiamo adesso in due stati di lavorazione pixel: S11 ed S12 dove vengono effettuati i passaggi per preparare il valore da shiftare.

Lo stato S13 si occupa di gestire con un case la parte relativa allo shift-level analizzando il contenuto di ***delta_value,*** noi lo abbiamo implementato con una concatenazione di vettore: vengono concatenati tanti zeri quante sono le posizione da shiftare. (Le concatenazioni sono fatte su un vettore ad 8bit trasformato in uno da 16bit).
Il vettore viene poi trasformato nuovamente in intero in S14 e controllato in S15 per assicurarci che non superi 255 (in tal caso scriveremo 255 in memoria).

- SCRITTURA SU MEMORIA E PREPARAZIONE PIXEL SUCC. [S16-S17]

Lo stato S15 ha predisposto il valore da scrivere su *o_data*: nello stato S16 [WRITING] è portato alto il segnale *o_we* che ci permette di scrive in memoria.

Il valore equalizzato è quindi stato scritto sull'indirizzo *o_address* e la macchina è pronta a elaborare un nuovo pixel: S17 predispone infatti l'indirizzo del nuovo pixel da equalizzare (salvato precedentemente), così da continuare l' equalizzazione dove ci eravamo fermati: si torna a S10 per equalizzare un atro pixel.

- DONE E NUOVA IMMAGINE [S18-S19]

Dallo stato S10, quando tutti i pixel saranno equalizzati, passeremo nello stato S18 [DONE] il quale porterà alto il segnale *o_done* : l' immagine è stata correttamente equalizzata! Ci spostiamo quindi in S19 dove aspettiamo che *i_start* sia portato in basso per riposizionarsi nello stato S1 e aspettare un nuovo segnale di *i_start* in modo da partire con un' ulteriore immagine da equalizzare.

4. RISULTATI SPERIMENTALI

4.1 RISULTATI DI SINTESI

La macchina è composta da 241 LUT e 321 FlipFlop (priva di latch).

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	241	0	134600	0.18
LUT as Logic	241	0	134600	0.18
LUT as Memory	0	0	46200	0.00
Slice Registers	321	0	269200	0.12
Register as Flip Flop	321	0	269200	0.12
Register as Latch	0	0	269200	0.00
F7 Muxes	0	0	67300	0.00
F8 Muxes	0	0	33650	0.00

report_utlization da TCL Console

Il Worst Negative Slack è stato poi calcolato utilizzando i tools di sintesi offerti da Vivado:

ne ricaviamo quindi che il periodo di clock massimo necessario alla macchina è:

$$T_{CK} = 100\ ns + \tau_{RAM} - WNS = 9.528\ ns$$

(Calcolato fatto considerando un ritardo di interazione con la memoria di 2ns)

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	92,472 ns	Worst Hold Slack (WHS):	0,164 ns	Worst Pulse Width Slack (WPWS):	4,500 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:	850	Total Number of Endpoints:	850	Total Number of Endpoints:	322

All user specified timing constraints are met.

Report Timing Summary (implementation post-synthesis)

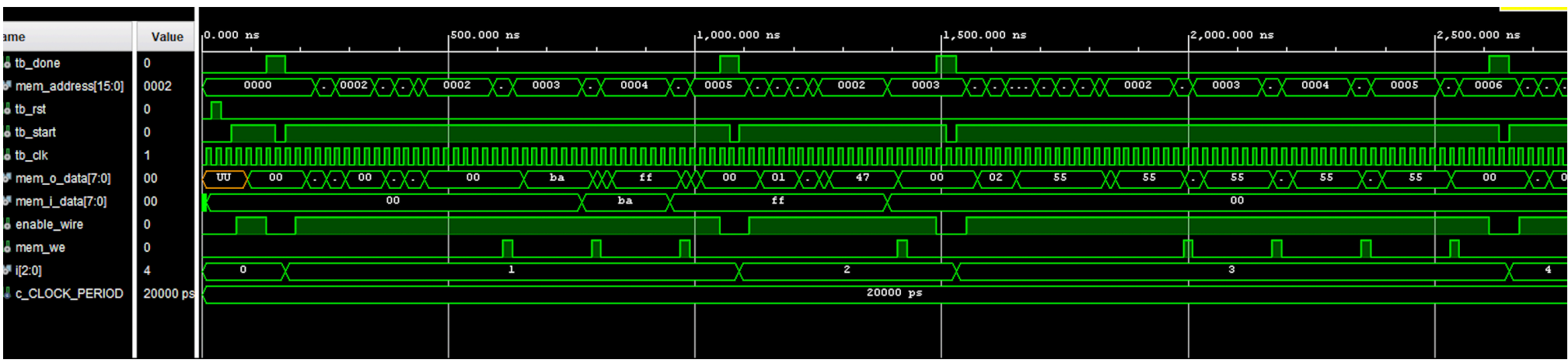
4.2 RISULTATI DEI TESTBENCH

Nella seguente sezione si vogliono esaminare i comportamenti della macchina sottoposta ai casi limite e ad alcuni “stress-test”.
Si vuole sottolineare che tutti i test che sono stati effettuati superano la Behavioral-Simulation ed entrambe le Post-Synthesis-Functional-Simulation correttamente.

- Per testare i **casi limite** abbiamo sfruttato il testbench di prova fornito dai Docenti modificando questo file nei seguenti modi:

1) tb ufficiale dei Docenti *(esempio con immagine 2x2)*
2) Tb con colonne immagine nulle
3) Tb con delta_value massimo *(255)*
4) Tb con dimensione immagine pari a 1.
5) Tb con delta_value nullo
6) Tb con righe immagine nulle

Una volta testati singolarmente, questi casi sono stati eseguiti tutti insieme: abbiamo infatti provveduto a creare dei generatori randomici in Python e in C che ci hanno permesso di testare più immagini (quindi più casi) con un solo test.
Di seguito il report dell’ equalizzazione di 5 immagini consecutive (contiene infatti i casi limite 2-6) senza segnale di reset dopo l’elaborazione di ogni immagine.



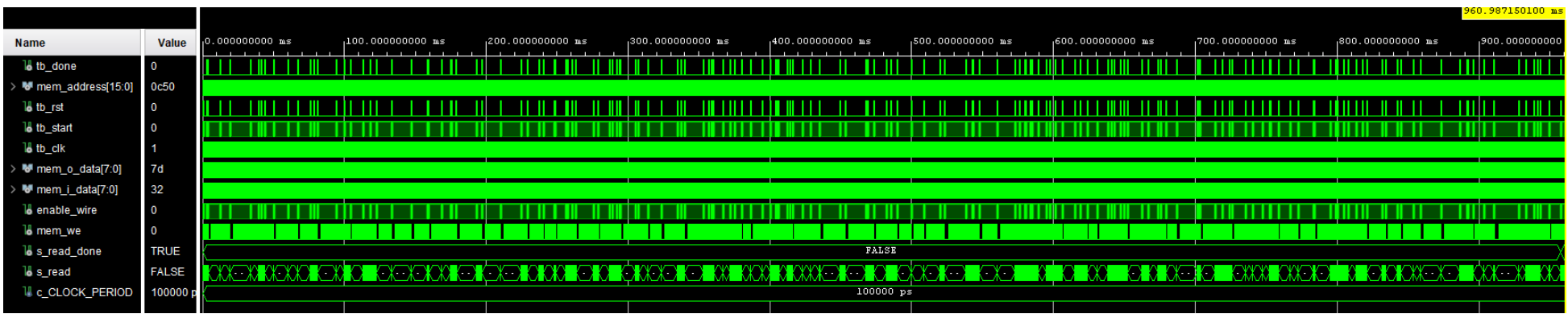
Post-Synthesis_simulation con casi limite eseguiti consecutivamente [clock_period 20ns]

Failure: Simulation Ended! Test Passed!
Simulation-time: 2810 ns

- Abbiamo inoltre sottoposto la nostra macchina ad alcune simulazioni più invasive: sono state simulate infatti sia l'equalizzazione consecutiva di 100 immagini (senza reset) che quella di 200 immagini (con reset dopo ogni immagine elaborata). Essendo infatti generati in modo automatico con dimensioni massima 128x128, questi test ci hanno permesso di valutare positivamente la nostra macchina in termini di robustezza (tante immagini) e casi svariati di combinazioni (si presuppone che vengano coperti tutti i casi di delta_value possibili!).

Di seguito i report di entrambi i test (il primo in Behavioral-Simulation e il secondo in Post_Synthesis-Functional):

Report 100 immagini (senza reset)



Report 200 immagini con reset dopo ogni equalizzazione

Failure: Simulation Ended! Test Passed!
Simulation-time: 960987150100 ps

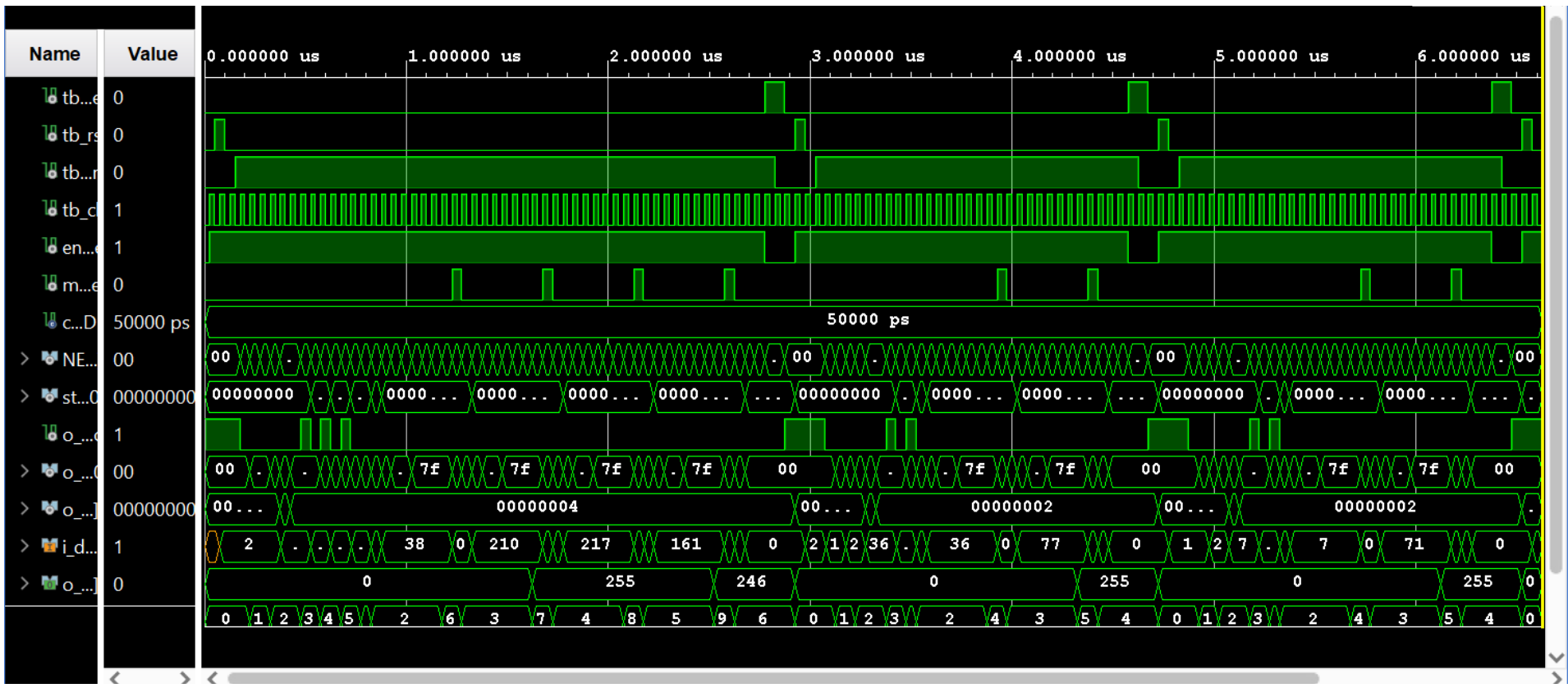
- Sono stati infine testati i casi di reset ottenendo risultati positivi per tutte e tre le casistiche esaminate:

-reset post-elaborazione dopo ogni immagine

-reset durante elaborazione con annullamento

-abbassamento di i_start con annullamento

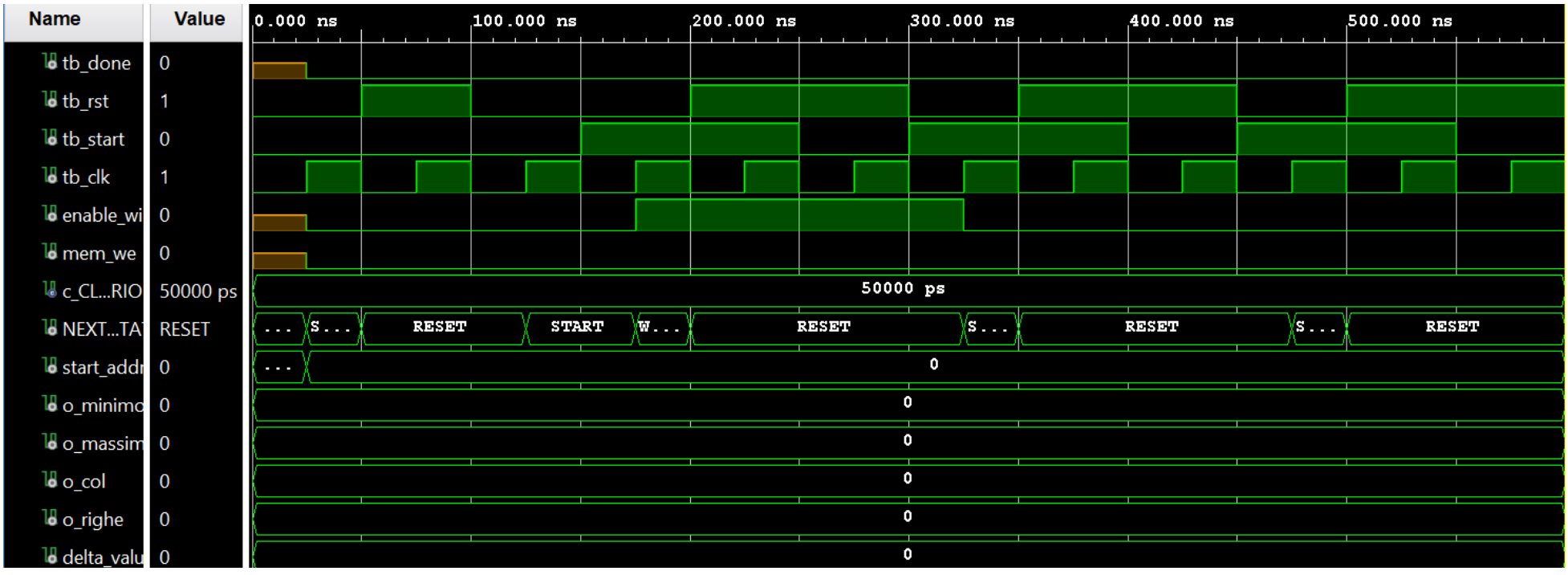
Di seguito i report in Post-Synthesis-Simulation di tutti e tre i casi:



Report reset post-elaborazione

Notiamo come il segnale i_rst è portato in alto alla fine di ogni singola elaborazione. La macchina viene quindi resettata prima di elaborare una nuova immagine (come si puo' leggere dalle forme d' onda, NEXT_STATE è 00).

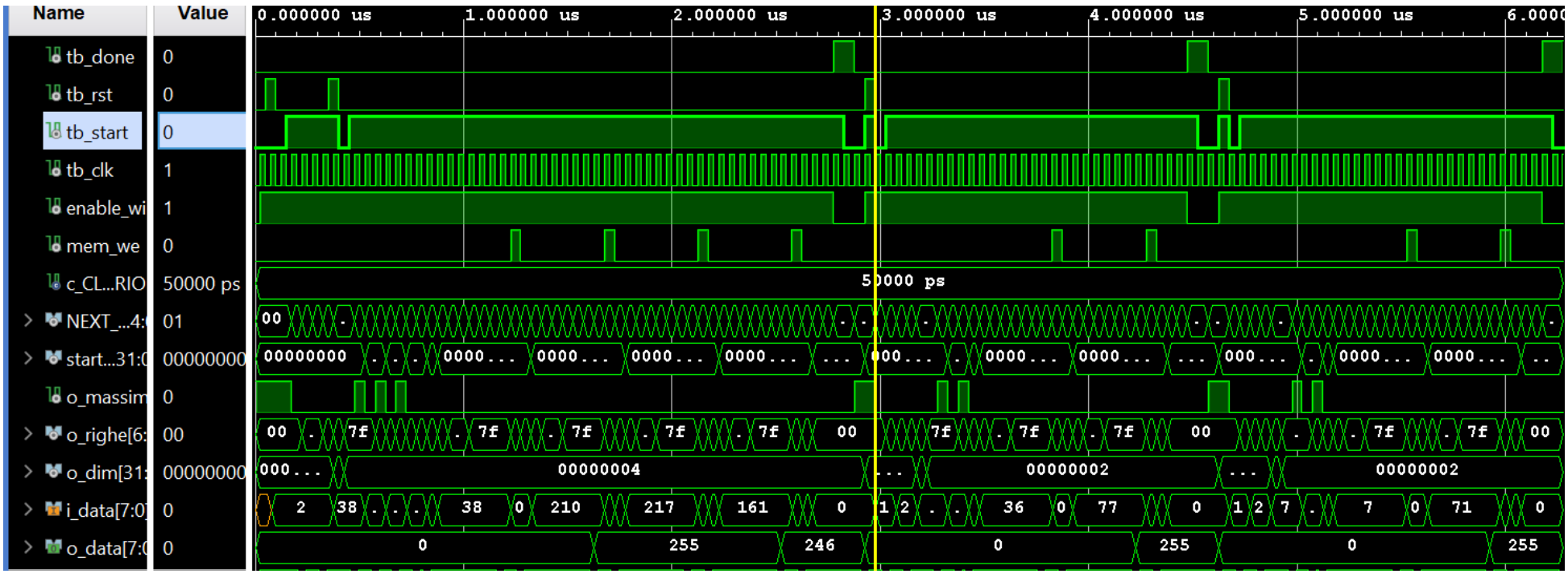
Failure: Simulation Ended! Test Passed!
Simulation-time: 6625,100 ns



Report reset asincrono

Notiamo come il segnale i_rst è portato in alto dopo che la macchina è già nello stato S2. In tutti i casi, come si intuisce facilmente dal segnale dello stato, si torna in RESET e si inizia una nuova computazione!

Failure: Simulation Ended! Test Passed!
Simulation-time: 600 us



Report abbassamento i_start

La macchina rimane nello stato di START finché il segnale di i_start non torna alto.

Failure: Simulation Ended! Test Passed!
Simulation-time: 6301 ns

5. CONCLUSIONI

Il lavoro è iniziato principalmente su carta, attraverso alcuni accenni di datapath e poi della macchina a stati : così facendo abbiamo capito quali/quanti segnali erano necessari per elaborare l'informazione e che tipo di elementi architettura dovevamo implementare.
Quando ci siamo trovati di fronte al codice, tramite alcune ricerche, abbiamo familiarizzato con il linguaggio e abbiamo pensato che l'alternativa di scrivere la macchina con unico process è più che valida, in quanto :

-si suddivide il carico delle operazioni per gli interi stati della macchina

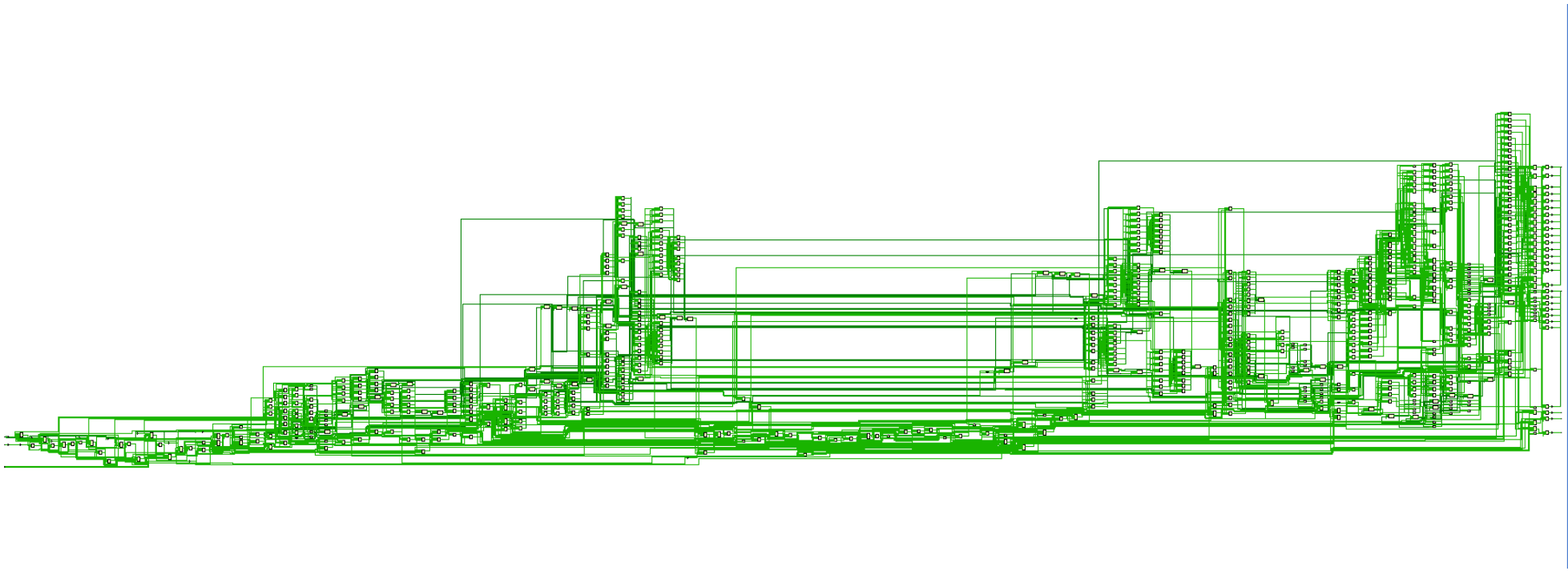
-il clock rispetta largamente la specifica, utilizziamo solo il 9,5% della specifica

-la macchina si resetta correttamente come da specifica

-le dimensioni occupate dal modulo sono irrisorie rispetto alla FPGA fornita

-ci è sembrata la scelta più valida tra quantità di codice/efficienza della macchina

Una volta scritto il codice siamo passati alla fase di debugging, dopo la quale abbiamo iniziato a testare il codice come sopra riportato: il corretto debugging ci ha permesso di ottenere una macchina fin da subito funzionante in post-synthesis!
Viene riportato il DataPath della macchina:



Report Utilization Summary (post-synthesis implementation)