POLITECNICO
MILANO 1863

Prova Finale – Reti Logiche
A.A. 2018/2019

Facoltà di Ingegneria Industriale e dell'Informazione
Corso di laurea in Ingegneria Informatica

Documentazione relativa al progetto svolto dagli studenti Daniele Nicolò e Gioele Mombelli.

Indice

La realizzazione del progetto qui presentato è stata svolta in autonomia come Prova Finale in conclusione del corso di Reti Logiche, A.A. 2018/2019, tenutosi presso il Politecnico di Milano (Facoltà di Ingegneria Industriale e dell'Informazione, corso di laurea in Ingegneria Informatica).

Daniele Nicolò – COD. PERSONA 10527191 – MATRICOLA 846897
daniele.nicolo@mail.polimi.it
Gioele Mombelli – COD. PERSONA 10520552 – MATRICOLA 845899
gioele.mombelli@mail.polimi.it

I – SPECIFICHE DI PROGETTO

La prova finale ha previsto lo sviluppo in VHDL di un componente hardware opportunamente specificato dalla documentazione fornita dal personale docente. Il software utilizzato è stato Xilinx Vivado 2018.3, e l'FPGA target è Artix-7 xc7a200tfbg484-1.
Si riporta di seguito un riassunto delle specifiche di progetto, con l'obiettivo di spiegare in modo chiaro il problema risolto dal componente, le convenzioni utilizzate nello spazio di indirizzamento della memoria, e, in generale, di fornire tutti i prerequisiti necessari per comprendere il funzionamento della macchina e le scelte progettuali effettuate.

I.1 - Descrizione generale del problema

È dato uno spazio bidimensionale definito come una matrice quadrata di dimensioni 256x256. In questo spazio sono collocati 8 punti, denominati "centroidi", ciascuno identificato da una coppia di coordinate (denominate "x" e "y") che ne determinano la posizione all'interno della matrice.
Vi è inoltre collocato un ulteriore punto, da qui in poi denominato "osservatore", anch'esso identificato da una coppia di coordinate. Il componente hardware da progettare ha lo scopo di calcolare quali centroidi risultino più vicini all'osservatore, considerando come distanza la distanza di Manhattan (Mahattan distance), così definita:

```
MhD = (|x-x0| + |y-y0|)
```

Degli 8 centroidi contenuti nello spazio, solamente alcuni vanno presi in considerazione per l'analisi della distanza, mentre altri devono essere ignorati. Ciò viene espresso mediante un valore denominato "maschera d'ingresso" ad 8 bit. Ogni bit corrisponde ad un centroide: nel caso in cui esso valga 1, il relativo centroide va considerato per l'analisi della distanza, mentre deve essere ignorato nel caso in cui tale bit valga 0. Per esempio, la seguente maschera di ingresso indica che soltanto il primo, il terzo e il settimo centroide devono essere considerati nell'analisi della distanza:

```
01000101
```

Il risultato della computazione è anch'esso un valore ad 8 bit, denominato "maschera di uscita": ogni bit avrà valore 1 se e solo se il corrispondente centroide è stato determinato essere il più vicino all'osservatore. Se più centroidi si dovessero trovare alla medesima distanza minima dall'osservatore, tutti i bit corrispondenti dovranno valere "1". Per esempio, se l'analisi ha decretato che i centroidi più vicini sono il primo e il settimo, la maschera di uscita è la seguente:

```
01000001
```

I.2 - Descrizione generale della memoria

Le coordinate dei centroidi, dell'osservatore, e i dati relativi alle maschere d'ingresso e di uscita sono memorizzati in una memoria RAM preesistente con cui il componente hardware deve interfacciarsi. Viene di seguito fornita una breve descrizione dello spazio di indirizzamento della memoria RAM:

INDIRIZZO	***DATO CONTENUTO***
0	*Maschera d'ingresso*
1	*X primo centroide*
2	*Y primo centroide*
3	*X secondo centroide*
4	*Y secondo centroide*
...	...
17	*X osservatore*
18	*Y osservatore*
19	*Maschera d'uscita*

La memoria è indirizzata al byte, e ogni elemento ha dimensione 8 bit. I centroidi vengono elencati dal primo (bit meno significativo della maschera d'ingresso) all'ottavo (bit più significativo della maschera d'ingresso), ponendo prima la coordinata X e poi la coordinata Y.

Si prega di notare che le coordinate dei centroidi si trovano ai seguenti indirizzi:

X: `(i*2) + 1` *(Con: 0<= i <= 7, per comprendere ognuno degli 8 centroidi).*
Y: `(i*2) + 2`

L'indirizzo 19, alla fine dell'elaborazione, dovrà contenere la maschera d'uscita corretta.

I.3 - Descrizione della entity del componente

La entity del componente da sviluppare è stata fornita dal personale docente. Ne si riporta qui il codice VHDL con una breve descrizione dei segnali in essa presenti.

```
entity project_reti_logiche is
port (
        i_clk         : in  std_logic;  -- CLOCK signal
        i_start       : in  std_logic;  -- START signal
        i_rst         : in  std_logic;  -- RESET signal
        o_done        : out std_logic;  -- DONE  signal

        i_data        : in  std_logic_vector(7 downto 0);   --[RAM]: input
        o_address     : out std_logic_vector(15 downto 0);  --[RAM]: output
        o_en          : out std_logic;                      --[RAM]: output
        o_we          : out std_logic;                      --[RAM]: output
        o_data        : out std_logic_vector (7 downto 0)   --[RAM]: output
    );

end project_reti_logiche;
```

i_clk: input

Segnale di CLOCK che regola il funzionamento del componente. Da specifica, il periodo di clock non deve essere superiore ai 100 ns: il componente deve quindi operare ad una frequenza minima di 10 Mhz.

i_start: input

Segnale di START che avvia l'elaborazione quando viene portato ad '1'. L'elaborazione inizia solamente quando un segnale di START viene ricevuto, e in nessun altro caso. START manterrà valore '1' finché l'elaborazione non sarà terminata, ovvero quando anche il segnale di DONE non verrà portato ad '1'.

i_rst: input

Segnale di RESET. Quando viene ricevuto, il componente si pone in uno stato noto, in attesa dell'arrivo di un segnale di start.

o_done: output

Segnale di DONE. Viene portato ad '1' a conclusione dell'elaborazione. Questo segnale deve rimanere al valore logico alto finché il segnale di start non viene riportato a '0'. Un nuovo segnale di START non può essere ricevuto finché il segnale di DONE non viene riportato a '0'.

i_data: input [8 bit]

Segnale in ingresso nel componente proveniente dalla memoria RAM. Contiene il dato che viene inviato dalla memoria in seguito ad una richiesta di lettura.

o_address: output [16 bit]

Segnale in uscita dal componente verso la memoria RAM. Contiene l'indirizzo al quale eseguire l'operazione di lettura o di scrittura.

o_en: output

Segnale in uscita dal componente verso la memoria RAM. Quando portato al valore logico '1', abilita la RAM (ENABLE) consentendo l'operazione di lettura o scrittura.

o_we: output

Segnale in uscita dal componente verso la memoria RAM. Quando vale '0', la RAM esegue un'operazione di lettura. Quando vale '1', la RAM esegue un'operazione di scrittura (WRITE ENABLE).

o_data: output [8 bit]

Segnale in uscita dal componente verso la RAM. Contiene il dato che deve essere scritto nel caso l'operazione effettuata sia una scrittura.

II – SCELTE PROGETTUALI

II.1– ALGORITMO

Il primo passo fondamentale nella progettazione del componente è stata la ricerca di un algoritmo che permettesse di risolvere in modo efficace il problema. La soluzione individuata viene di seguito esposta mediante pseudocodice accompagnato da una spiegazione in linguaggio naturale.

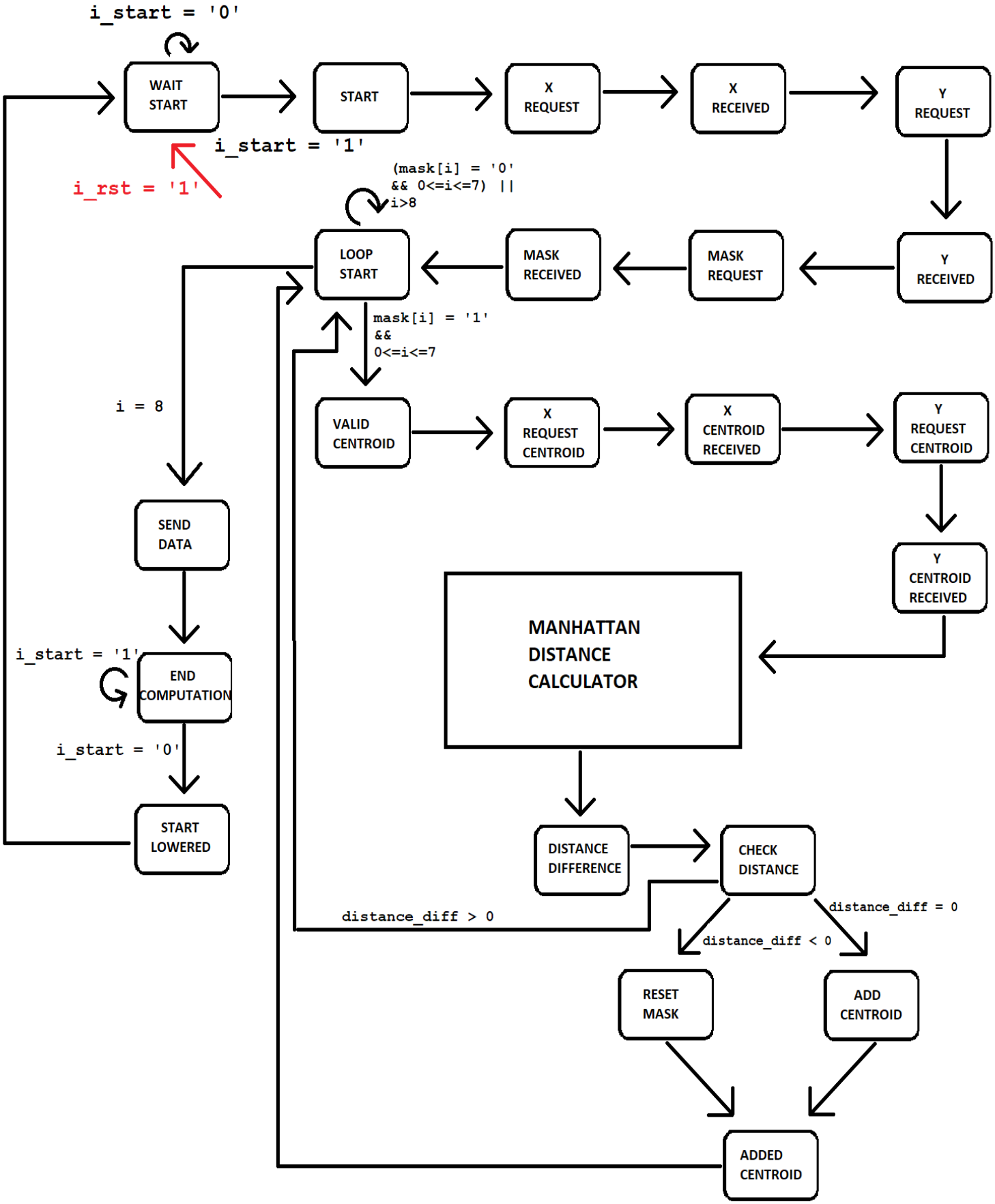
```
Main(){

//Variabili utilizzate:

      int x_osservatore;
      int y_osservatore;
      int x_centroide;
      int y_centroide;
      int distanza_min = 511;
      int distanza_corrente;
      int diff_x;
      int diff_y;
      int diff_distanze;
      int i = 0;
      bit maschera_ingresso[8];
      bit maschera_uscita[8] = "00000000";

//Letture della memoria:

      x_osservatore = RAM.read(17);
      y_osservatore = RAM.read(18);
      maschera_ingresso = RAM.read(0);

//Algoritmo:

      for (i=0; i<8; i++){

            if(maschera_ingresso[i] == 1){

                  x_centroide = RAM.read(2*i + 1);
                  y_centroide = RAM.read(2*i + 2);

                  diff_x = abs(x_osservatore - x_centroide);
                  diff_y = abs(y_ossservatore - y_centroide);
                  distanza_corrente = diff_x + diff_y;

                  diff_distanze = distanza_corrente - distanza_min;

                  if(diff_distanze<0){
                        min_distanza = distanza_corrente;
                        maschera_uscita = "00000000";
                        maschera_uscita[i] = '1';
                  }
                  elseif(diff_distanze == 0) {
                        maschera_uscita[i] = '1';
                  }
            }
      }

//Scrittura in memoria e termine

            RAM.write(19, maschera_uscita);
            return;}
```

L'algoritmo procede secondo un'idea semplice: si analizzano sequenzialmente tutti i centroidi validi, ovvero quelli stabiliti dalla maschera di ingresso. Per ognuno di essi, si calcola la distanza di Manhattan dall'osservatore. In memoria, inoltre, si conserva la minima distanza tra centroide e osservatore mai calcolata. Se la distanza del centroide in analisi dovesse risultare minore di tale distanza minima, la maschera di uscita viene aggiornata per indicare solamente il centroide in analisi come centroide più vicino all'osservatore, salvandone la distanza come nuova distanza minima. Se la distanza calcolata dovesse risultare uguale alla distanza minima, si aggiorna la maschera d'uscita aggiungendo ai centroidi già presenti anche il corrente, ad indicare che anch'esso si trova alla stessa minima distanza. Nel caso in cui la distanza dovesse risultare maggiore, la maschera di uscita non viene alterata, perché significa che vi è almeno un altro centroide più vicino all'osservatore.
Terminato il processo, si ottiene la maschera d'uscita corretta.

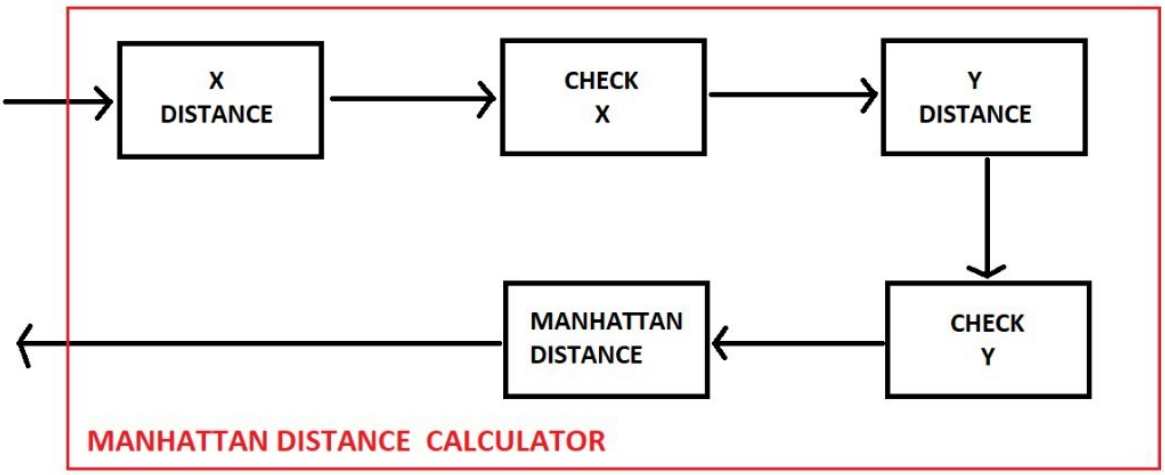
Più in particolare, si inizia con la dichiarazione delle variabili utilizzate. Si noti che la distanza minima viene inizializzata a 511 come valore di default. I centroidi, infatti, si possono trovare ad una distanza massima dall'osservatore pari a 510: ciò fa in modo che, qualsiasi sia la posizione reciproca di osservatore e primo centroide in analisi, la loro distanza venga automaticamente rilevata come minima, per essere quindi sovrascritta nell'apposita variabile e utilizzata correttamente per i confronti successivi. Ciò rende anche possibile risolvere i casi in cui un solo centroide sia da considerare valido, riconoscendolo automaticamente come il più vicino all'osservatore.
In seguito si ottengono i dati relativi alla posizione dell'osservatore e alla maschera di ingresso tramite delle richieste di lettura della memoria.
Si prosegue con l'inizio del ciclo che analizza i centroidi. Si itera su ogni bit della maschera d'ingresso con il ciclo for di indice "i". Nel caso in cui l'i-esimo bit sia 1, l'i-esimo centroide è da considerarsi valido.
Ne si ricavano quindi le coordinate con delle richieste di lettura della memoria, i cui indirizzi sono dipendenti da "i". Una volta ottenute, esse vengono utilizzate per calcolare, in modulo, la distanza dall'osservatore sull'asse delle ordinate e sull'asse delle ascisse ("diff_x" e "diff_y"). Queste distanze vengono poi sommate per ottenere la distanza di Manhattan dell'i-esimo centroide dall'osservatore, salvata nella variabile "distanza_corrente". Successivamente si confronta questa distanza con la minore mai osservata, sottraendole quest'ultima. Se il risultato è negativo, significa che la distanza corrente è minore della minima mai osservata, risultando di fatto essere la nuova distanza minima. La maschera di uscita viene quindi resettata, portandone ogni bit a '0', e ponendo ad '1' solamente il bit in i-esima posizione, corrispondente all'i-esimo centroide.
Se il risultato del confronto è 0, il centroide si trova esattamente alla minima distanza dall'osservatore. Bisogna quindi portare ad '1' il bit in i-esima posizione nella maschera di uscita, lasciando inalterati gli altri.
Se il risultato del confronto è positivo, infine, non vi è alcun bisogno di alterare la maschera d'uscita.
Il ciclo viene ripetuto per ogni bit della maschera d'ingresso.
Alla fine dell'elaborazione, si invia una richiesta di scrittura alla memoria per salvare il dato prodotto.

II.2 – MODELLIZZAZIONE MEDIANTE FSM

L'implementazione dell'algoritmo è stata effettuata modellizzando prima una macchina a stati (FSM) che ne svolgesse le operazioni.



Il blocco denominato "*Manhattan distance calculator*" comprende il gruppo di stati che effettuano il calcolo della distanza di Manhattan tra il centroide e l'osservatore. Per una maggior comodità di rappresentazione, si è deciso di rappresentare separatamente il relativo diagramma a stati, di seguito riportato:



In questa rappresentazione della FSM si possono notare alcuni stati il cui stato successivo non è univoco: su tali diramazioni sono quindi stati indicati i valori delle variabili che ne determinano la scelta. È inoltre stato indicato lo stato in cui la macchina si porta quando riceve il segnale di RESET rappresentato in rosso.

L'architettura del componente è stata quindi specificata in VHDL applicando il modello generale di una FSM, seguendo con precisione la nomenclatura degli stati del grafico presentato e il flusso di esecuzione dell'algoritmo descritto nel paragrafo II.1.

II.3 – SPECIFICA VHDL DEL COMPONENTE

La specifica VHDL del componente è composta dalla entity, presentata nella sezione I, e da due processi situati dopo l'elenco degli stati della macchina e la dichiarazione dei segnali interni. Si suggerisce di seguito un percorso di lettura del codice, assicurandosi di avere chiaro l'algoritmo implementato e la struttura della macchina a stati su cui si basa il componente hardware.

II.3.1 – Descrizione dei segnali

I segnali sono stati utilizzati per la gestione delle variabili dell'algoritmo.
Ogni segnale ha il suo corrispettivo next (ad esempio, un segnale s possiede sempre il suo next_s). Se uno stato deve modificare un segnale, anziché modificarlo direttamente, effettuerà la modifica solamente sul segnale "next". Il processo "registers", descritto più avanti, si occuperà di caricare in ogni segnale il corrispettivo "next" ad ogni ciclo di clock. I valori binari sono accompagnati dall'indicazione relativa alla rappresentazione utilizzata: SIGNED significa che sono considerati come valori binari in codifica modulo e segno, UNSIGNED significa che sono considerati come valori binari semplici.

Signal current_state, next_state :
`state_type;`

Segnale che rappresenta gli stati della FSM, di tipo state_type.

Signal x, y, next_x, next_y:
`std_logic_vector(8 downto 0); SIGNED`

Questi segnali rappresentano le coordinate dell' osservatore.
Esse sono rappresentate tramite uno standard logic vector di 9 bit, di cui 8 per il valore in modulo di x o y e 1 per il segno.

Signal mask, next_mask:
`std_logic_vector(7 downto 0); UNSIGNED`

Questo segnale rappresenta la maschera di input, ovvero uno standard logic vector di 8 bit, ognuno rappresentante un centroide.
Se il bit vale '1' il centroide è valido e va considerato nel calcolo della Manhattan distance.
Se invece vale '0' il centroide non va considerato.

Signal x_centroid, y_centroid, next_x_centroid, next_y_centroid :
`std_logic_vector(8 downto 0); SIGNED`

Questi segnali servono per salvare le coordinate del centroide da analizzare.
La loro rappresentazione è la stessa di x, y, next_x e next_y.
Le coordinate del centroide vanno salvate in questi segnali se e solo se il bit corrispondente al centroide della maschera di input vale '1'.

signal current_distance, next_current_distance:
`std_logic_vector(9 downto 0); SIGNED`

Questo segnale rappresenta la distanza dall'osservatore del centroide che viene analizzato nell'iterazione corrente.
È uno standard logic vector composto da 10 bit: la distanza (somma delle differenze di ascisse e ordinate del centroide e del punto) può avere dimensione 9 bit al massimo (510), ai quali bisogna concatenare un ulteriore bit per rispettare la convenzione signed.

Signal lowest_distance, next_lowest_distance :
`std_logic_vector (7 downto 0); SIGNED`

Questo segnale rappresenta la distanza del centroide più vicino all'osservatore.
Viene aggiornata ogni volta che si trova un centroide più vicino, altrimenti resta invariata.
Anch'essa è uno std_logic_vector da 10 bit, di cui l'ultimo usato per la convenzione signed.

Signal mask_out, next_mask_out:
`std_logic_vector (7 downto 0); UNSIGNED`

Questo segnale rappresenta la maschera di uscita. È un segnale che viene aggiornato ogni volta che un centroide risulta essere il più vicino (o tra i più vicini) all'osservatore.
La rappresentazione è a 8 bit e segue la logica dei segnali mask e next_mask.

Signal diff_x, diff_y, next_diff_x, next_diff_y :
`std_logic_vector (8 downto 0); SIGNED`

Questi segnali servono per salvare la differenza (x_centroid – x) e analogamente per le y.
Questa differenza sarà successivamente utilizzata per il calcolo della Manhattan distance.
Essendo ottenuta come differenza di due std_logic_vector da 9 bit ciascuno, è anch'essa uno std_logic_vector da 9 bit. Dato che è una differenza, essa può anche assumere valori negativi: per questo motivo si è scelto di rappresentare il suo valore (e quello delle coordinate dei punti) con la convenzione signed.

Signal i, next_i :
`std_logic_vector (4 downto 0); UNSIGNED`

Questo segnale rappresenta l'indice utile per le iterazioni del ciclo che parte dallo stato loop_start.
Il suo valore all'interno del ciclo varia da 0 a 8. Tuttavia, esso viene anche utilizzato per calcolare gli indirizzi a cui effettuare le letture della memoria (si veda lo pseudocodice definito nel paragrafo II.1). Dato che potrebbe capitare di dover leggere all'indirizzo 16, è necessario che "i" abbia 5 bit per poter rappresentare correttamente tale valore. Ecco perché è dichiarato come un vettore std_logic a 5 bit.

Signal distance_diff, next_distance_diff:
`std_logic_vector (9 downto 0); SIGNED`

Questo segnale serve per rappresentare la differenza tra current_distance e lowest_distance.
Essendo ricavato durante l'elaborazione come una differenza tra due std_logic_vector da 10 bit, è anch'esso uno std_logic_vector da 10 bit. Il decimo bit rappresenta sempre il segno.

II.3.2 - Processo "registers"

La gestione dei segnali della macchina, e quindi dei registri contenenti le variabili, è affidata ad un process sensibile al CLOCK e al segnale di RESET. In particolare, ogni volta che si presenta un fronte di salita nel segnale di CLOCK, tutti i registri vengono aggiornati con i nuovi valori prodotti in elaborazione, salvati nei segnali "next". Inoltre, si aggiorna anche lo stato della macchina.

```
120                 elsif rising_edge(i_clk) then
121
122                     current_state <= next_state;
123                     x <= next_x;
124                     y <= next_y;
125                     mask <= next_mask;
126                     x_centroid <= next_x_centroid;
127                     y_centroid <= next_y_centroid;
128                     mask_out <= next_mask_out;
129                     diff_x <= next_diff_x;
130                     diff_y <= next_diff_y;
131                     current_distance <= next_current_distance;
132                     distance_diff <= next_distance_diff;
133                     lowest_distance <= next_lowest_distance;
134                     i <= next_i;
```

Processo di aggiornamento dei registri, sezione relativa al CLOCK.

Ogni volta che si rileva un segnale di RESET, ogni segnale viene inizializzato al suo valore di default. La macchina, inoltre, si porta nello stato iniziale, `wait_start`, in attesa di un nuovo segnale START.

II.3.3 – Processo "logic"

Process che gestisce la logica degli stati della macchina, ovvero il loro comportamento. È costituito da un costrutto switch-case che controlla lo stato in cui si trova la macchina, e in base ad esso effettua determinate operazioni. È un process sensibile alle variazioni dei registri e ai segnali in ingresso nel componente.
Innanzitutto, viene compiuto un assegnamento preventivo di tutti i segnali "next" al loro valore precedente, e delle uscite del componente al loro valore di default. Ciò viene fatto onde evitare di generare segnali non inizializzati per errore, e di introdurre così dei latch indesiderati nella fase di sintesi. Ogni stato dovrà quindi assegnare soltanto i segnali che è previsto modificare, senza alterare o preoccuparsi degli altri. Segue una descrizione dettagliata del comportamento di ogni stato.

II.3.4 – Descrizione degli stati

Prima di elencare gli stati e il loro comportamento, è opportuno spiegare le procedure di lettura e scrittura per la RAM, che compariranno spesso nella descrizione degli stati:

- LETTURA DELLA MEMORIA RAM:

Per leggere un dato dalla RAM è necessario effettuare una richiesta di lettura di quel dato. Il procedimento è il seguente:

- Inviare alla RAM l'indirizzo del dato da leggere caricandolo nel segnale o_address;
- Settare al valore alto il bit di enable per la RAM, quindi o_en = '1';
- Assegnare al bit di enable per la scrittura il valore basso, quindi o_we = '0';

Successivamente, attraverso uno stato di attesa, si aspetta che la RAM fornisca il dato richiesto. Ciò avviene dopo un ciclo di clock.

Infine, nello stato successivo, si può utilizzare il dato, che viene ricevuto tramite il segnale di input proveniente dalla RAM (i_data).

- SCRITTURA NELLA MEMORIA RAM:

Per scrivere nella RAM il procedimento è simile a quello necessario per la lettura. Infatti, per inviare un dato alla ram, è necessario:

- Settare il segnale di enable per la RAM ad '1' -> o_en = '1';
- Settare il segnale di enable per la scrittura ad '1', quindi o_we = '1';
- Indicare il dato da scrivere attraverso il segnale di output della ram (o_data <= a);
- Indicare l'indirizzo in cui scrivere il dato, tramite o_address;

Successivamente bisognerà aspettare un ciclo di clock affinchè la RAM riceva il dato.

È ora possibile analizzare i singoli stati e il loro comportamento.

wait_start:

Questo è il primo stato della FSM. È uno stato passivo che attende che il segnale i_start passi da '0' a '1'.
Finché i_start = '0', il componente resta in wait_start. Quando i_start = '1', passa allo stato successivo, ossia start.

start:

In questo stato, ricevuto il segnale i_start, inizia l'elaborazione e viene inviata alla RAM la richiesta della coordinata x dell'osservatore, in modo da poterla leggere. Il componente passa quindi allo stato x_request.

x_request:

Questo è uno stato d'attesa, in cui si aspetta che la RAM fornisca il dato richiesto. Successivamente, la macchina va allo stato x_received.

x_received:

Una volta ricevuta la x dell'osservatore, viene salvata nel segnale next_x, aggiungendo uno '0' a sinistra per rispettare la convenzione signed. La memoria infatti fornisce il valore ad 8 bit con la convenzione unsigned. Essendo una distanza, esso sarà sempre positivo, quindi può essere convertito in binario signed semplicemente concatenandolo ad uno 0.
Inoltre, si invia una richiesta alla RAM per leggere la coordinata y dell'osservatore, per passare allo stato y_request.

y_request:

Questo stato è analogo a x_received e il suo successivo è y_received.

y_received:

Si riceve dalla RAM la y dell'osservatore, la si salva su next_y in modo analogo a come è stato fatto per x e si invia alla RAM la richiesta per la lettura della maschera d'ingresso.
Si passa dunque allo stato mask_request.

mask_request:

Anche questo stato segue la stessa logica di x_received, essendo uno stato di attesa della risposta della memoria, e il successivo è mask_received.

mask_received:

In questo stato si riceve la maschera di input dalla RAM, che viene salvata su next_mask, e si passa a loop_start.

loop_start:

Questo stato usa un indice (i) per scorrere i centroidi. Il valore dell'indice è controllato mediante un costrutto switch-case.

Quando 0 <= i <=7, se l'i-esimo bit della maschera di input vale '1', allora il centroide è valido e va considerato per il calcolo della manhattan distance. In questo caso si passa allo stato valid_centroid.
Se invece il bit è uguale a '0', allora il centroide non è valido, quindi si resta in loop_start e si incrementa i di 1.

Quando i = 8 invece, vuol dire che tutti i centroidi validi sono stati analizzati e quindi si può uscire da loop_start, scrivere sulla RAM il valore della maschera d'uscita e passare allo stato send_data.

valid_centroid:

In questo stato si manda alla RAM la richiesta per ricevere la x del centroide e si passa allo stato x_request_centroid.

x_request_centroid:

Questo stato è analogo a x_request e il suo successivo è x_centroid_received.

x_centroid_received:

Qui si riceve la x del centroide dalla RAM, viene salvata in next_x_centroid in modo analogo al procedimento per next_x nello stato x_received, e si richiede alla RAM la y del centroide. Il prossimo stato sarà y_request_centroid.

y_request_centroid:

Anche questo stato è analogo a x_request e il successivo è y_centroid_received.

y_centroid_received:

In questo stato si riceve la y del centroide, che viene salvata in next_y_centroid. Lo stato successivo è x_distance, ovvero l'inizio del blocco per il calcolo della Manhattan distance.

MANHATTAN DISTANCE

x_distance:

In questo stato si calcola la differenza tra la x del centroide e quella dell'osservatore. Il risultato è salvato in next_diff_x e lo stato successivo è check_x;

check_x:

Controlla se la differenza calcolata allo stato precedente è positiva o negativa.
Se è negativa, la inverte di segno, effettuando nuovamente l'operazione di sottrazione ma scambiandone minuendo e sottraendo.
Successivamente si passa a y_distance.

y_distance:

Questo stato è analogo a x_distance ed ha come successivo check_y.

check_y:

Questo stato è analogo a check_x e il successivo è manhattan_distance.

manhattan_distance:

Qui viene calcolata la Manhattan distance, ovvero la somma delle distanze delle ascisse e ordinate. Il risultato viene salvato in next_current_distance, concatenandolo ad uno '0' per rispettare la convenzione di segno.
Si passa quindi allo stato distance_difference e si esce così dal blocco MANHATTAN DISTANCE CALCULATOR.

distance_difference:

In questo stato viene calcolata la differenza tra la current_distance e la lowest_distance. Il risultato viene messo in next_distance_diff e lo stato successivo è check_distance.

check_distance:

Qui si controlla quanto vale distance_diff:

- Se distance_diff < 0 bisogna aggiornare la next_lowest_distance con il valore della current distance e si passa allo stato reset_mask;
- Se distance_diff = 0 vuol dire che il centroide ha distanza uguale alla minima e si passa allo stato add_centroid;
- Se distance_diff > 0 vuol dire che il centroide non è il più lontano, si incrementa di 1 i e si torna allo stato loop_start.

add_centroid:

Si raggiunge questo stato se il centroide analizzato è alla stessa distanza del più vicino rispetto all'osservatore.
In questo stato si aggiunge un '1' alla maschera d'uscita (next_mask_out) all'(i)-esimo bit.
Fatto ciò, si passa allo stato added_centroid.

added_centroid:

Termine dell'aggiunta del centroide. In questo stato si incrementa di 1 i e si torna a loop_start.

reset_mask:

Si raggiunge questo stato da check_distance se il centroide analizzato è più vicino rispetto al centroide a minima distanza dall'osservatore.
In questo caso si resetta la maschera d'uscita, e ad essa viene messo un '1' all'(i)-esimo bit.
Successivamente si passa allo stato added_centroid.

send_data:

Si arriva in questo stato da loop_start una volta finito di analizzare tutti i centroidi.
Si mette il segnale o_done = '1' per indicare la fine della computazione e si passa allo stato end_computation.

end_computation:

In questo stato si aspetta che i_start torni = '0'.
Finchè resta = '1' si rimane in end_computation, una volta passato a '0' si mette o_done = '0' e si entra in lowered_start.

lowered_start:

Questo è l'ultimo stato. Qui si resettano tutti i segnali al loro valore di default e si torna al primo stato, ovvero wait_start. In questo modo, il componente è pronto per iniziare una nuova computazione.

II.4 – OTTIMIZZAZIONI

Durante la fase di progettazione, sono state individuate alcune ottimizzazioni per cercare di migliorare le prestazioni del componente.

La lettura dei dati dalla memoria richiede un ciclo di clock di attesa affinché il dato venga inviato e si possa ritenere stabilizzato; per diminuire il numero di letture effettuate, e quindi gli stati di attesa, si è deciso di richiedere alla memoria i dati relativi alle coordinate del centroide solamente nel caso in cui il centroide sia effettivamente da analizzare. Qualunque centroide identificato da uno "0" nella maschera d'ingresso viene ignorato, e per esso nessuna operazione di lettura viene effettuata. Questa ottimizzazione è stata implementata in modo semplice grazie al ciclo di indice "i", e risulta evidente anche dallo pseudocodice del paragrafo II.1. Il tempo impiegato dal componente per risolvere il problema dipende quindi dal numero di centroidi in analisi, risultando minimo per il caso banale (nessun centroide da analizzare) e massimo per il caso pessimo (tutti i centroidi da analizzare). Ulteriori informazioni e considerazioni sulle prestazioni temporali del componente sono presentate nella sezione IV.

Inoltre, è stato implementato un meccanismo ispirato al pipelining: quando si riceve un dato dalla memoria RAM durante la lettura delle coordinate dei centroidi e del punto, si prepara già la richiesta successiva, caricando l'indirizzo di lettura sull'apposita uscita e attivando i segnali necessari, risparmiando di fatto un ciclo di clock ogni volta che vengono effettuate più richieste di lettura in modo sequenziale.

III - TESTING

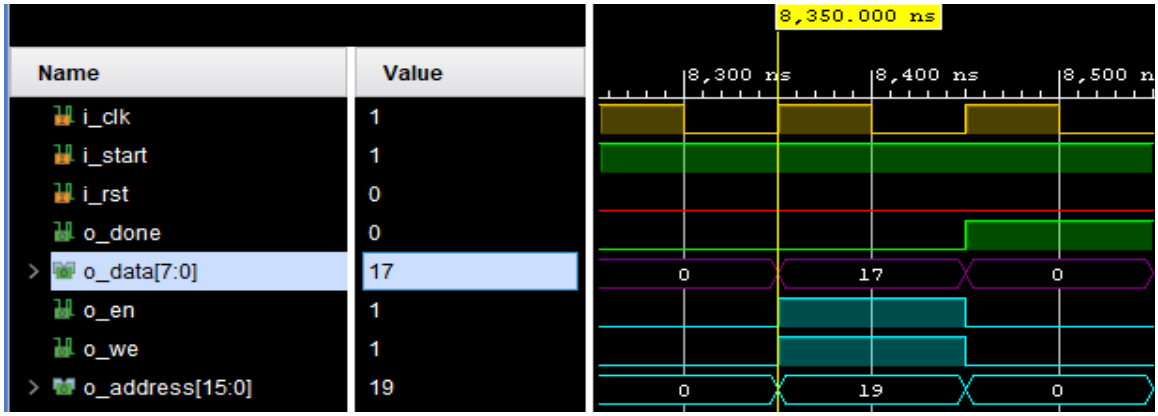
Si presenta in questa sezione un'analisi dei casi di test ai quali il componente è stato sottoposto.
I testbench utilizzati sono stati modellati sulla base dell'esempio fornito dal personale docente, e il loro codice è fornito nella sezione V – Appendice.
Il testing è stato effettuato per sottoporre il componente hardware ad elaborazioni particolarmente complesse, con lo scopo di verificarne la robustezza e il corretto comportamento. In particolare, si sono analizzati:

- *Casi limite in termini di spazio e dimensione dei dati;*
- *Corretto comportamento di fronte a segnali ricevuti o attesi.*

Di seguito sono esposti i test utilizzati:

0) Testbench di prova

Testbench fornito dal personale docente. L'osservatore si trova alle coordinate (78, 33) ed è richiesta l'analisi del primo, del quarto, del quinto, del sesto e dell'ottavo centroide. Di essi, il primo e il quinto si trovano alla distanza minima dall'osservatore. La maschera di uscita prevista è quindi 00010001 (17), e il componente la produce correttamente.



Risultato della simulazione di Vivado. In memoria viene scritta la maschera d'uscita corretta all'indirizzo 19.

1) Distanza massima tra l'osservatore e il centroide

In questo test l'osservatore è posizionato alle coordinate (0, 0) e si considera un unico centroide da analizzare posto alle coordinate (255, 255), al limite estremo dello spazio. Con ciò si affronta il caso limite in cui la distanza osservatore - centroide (current_distance) è massima, e si verifica il corretto comportamento del componente anche in questo caso.

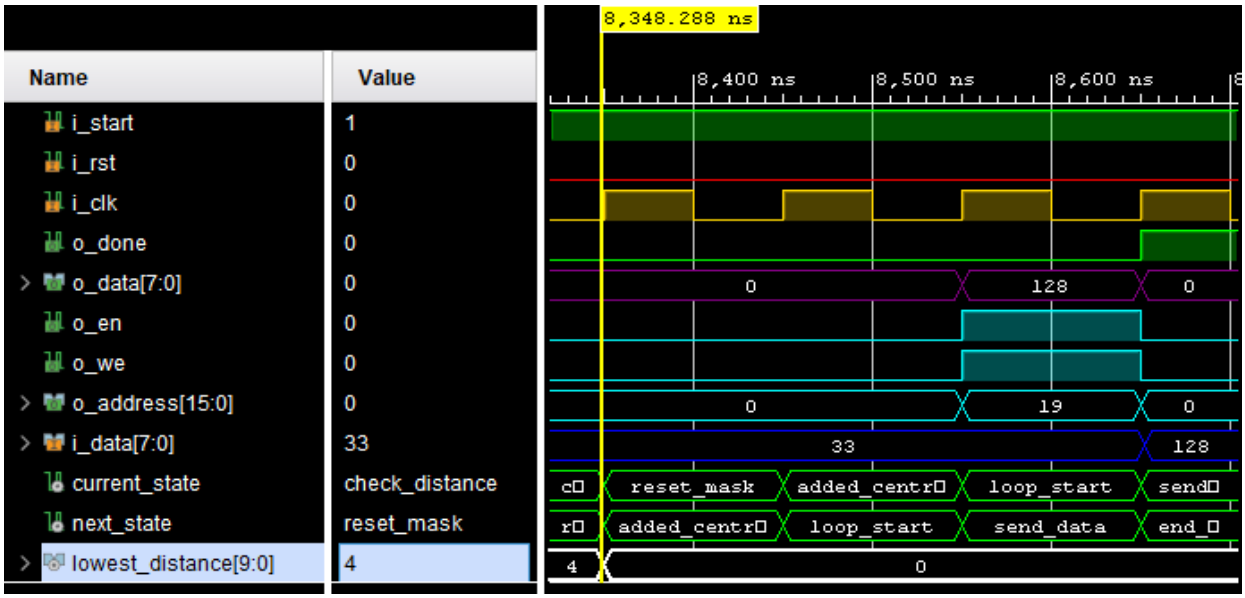
Il test si chiama massima_distanza, il centroide da considerare è solo il primo e l'output atteso è "00000001". Il componente lo produce correttamente.

La distanza corrente viene correttamente riconosciuta, e segnalata come distanza minima, anche se presenta il massimo valore osservabile.

2) Distanza minima tra l'osservatore e il centroide (coincidenti)

In questo test l'osservatore e uno dei centroidi da analizzare sono alle coordinate (78, 33) avendo quindi distanza nulla. In questo caso il componente hardware deve riuscire a produrre il corretto risultato anche se i due punti coincidono.

Il test si chiama centroide_coincidente_punto, la maschera d'ingresso è 10111001 e il centroide coincidente è l'ottavo. L'output atteso è "10000000". Il componente lo produce correttamente.



La distanza pari a 0 viene correttamente calcolata.

3) I centroidi sono tutti coincidenti

In questo test tutti tutti gli 8 centroidi sono stati posizionati alle stesse coordinate.
La maschera d'ingresso prevede l'analisi solamente di alcuni di essi.
In questo caso, essendo i centroidi coincidenti, l'output atteso è uguale alla maschera di ingresso.

Il test si chiama centroidi_coincidenti, i centroidi si trovano tutti alle coordinate (75, 32), la maschera di ingresso è "10111001" , ovvero 185, e coincide con la maschera di output, correttamente prodotta dal componente.

4) Più centroidi sono alla stessa distanza minima dall'osservatore

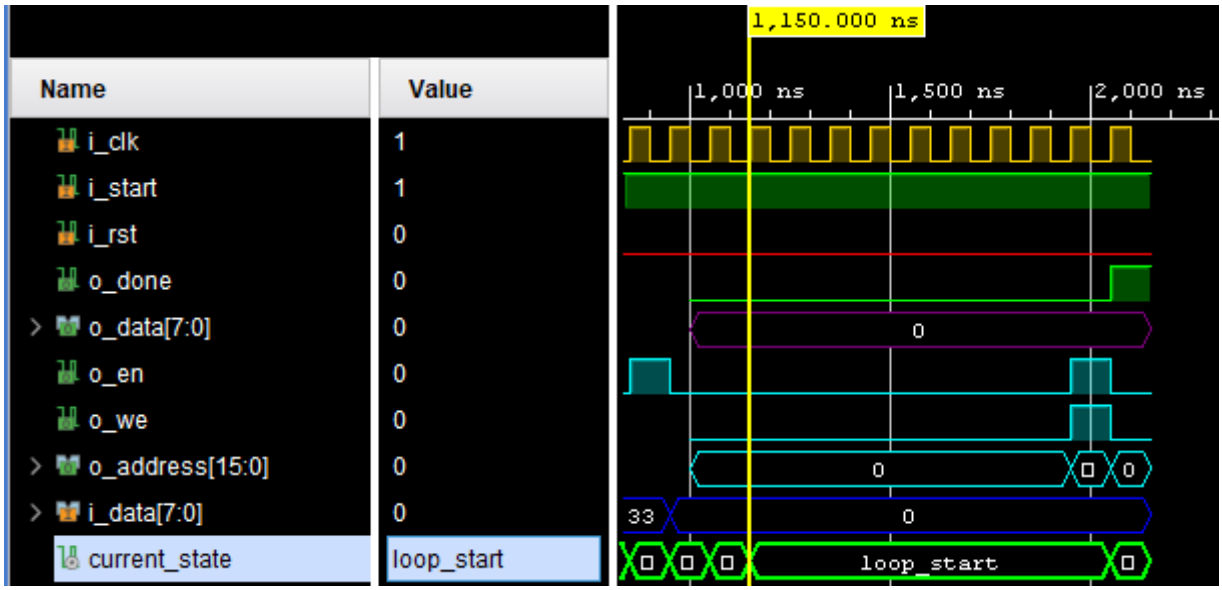
In questo test, i centroidi da considerare sono due, in particolare il primo e il secondo.
Nel dettaglio, l'osservatore è stato posizionato alle coordinate (0, 0), mentre i due centroidi sono rispettivamente alle coordinate (100, 0) e (0, 100).
Questo caso di test è simile al precedente, ed essendo le due distanze uguali, ci si aspetta che la maschera di output coincida con quella di input.

Il test si chiama centroidi_stessa_distanza, la maschera di ingresso è uguale a "00000011", ovvero 3, ed è uguale quella d'uscita, correttamente prodotta dal componente.

5) Nessun centroide è tra quelli da analizzare (Maschera di input nulla)

In questo test la maschera di input è settata a "00000000". Nessuno dei centroidi viene quindi considerato durante l'elaborazione.
Nello specifico, il componente non passerà mai dallo stato loop_start allo stato valid_centroid, perché ogni bit della maschera controllato da loop_start sarà uguale a '0'.
Di conseguenza, la maschera di output attesa è nulla.

Il test si chiama maschera_nulla, la maschera di input è "00000000" ed è uguale a quella di output, correttamente prodotta dal componente.



Il componente non esegue mai la diramazione verso lo stato valid_centroid, restando in loop_start.

6) Tutti i centroidi sono da analizzare (maschera di input massima)

In questo test la maschera d'ingresso è "11111111", costringendo il componente a valutare la distanza di ogni centroide. Il ciclo che inizia dallo stato loop_start troverà ogni bit della maschera ad '1' e, per ogni centroide, passerà allo stato valid_centroid, in modo da analizzarlo.

Il test si chiama maschera_massima, e i centroidi più vicini sono il primo e il quarto. Di conseguenza la maschera di output sarà uguale a "00000101", ossia 5. Il dato viene correttamente prodotto dal componente.

7) I centroidi da analizzare e l'osservatore si trovano nei punti estremi della matrice 256x256

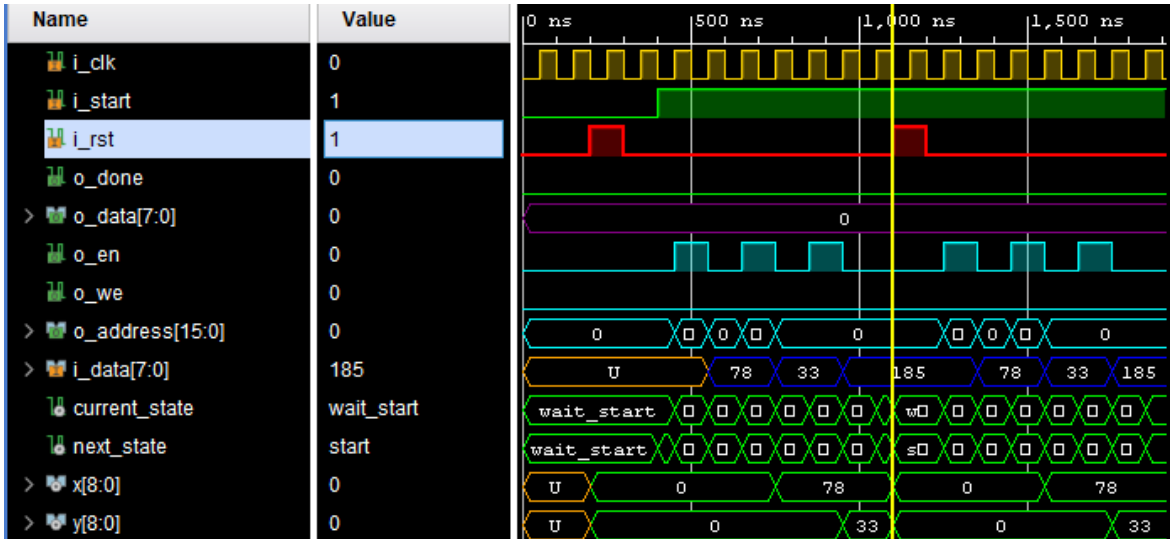
Test effettuato per accertarsi che il componente si comporti nella corretta maniera anche quando deve operare agli estremi della matrice 256x256.

Il test si chiama punti_estremi, l'osservatore si trova alle coordinate (0, 0), i centroidi da analizzare sono i primi 3 e si trovano rispettivamente alle coordinate (255, 0), (0, 255) e (255, 255) (Maschera d'ingresso = 7). In questo caso, i centroidi più vicini sono i primi 2, e quindi l'output atteso è "00000011", ovvero 3. Il dato viene correttamente prodotto dal componente.

8) Il componente riceve un segnale di reset (i_rst = '1') successivo al primo reset

In questo test si è voluto verificare che il componente hardware
non avrebbe avuto problemi nel caso in cui avesse ricevuto un segnale di
RESET in un momento qualunque dell'elaborazione.
Il comportamento atteso in questo caso è un riavvio della computazione appena il segnale di RESET torna a '0' (i_rst = '0'), seguito da una normale esecuzione della stessa.

Il test si chiama reset_ricevuto. Dopo 1100 ns dall'inizio della
computazione si ha la ricezione di un secondo segnale di RESET.
Il resto del testbench è uguale a quello di prova fornito dal personale docente e l'output atteso è lo stesso, ovvero 17 ("00010001"). Il componente produce correttamente il risultato, e il riavvio dell'elaborazione procede senza problemi.



Un secondo segnale di RESET è ricevuto. L'elaborazione ricomincia correttamente.

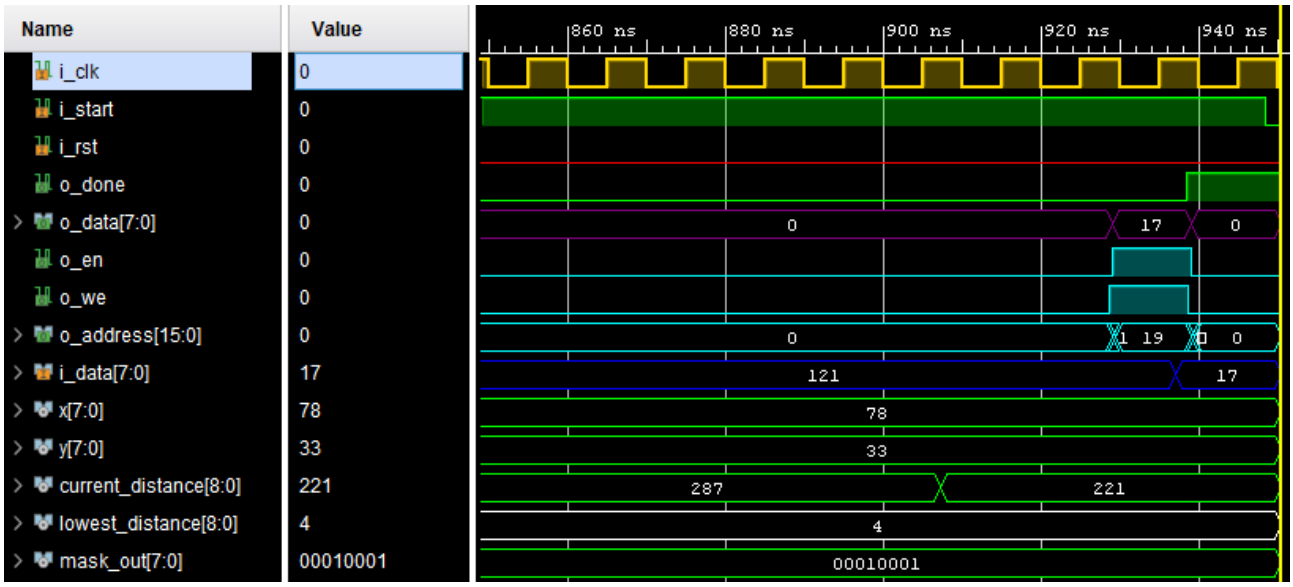
9) Nello stato end_computation il segnale i_start non si abbassa mai

Test effettuato per affrontare l'eventualità in cui alla fine della computazione
non venga mai abbassato il segnale di START.
La macchina a stati può infatti passare da end_computation allo stato successivo, ovvero lo stato lowered_start, solo nel caso in cui i_start passi da '1' a '0'.
In tutti gli altri casi rimane in end_computation. Una nuova computazione
non può quindi essere iniziata se prima non si abbassa il segnale START.
Il risultato atteso del test, quindi, è che la computazione non termini mai.

Il test si chiama start_non_abbassato e, a differenza degli altri test, START non
viene mai portato a '0' al termine dell'elaborazione.
Perciò la computazione non terminerebbe mai se non fosse bloccata manualmente tramite Vivado dopo un po' di tempo. Il componente presenta correttamente questo comportamento.

IV – RISULTATI DELLA SINTESI

Il componente hardware risulta sintetizzabile, e tutti i casi di test a cui è stato sottoposto vengono superati anche in Post-Synthesis Timing Simulation. Inoltre, il corretto funzionamento del componente è stato accertato anche per periodi di clock molto inferiori a 100 ns. Si fornisce qui la rappresentazione delle forme d'onda relative all'esecuzione del test fornito dal personale docente con un periodo di clock di 10 ns:



V – APPENDICE

Viene di seguito riportato il codice sorgente dei casi di test utilizzati.

0) Testbench fornito dal personale docente

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;

entity project_tb is
end project_tb;

architecture projecttb of project_tb is
constant c_CLOCK_PERIOD            : time := 100 ns;
signal   tb_done                         : std_logic;
signal   mem_address             : std_logic_vector (15 downto 0) := (others => '0');
signal   tb_rst                          : std_logic := '0';
signal   tb_start                        : std_logic := '0';
signal   tb_clk                              : std_logic := '0';
signal   mem_o_data,mem_i_data          : std_logic_vector (7 downto 0);
signal   enable_wire                 : std_logic;
signal   mem_we                              : std_logic;

type ram_type is array (65535 downto 0) of std_logic_vector(7 downto 0);

-- come da esempio su specifica
signal RAM: ram_type := (0 => std_logic_vector(to_unsigned( 185 , 8)),
                         1 => std_logic_vector(to_unsigned( 75 , 8)),
                         2 => std_logic_vector(to_unsigned( 32 , 8)),
                         3 => std_logic_vector(to_unsigned( 111 , 8)),
                         4 => std_logic_vector(to_unsigned( 213 , 8)),
                         5 => std_logic_vector(to_unsigned( 79 , 8)),
                         6 => std_logic_vector(to_unsigned( 33 , 8)),
                         7 => std_logic_vector(to_unsigned( 1 , 8)),
                         8 => std_logic_vector(to_unsigned( 33 , 8)),
                         9 => std_logic_vector(to_unsigned( 80 , 8)),
                         10 => std_logic_vector(to_unsigned( 35 , 8)),
                         11 => std_logic_vector(to_unsigned( 12 , 8)),
                         12 => std_logic_vector(to_unsigned( 254 , 8)),
                         13 => std_logic_vector(to_unsigned( 215 , 8)),
                         14 => std_logic_vector(to_unsigned( 78 , 8)),
                         15 => std_logic_vector(to_unsigned( 211 , 8)),
                         16 => std_logic_vector(to_unsigned( 121 , 8)),
                         17 => std_logic_vector(to_unsigned( 78 , 8)),
                         18 => std_logic_vector(to_unsigned( 33 , 8)),
                       others => (others =>'0'));

component 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 component project_reti_logiche;


begin
UUT: project_reti_logiche
port map (
          i_clk       => tb_clk,
          i_start       => tb_start,
          i_rst       => tb_rst,
          i_data      => mem_o_data,
          o_address   => mem_address,
          o_done      => tb_done,
          o_en            => enable_wire,
          o_we                    => mem_we,
          o_data      => mem_i_data
```

```
          );

p_CLK_GEN : process is
begin
    wait for c_CLOCK_PERIOD/2;
    tb_clk <= not tb_clk;
end process p_CLK_GEN;


MEM : process(tb_clk)
begin
    if tb_clk'event and tb_clk = '1' then
        if enable_wire = '1' then
            if mem_we = '1' then
                RAM(conv_integer(mem_address))  <= mem_i_data;
                mem_o_data                      <= mem_i_data after 2 ns;
            else
                mem_o_data <= RAM(conv_integer(mem_address)) after 2 ns;
            end if;
        end if;
    end if;
end process;


test : process is
begin
    wait for 100 ns;
    wait for c_CLOCK_PERIOD;
    tb_rst <= '1';
    wait for c_CLOCK_PERIOD;
    tb_rst <= '0';
    wait for c_CLOCK_PERIOD;
    tb_start <= '1';
    wait for c_CLOCK_PERIOD;
    wait until tb_done = '1';
    wait for c_CLOCK_PERIOD;
    tb_start <= '0';
    wait until tb_done = '0';

    -- Maschera di output = 00010001
    assert RAM(19) = std_logic_vector(to_unsigned( 17 , 8)) report "TEST FALLITO" severity failure;

    assert false report "Simulation Ended!, TEST PASSATO" severity failure;
end process test;

end projecttb;
```

Per i successivi casi di test, viene solamente fornita l'inizializzazione della memoria RAM e il comando di assert per verificare il dato prodotto.

1) massima_distanza

```
signal RAM: ram_type := (0 => std_logic_vector(to_unsigned( 1 , 8)),
                         1 => std_logic_vector(to_unsigned( 255 , 8)),
                         2 => std_logic_vector(to_unsigned( 255 , 8)),
                         3 => std_logic_vector(to_unsigned( 111 , 8)),
                         4 => std_logic_vector(to_unsigned( 213 , 8)),
                         5 => std_logic_vector(to_unsigned( 79 , 8)),
                         6 => std_logic_vector(to_unsigned( 33 , 8)),
                         7 => std_logic_vector(to_unsigned( 1 , 8)),
                         8 => std_logic_vector(to_unsigned( 33 , 8)),
                         9 => std_logic_vector(to_unsigned( 80 , 8)),
                         10 => std_logic_vector(to_unsigned( 35 , 8)),
                         11 => std_logic_vector(to_unsigned( 12 , 8)),
                         12 => std_logic_vector(to_unsigned( 254 , 8)),
                         13 => std_logic_vector(to_unsigned( 215 , 8)),
                         14 => std_logic_vector(to_unsigned( 78 , 8)),
                         15 => std_logic_vector(to_unsigned( 211 , 8)),
                         16 => std_logic_vector(to_unsigned( 121 , 8)),
                         17 => std_logic_vector(to_unsigned( 0 , 8)),
                         18 => std_logic_vector(to_unsigned( 0 , 8)),
                        others => (others =>'0'));

-- Maschera di output = 00000001
    assert RAM(19) = std_logic_vector(to_unsigned( 1 , 8)) report "TEST FALLITO" severity failure;
```

2) centroide_coincidente_punto

```
signal RAM: ram_type := (0 => std_logic_vector(to_unsigned( 185 , 8)),
                         1 => std_logic_vector(to_unsigned( 75 , 8)),
                         2 => std_logic_vector(to_unsigned( 32 , 8)),
                         3 => std_logic_vector(to_unsigned( 111 , 8)),
                         4 => std_logic_vector(to_unsigned( 213 , 8)),
                         5 => std_logic_vector(to_unsigned( 79 , 8)),
                         6 => std_logic_vector(to_unsigned( 33 , 8)),
                         7 => std_logic_vector(to_unsigned( 1 , 8)),
                         8 => std_logic_vector(to_unsigned( 33 , 8)),
                         9 => std_logic_vector(to_unsigned( 80 , 8)),
                         10 => std_logic_vector(to_unsigned( 35 , 8)),
                         11 => std_logic_vector(to_unsigned( 12 , 8)),
                         12 => std_logic_vector(to_unsigned( 254 , 8)),
                         13 => std_logic_vector(to_unsigned( 215 , 8)),
                         14 => std_logic_vector(to_unsigned( 78 , 8)),
                         15 => std_logic_vector(to_unsigned( 78 , 8)),
                         16 => std_logic_vector(to_unsigned( 33 , 8)),
                         17 => std_logic_vector(to_unsigned( 78 , 8)),
                         18 => std_logic_vector(to_unsigned( 33 , 8)),
                        others => (others =>'0'));

  -- Maschera di output = 10000000
    assert RAM(19) = std_logic_vector(to_unsigned( 128 , 8)) report "TEST FALLITO" severity failure;
```

3) centroidi_coincidenti

```
signal RAM: ram_type := (0 => std_logic_vector(to_unsigned( 185 , 8)),
                         1 => std_logic_vector(to_unsigned( 75 , 8)),
                         2 => std_logic_vector(to_unsigned( 32 , 8)),
                         3 => std_logic_vector(to_unsigned( 75 , 8)),
                         4 => std_logic_vector(to_unsigned( 32 , 8)),
                         5 => std_logic_vector(to_unsigned( 75 , 8)),
                         6 => std_logic_vector(to_unsigned( 32 , 8)),
                         7 => std_logic_vector(to_unsigned( 75 , 8)),
                         8 => std_logic_vector(to_unsigned( 32 , 8)),
                         9 => std_logic_vector(to_unsigned( 75 , 8)),
                         10 => std_logic_vector(to_unsigned( 32 , 8)),
                         11 => std_logic_vector(to_unsigned( 75 , 8)),
                         12 => std_logic_vector(to_unsigned( 32 , 8)),
                         13 => std_logic_vector(to_unsigned( 75 , 8)),
                         14 => std_logic_vector(to_unsigned( 32 , 8)),
                         15 => std_logic_vector(to_unsigned( 75 , 8)),
                         16 => std_logic_vector(to_unsigned( 32 , 8)),
                         17 => std_logic_vector(to_unsigned( 78 , 8)),
                         18 => std_logic_vector(to_unsigned( 33 , 8)),
                        others => (others =>'0'));

 -- Maschera di output = 10111001
    assert RAM(19) = std_logic_vector(to_unsigned( 185 , 8)) report "TEST FALLITO" severity failure;
```

4) centroidi_stessa_distanza

```
signal RAM: ram_type := (0 => std_logic_vector(to_unsigned( 3 , 8)),
                         1 => std_logic_vector(to_unsigned( 100 , 8)),
                         2 => std_logic_vector(to_unsigned( 0 , 8)),
                         3 => std_logic_vector(to_unsigned( 0 , 8)),
                         4 => std_logic_vector(to_unsigned( 100 , 8)),
                         5 => std_logic_vector(to_unsigned( 79 , 8)),
                         6 => std_logic_vector(to_unsigned( 33 , 8)),
                         7 => std_logic_vector(to_unsigned( 1 , 8)),
                         8 => std_logic_vector(to_unsigned( 33 , 8)),
                         9 => std_logic_vector(to_unsigned( 80 , 8)),
                         10 => std_logic_vector(to_unsigned( 35 , 8)),
                         11 => std_logic_vector(to_unsigned( 12 , 8)),
                         12 => std_logic_vector(to_unsigned( 254 , 8)),
                         13 => std_logic_vector(to_unsigned( 215 , 8)),
                         14 => std_logic_vector(to_unsigned( 78 , 8)),
                         15 => std_logic_vector(to_unsigned( 211 , 8)),
                         16 => std_logic_vector(to_unsigned( 121 , 8)),
                         17 => std_logic_vector(to_unsigned( 0 , 8)),
                         18 => std_logic_vector(to_unsigned( 0 , 8)),
                        others => (others =>'0'));

  -- Maschera di output = 00000011
    assert RAM(19) = std_logic_vector(to_unsigned( 3 , 8)) report "TEST FALLITO" severity failure;
```

5) maschera_nulla

```
signal RAM: ram_type := (0 => std_logic_vector(to_unsigned( 0 , 8)),
                         1 => std_logic_vector(to_unsigned( 75 , 8)),
                         2 => std_logic_vector(to_unsigned( 32 , 8)),
                         3 => std_logic_vector(to_unsigned( 111 , 8)),
                         4 => std_logic_vector(to_unsigned( 213 , 8)),
                         5 => std_logic_vector(to_unsigned( 79 , 8)),
                         6 => std_logic_vector(to_unsigned( 33 , 8)),
                         7 => std_logic_vector(to_unsigned( 1 , 8)),
                         8 => std_logic_vector(to_unsigned( 33 , 8)),
                         9 => std_logic_vector(to_unsigned( 80 , 8)),
                         10 => std_logic_vector(to_unsigned( 35 , 8)),
                         11 => std_logic_vector(to_unsigned( 12 , 8)),
                         12 => std_logic_vector(to_unsigned( 254 , 8)),
                         13 => std_logic_vector(to_unsigned( 215 , 8)),
                         14 => std_logic_vector(to_unsigned( 78 , 8)),
                         15 => std_logic_vector(to_unsigned( 211 , 8)),
                         16 => std_logic_vector(to_unsigned( 121 , 8)),
                         17 => std_logic_vector(to_unsigned( 78 , 8)),
                         18 => std_logic_vector(to_unsigned( 33 , 8)),
                        others => (others =>'0'));

   -- Maschera di output = 00000000
    assert RAM(19) = std_logic_vector(to_unsigned( 0 , 8)) report "TEST FALLITO" severity failure;
```

6) maschera_massima

```
signal RAM: ram_type := (0 => std_logic_vector(to_unsigned( 255 , 8)),
                         1 => std_logic_vector(to_unsigned( 79 , 8)),
                         2 => std_logic_vector(to_unsigned( 33 , 8)),
                         3 => std_logic_vector(to_unsigned( 111 , 8)),
                         4 => std_logic_vector(to_unsigned( 213 , 8)),
                         5 => std_logic_vector(to_unsigned( 79 , 8)),
                         6 => std_logic_vector(to_unsigned( 33 , 8)),
                         7 => std_logic_vector(to_unsigned( 1 , 8)),
                         8 => std_logic_vector(to_unsigned( 33 , 8)),
                         9 => std_logic_vector(to_unsigned( 80 , 8)),
                         10 => std_logic_vector(to_unsigned( 35 , 8)),
                         11 => std_logic_vector(to_unsigned( 12 , 8)),
                         12 => std_logic_vector(to_unsigned( 254 , 8)),
                         13 => std_logic_vector(to_unsigned( 215 , 8)),
                         14 => std_logic_vector(to_unsigned( 78 , 8)),
                         15 => std_logic_vector(to_unsigned( 211 , 8)),
                         16 => std_logic_vector(to_unsigned( 121 , 8)),
                         17 => std_logic_vector(to_unsigned( 78 , 8)),
                         18 => std_logic_vector(to_unsigned( 33 , 8)),
                        others => (others =>'0'));


    -- Maschera di output = 00000101
    assert RAM(19) = std_logic_vector(to_unsigned( 5 , 8)) report "TEST FALLITO" severity failure;
```

7) punti_estremi

```
signal RAM: ram_type := (0 => std_logic_vector(to_unsigned( 7 , 8)),
                         1 => std_logic_vector(to_unsigned( 255 , 8)),
                         2 => std_logic_vector(to_unsigned( 0 , 8)),
                         3 => std_logic_vector(to_unsigned( 0 , 8)),
                         4 => std_logic_vector(to_unsigned( 255 , 8)),
                         5 => std_logic_vector(to_unsigned( 255 , 8)),
                         6 => std_logic_vector(to_unsigned( 255 , 8)),
                         7 => std_logic_vector(to_unsigned( 1 , 8)),
                         8 => std_logic_vector(to_unsigned( 33 , 8)),
                         9 => std_logic_vector(to_unsigned( 80 , 8)),
                         10 => std_logic_vector(to_unsigned( 35 , 8)),
                         11 => std_logic_vector(to_unsigned( 12 , 8)),
                         12 => std_logic_vector(to_unsigned( 254 , 8)),
                         13 => std_logic_vector(to_unsigned( 215 , 8)),
                         14 => std_logic_vector(to_unsigned( 78 , 8)),
                         15 => std_logic_vector(to_unsigned( 211 , 8)),
                         16 => std_logic_vector(to_unsigned( 121 , 8)),
                         17 => std_logic_vector(to_unsigned( 0 , 8)),
                         18 => std_logic_vector(to_unsigned( 0 , 8)),
                        others => (others =>'0'));


    -- Maschera di output = 00000011
    assert RAM(19) = std_logic_vector(to_unsigned( 3 , 8)) report "TEST FALLITO" severity failure;
```

8) reset_ricevuto

Per gli ultimi due casi di test, viene solamente fornita una porzione del process "test":

```
test : process is
begin
    wait for 100 ns;
    wait for c_CLOCK_PERIOD;
    tb_rst <= '1';
    wait for c_CLOCK_PERIOD;
    tb_rst <= '0';
    wait for c_CLOCK_PERIOD;
    tb_start <= '1';
    wait for c_CLOCK_PERIOD;
    wait for 500 ns;
    wait for c_CLOCK_PERIOD;
    tb_rst <= '1';
    wait for c_CLOCK_PERIOD;
    tb_rst <= '0';
    wait for c_CLOCK_PERIOD;
    tb_start <= '1';
    wait for c_CLOCK_PERIOD;
    wait until tb_done = '1';
    wait for c_CLOCK_PERIOD;
    tb_start <= '0';
    wait until tb_done = '0';
```

9) start_non_abbassato

```
test : process is
begin
    wait for 100 ns;
    wait for c_CLOCK_PERIOD;
    tb_rst <= '1';
    wait for c_CLOCK_PERIOD;
    tb_rst <= '0';
    wait for c_CLOCK_PERIOD;
    tb_start <= '1';
    wait for c_CLOCK_PERIOD;
    wait until tb_done = '1';
    wait for c_CLOCK_PERIOD;
    --tb_start <= '0';
    wait until tb_done = '0';
```