15 Marzo 2019

# Prova finale ( Progetto di Reti Logiche )

Il seguente progetto è stato realizzato da :

**Lunardi Emanuele** matricola: **868046** codice persona: **10567285**

**Loria Luca** matricola: **866939** codice persona: **10555747**

# 1. Introduzione :

Lo scopo del progetto è la realizzazione di un componente descritto in linguaggio VHDL che si interfaccia con una memoria RAM per leggere le coordinate in uno spazio bidimensionale di:

- un punto di riferimento;
- le coordinate di otto centroidi;
- una stringa di otto bit che rappresenta la maschera di ingresso.

Il compito del componente è quello di trovare il centroide che si trova più vicino in termini di Manhattan distance al punto di riferimento (possono esserci anche più centroidi equidistanti dal punto dato). Inoltre, viene analizzata la maschera d'ingresso per definire quali centroidi devono essere esaminati: se l'i-esimo punto della maschera è posto a 1 il centroide i-esimo dovrà essere preso in considerazione, altrimenti no. Dopo aver processato i centroidi validi, il componente scriverà sulla RAM la maschera di uscita una stringa di otto bit, in cui l'i-esimo bit è posto a 1 se l'i-esimo centroide è il più vicino al punto di riferimento.

# 2. Specifica del componente e scelte progettuali:

## 2.1 Entity presenti

L'approccio seguito per svolgere questo progetto è l'utilizzo di più entità collegate fra loro. Il componente descritto è composto da

- un'entità principale chiamata "project_reti_logiche" che si interfaccia con la memoria RAM;
- tre entità che svolgono la funzione di registri e da altre tre entità che ci occupano di fare i calcoli al fine di calcolare la Manhattan Distance. I registri sono rappresentati dalle entità : "Register8Bit" , "Register9Bit" e "Register4Bit".

L'entità principale è composta da un contatore a 5 bit e da una macchina a stati finiti: il contatore genera gli indirizzi su cui si vogliono leggere i dati della memoria ed è composto da 20 stati così da coprire tutti gli indirizzi utili (da 0 a 19).
Il segnale "o_address" , composto da 16 bit e che comunica l'indirizzo da cui leggere alla memoria , sarà ottenuto dalla concatenazione di undici zeri (nel progetto è stata definita una costante "zero_vector") e l'uscita del contatore composto da 5 bit (nel progetto chiamata "current_address").

La macchina a stati finiti , invece , governa lo svolgersi di tutte le operazioni comandando con opportuni segnali di enable, i registri, il contatore e i circuiti di calcolo  descritti nelle entità “CalculateDiffY” , “CalculateDistance” e “MinCheck”.
Il compito dell’entità “CalculateDiffY” è il calcolo della differenza tra la coordinata Y del punto di riferimento e la coordinata Y del centroide attualmente in esame. “CalculateDistance” calcola invece la somma tra la differenza delle Y, calcolata dall’ entità precedente, e la differenza della coordinata X del punto di riferimento e la coordinata X del centroide in esame. Infine l’entità “MinCheck” confronta la somma calcolata dall’entità precedente con il valore minimo di somma attuale(memorizzato in un apposito registro).  Nel caso in cui si rileva un nuovo valore minimo, l’entity provvede a modificare il registro del minimo e il relativo bit del vettore che verrà scritto alla fine nella memoria RAM.

**Importante** : la descrizione dei processi contenuti in queste ultime tre entità avverrà negli stati(che hanno lo stesso nome della relativa entità) che compongono la FSM.

## 2.2 Entity “project_reti_logiche”:

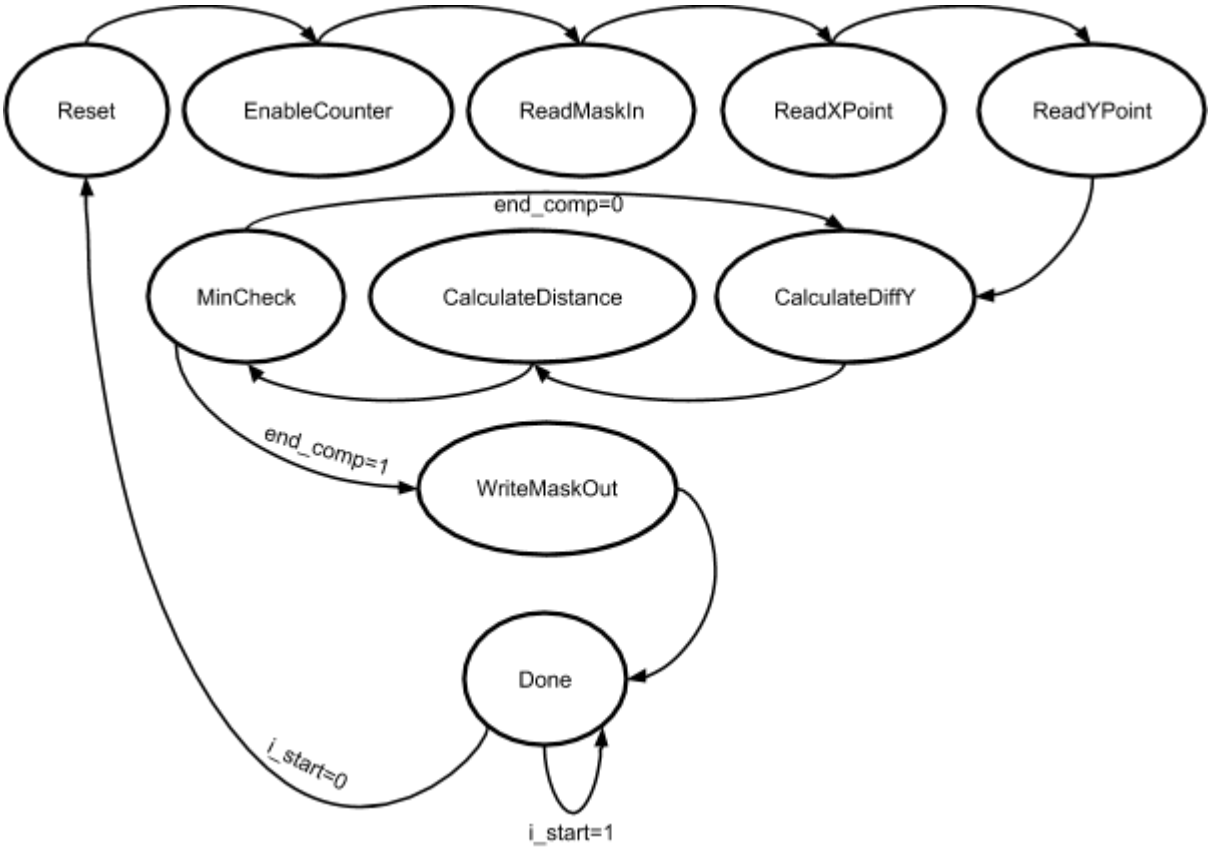
Questa entità è quella che si occupa di interfacciarsi con la memoria RAM attraverso l’implementazione di una FSM spiegata dettagliatamente nel paragrafo 2.2.1 e di un contatore a 5 bit spiegato nel paragrafo 2.2.2. Essa presenta la seguente interfaccia:

Codice **VHDL**:

```
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;
```

### 2.2.1 Descrizione della FSM  e dell'algoritmo implementato per calcolare la maschera di uscita

L'FSM ha il compito di stabilire quali operazioni vanno svolte e in quale ordine in ogni momento della computazione. La struttura è schematizzabile nel seguente modo:




Lo stato attuale della macchina a stati finiti è stato salvato nel segnale "current_state" mentre lo stato successivo in "next_state". La funzione di stato prossimo è stata implementata nel processo "state_reg_FSM" in cui è presente un reset asincrono che, se abilitato, porta la macchina nello stato denominato appunto **Reset**. Il segnale "current_state" viene posto uguale a quello di "next_state" solamente sul fronte di salita del clock e se "i_start" è posto a 1 , dunque finchè la RAM non abilita il segnale di inizio computazione, la macchina rimane nello stato di **Reset**. Dopo aver salvato le informazioni relative alla maschera di ingresso e le coordinate del punto di riferimento ,  si comincia ad esaminare i centroidi partendo dall'ottavo fino ad arrivare all primo. Per tenere traccia di ciò verrà usato un unsigned chiamato **index**, memorizzato in un apposito registro. Esso conterrà sempre il numero del centroide che attualmente si sta considerando: sarà inizializzato a "0111" che corrisponde all'ottavo punto e verrà decrementato fino ad arrivare al valore "0000" che corrisponderà al primo punto,  cioè l'ultimo da esaminare. Segue una descrizione dettagliata degli stati della FSM.

1. **Reset**: come indica il nome è lo stato di reset della macchina. E' uno stato di idle in cui si aspetta l'inizio di una nuova computazione, non appena "i_start" passa a 1  si  procede allo stato successivo per iniziare la computazione;

2. **EnableCounter**: stato di preambolo, permette di sincronizzare tutte le operazioni successive con il clock. Un'altra sua funzione è di abilitare il contatore che genera gli indirizzi di memoria della RAM e di abilitare il salvataggio della **mask_in** nel relativo registro (en_mask_in=1) che avverrà al ciclo di clock successivo;

3. **ReadMaskIn**: nel ciclo di clock in cui questo stato è attivo avviene la lettura e il salvataggio della mask_in nell'apposito registro abilitato nello stato precedente, dato che su "i_data" è presente il valore della maschera in ingresso (letto dall'indirizzo 0 della RAM). Inoltre abilita il salvataggio della coordinata x del punto (**x_point**) nel relativo registro (en_x_point=1) che avverrà al ciclo di clock successivo;

4. **ReadXPoint**: nel ciclo di clock in cui questo stato è attivo avviene la lettura e il salvataggio del x_point nel relativo registro, dato che su "i_data" è presente il valore della coordinata X del punto di riferimento (letta all'indirizzo 17 della RAM). Inoltre abilita il salvataggio della coordinata y del punto (**y_point**) nel relativo registro (en_y_point=1) che avverrà al ciclo di clock successivo;

5. **ReadYPoint**: nel ciclo di clock in cui questo stato è attivo avviene la lettura e il salvataggio del y_point nel relativo registro , dato che su "i_data" è presente il valore della coordinata Y del punto di riferimento (letta all'indirizzo 18 della RAM). Il segnale en_diffY=1 consente di abilitare il circuito descritto nella entity "CalculateDiffY" che lavorerà nel prossimo ciclo di clock;

6. **CalculateDiffY**: in questo stato l'entità "CalculateDiffY" è abilitata e calcola la differenza tra il punto di riferimento e il centroide attualmente in esame. Si abilita inoltre il registro (**sub_register**) per memorizzare tale risultato che verrà utilizzato nel prossimo stato. Infine si setta il segnale en_add_diff=1 per abilitare il circuito descritto nella entity "CalculateDistance" che lavorerà nel prossimo ciclo di clock;

   Descrizione del processo "**diff_process**" descritto nell'entity "CalculateDiffY":
   In questo processo, i_data contiene sempre la coordinata y dell'i-esimo centroide di cui si sta calcolando la Manhattan distance. Il procedimento del calcolo è così schematizzabile:
   a. Si controlla se il centroide in analisi (**index**-esimo) è abilitato dalla mask_in. Ciò avviene verificando la maschera di ingresso attraverso la funzione to_integer applicata a **index.** Se l'index-esimo bit della mask_in è a 0 si procede allo stato successivo senza effettuare operazioni (la maschera indica che il centroide selezionato non è da prendere in considerazione), mentre se l'index-esimo bit della mask_in è a 1 si procede normalmente;
   b. Attraverso un comparatore si stabilisce quale tra i_data e y_point è il valore maggiore. Questa operazione viene effettuata al fine di evitare di ottenere valori negativi come risultato della sottrazione tra i valori delle coordinate y;
   c. Si calcola la differenza tra il valore massimo appena stabilito e l'altro valore e la si salva nel registro suby;

7. **CalculateDistance**: in questo stato l'entità "CalculateDistance" calcola la somma tra: la differenza delle coordinate y, calcolata allo stato precedente e salvata nell'apposito registro, e la differenza delle coordinate x (Manhattan distance). Questa somma verrà memorizzata nel registro result_register che ha come ingresso out_result_tmp e uscita out_result.
   Si pone il segnale min_en=1 per abilitare il circuito descritto nell'entità "MinCheck" che lavorerà nel prossimo ciclo di clock. Viene infine disabilitato temporaneamente il contatore degli indirizzi (en_counter=0) per mantenere sincronizzato il contatore degli indirizzi con la FSM.

   Descrizione del processo "**sum_process**" descritto nell'entità "CalculateDistance":
   In questo processo, i_data contiene sempre la coordinata x dell'i-esimo centroide di cui si sta calcolando la Manhattan distance.Il procedimento è così schematizzabile:
   a. Si verifica se il centroide in analisi (**index**-esimo) è abilitato dalla mask_in. In caso negativo, al risultato della somma vengono messi tutti i 9 bit a 1, ovvero un valore impossibile da ottenere che sarà sempre maggiore del minimo assoluto(vedi punto a.iii dello stato MinCheck) . In caso positivo si procede normalmente;
   b. Attraverso un comparatore si stabilisce quale tra i_data e x_point è il valore maggiore. Questa operazione viene effettuata al fine di evitare di ottenere valori negativi come risultato della differenza tra le coordinate x;
   c. Si calcola la differenza tra il valore massimo appena stabilito e l'altro valore. Il risultato viene poi sommato al valore **suby** (ovvero la differenza delle coordinate y del medesimo centroide) memorizzato nel registro sub_register nello stato precedente , completando il calcolo della Manhattan distance;

8. **MinCheck**:è lo stato più importante della FSM. In questo stato l'entità "MinCheck" confronta il valore della Manhattan distance contenuta in out_result (ovvero la distanza dal centroide numero **index**) con il valore del minimo assoluto delle distanze calcolato fino ad ora (min).
   Si riabilita il contatore (en_counter = 1) così da avere i dati pronti nel prossimo stato della FSM.
   Se il centroide appena analizzato era l'ultimo della mask (**index**=0 segnalato dal segnale "end_comp" = 1) allora si procede allo stato successivo, altrimenti si ripetono tutte le operazioni sul centroide successivo a partire dallo stato CalculateDiffY (6) riabilitando il circuito dell'entità "CalculateDiffY" con il segnale en_diffY=1.

   Descrizione del processo "**min_process**" descritto nell'entità "MinCheck":
   Inizialmente il minimo è inizializzato con il maggior valore che può assumere , cioè 510. Il procedimento è così schematizzabile:
   a. Attraverso un comparatore, vengono confrontati **out_result** e **min**:
      i. se min > out_result , allora si è trovato un nuovo minimo che, oltre a rimpiazzare il precedente **min**, imposta tutti i valori della maschera d'uscita (**output_data**) a 0 tranne quello in analisi (**index**-esimo) a 1;
      ii. se min =out_result , si è trovato un centroide equidistante al min assoluto: viene impostato a 1 il relativo bit (**index**-esimo) sulla maschera d'uscita (**output_data**) mantenendo inalterato il contenuto degli altri bit;
      iii. se min < out_result_tmp, il centroide è più distante del minimo assoluto e quindi viene impostato a 0 il relativo bit sulla maschera d'uscita (**output_data**)

mantenendo inalterato il contenuto degli altri bit. Notare bene che se il punto in esame non è da esaminare a causa di uno 0 nella maschera di ingresso si entra sempre in questa condizione dato che nello stato precedente si è posto out_result a “111111111” (511) che sarà sicuramente maggiore di “111111110”(510) che è il valore più grande ammissibile dal minimo.

b. Viene decrementato l’**index** della mask, passando così ad analizzare il centroide successivo. Se l’index è 0 (ultimo punto ) esso non si decrementa e si pone a 1 il segnale “end_comp”.

9. **WriteMaskOut**: in questa fase si procede a copiare la maschera d’uscita. Inoltre viene settato il segnale o_we=1 così da comunicare la volontà di scrivere in memoria il risultato. Il contatore degli indirizzi viene fermato (en_counter=0) poiché si è giunti all’ultimo indirizzo (19).

10. **Done:** è lo stato finale della macchina che serve a comunicare alla memoria RAM la scrittura del risultato all’indirizzo prestabilito settando il segnale **o_done** a 1. Esso viene riportato a 0 solo quando la RAM porta **i_start** a 0 indicando la fine della computazione . Quando i_start viene posto a 0 si ritorna nello stato Reset in attesa di una nuova computazione.

### 2.2.2 Descrizione del contatore a 5 bit

Lo scopo di questo contatore è la generazione degli indirizzi che verranno scritti sul segnale “o_address”. Esso consentirà di leggere dalla memoria dall’indirizzo richiesto così da fornire il contenuto sul segnale “i_data”. Il contatore è composto da 20 stati (nel progetto indicati s0 , s1, ... fino a s19 ) , in cui lo stato corrente verrà memorizzato nel segnale “current_counter” mentre lo stato successivo nel segnale “next_counter”. La funzione di stato prossimo è implementata nel processo “state_reg_counter” in cui è presente un reset asincrono che, se abilitato, porta il contatore nello stato di reset denominato s0. Lo stato corrente diventerà lo stato futuro solo se il segnale di enable del contatore (“en_counter”) è alto. Il cambio di stato avviene sul fronte di discesa del clock perchè così facendo la FSM e il contatore rimangono sincronizzati: l’indirizzo nuovo prodotto dal contatore verrà settato sul fronte di discesa del clock, dunque la RAM fornirà il valore contenuto in tale indirizzo su “i_data” sul fronte di salita del clock in cui ci sarà il nuovo stato della FSM.
L’uscita del contatore è salvata sul segnale “current_counter” che, come scritto precedentemente, verrà concatenato a 11 zeri per ottenere l’indirizzo desiderato da scrivere su “o_address”. Il ciclo di conteggio avviene nel seguente modo : dopo aver letto la maschera di ingresso ( indirizzo 0) , la coordinata X del punto dato ( indirizzo 17) e la coordinata Y del punto dato (indirizzo 18), si inizierà a leggere la coordinata Y dell’ottavo centroide(posta all’indirizzo 16) seguita dalla coordinata X dell’ottavo centroide( indirizzo 15) fino ad arrivare al primo ed ultimo centroide.L’ultimo indirizzo prodotto sarà 19 , cioè quello in cui si scriverà il valore della maschera di uscita.
Quindi, l’ordine degli indirizzi generati è 0 , 17 , 18 , 16 , 15 , 14 , 13 , 12 , 11 , 10 , 9 , 8 , 7, 6 , 5 , 4 , 3 , 2 , 1 , 19. Ogni indirizzo corrisponde ad uno stato del contatore.

## 2.3 Entity “Register8Bit”:

Questa entità serve per descrivere un registro a 8 bit e la sua  interfaccia è la seguente:

Codice **VHDL**:

```
entity Register8 is
   port (
            inputR8 : in std_logic_vector ( 7 downto 0) ;
            clkR8 : in std_logic ;
            rstR8 : in std_logic;
            enableR8 : in std_logic;
            outputR8 : out std_logic_vector ( 7 downto 0));
end Register8;
```

Il  funzionamento del registro è descritto all’interno di un processo chiamato “save_process” in cui viene gestito un reset asincrono : se il segnale di reset è alto, allora l’uscita assumerà un valore di default di tutti zeri. Se invece il segnale di enable è alto e si verifica un fronte di salita del clock ,  si salverà sul vettore di uscita il vettore in ingresso.
Questo registro a 8 bit serve per memorizzare:

1) Il vettore contenente la maschera di ingresso(**mask_in**) che verrà letto dalla RAM all’Indirizzo 0.
2) Il vettore contenente la coordinata X del punto dato all’indirizzo 17 (**x_point**).
3) Il vettore contenente la coordinata Y del punto dato all’indirizzo 18 (**y_point**).
4) Un vettore che conterrà il risultato temporaneo della maschera di uscita (**output_data**). Quando saranno stati esaminati tutti i centroidi questo vettore verrà scritto su “o_data” che verrà poi scritto in memoria all’indirizzo 19.

Il segnale di enable è fondamentale dato che  nella entity principale ci sarà una port map che assegnerà l’input dei registri ( che memorizzano la maschera di ingresso , x_point e y_point )  al segnale “i_data” e dunque ,  è necessario che il registro memorizzi il valore solamente se su “i_data” è presente il valore corretto . La FSM descritta nella entity principale si occuperà di abilitare il segnale di enable nel momento opportuno ( esempio : metterà il segnale di enable alto al registro che memorizzerà la maschera di ingresso solo se su “i_data” ci sarà effettivamente il valore della maschera di ingresso).
Le istanze di tale registro sono nella entity principale e sono le seguenti : “mask_register “ , “x_register” , “y_register” e “data_register”.

## 2.4 Entity “Register9Bit”:

Questa entità viene impiegata per descrivere un registro a 9 bit. La sua interfaccia è composta da:
Codice **VHDL**:

```
entity Register9 is
   port (
            inputR9 : in std_logic_vector ( 8 downto 0) ;
            clkR9 : in std_logic ;
            rstR9 : in std_logic;
            outputR9 : out std_logic_vector ( 8 downto 0));
end Register9;
```

Anche qui il funzionamento del registro è descritto in un processo chiamato ”save_process”. L’entity implementa un reset asincrono. Questo registro assegna l’ingresso all’uscita solamente quando si verifica un fronte di discesa del clock , a differenza del registro ad 8 bit che funzionava sul fronte di salita. E’ stata effettuata questa scelta in quanto la FSM descritta nella entity principale lavora sul fronte di salita del clock e, per avere i dati pronti al successivo ciclo di clock, deve memorizzare i dati nel registro prima che si presenti il successivo rising edge del clock.
Questo registro serve a memorizzare :

1) Il risultato della differenza tra la coordinata Y del  punto di riferimento e la coordinata Y del centroide che si sta esaminando (**temp_sub**).
2) Il risultato della somma tra **suby** e la differenza tra la coordinata X del  punto di riferimento e la coordinata X del centroide che si sta esaminando (**out_result_tmp**).
3) Il valore del minimo assoluto: è il valore minimo della somma descritta al punto precedente che per ora si è trovato (**min_tmp**).

Le istanze di tale registro sono nella entity principale e sono le seguenti : “sub_register” , “result_register” e “min_register”.

## 2.5 Entity “Register4Bit”:

Questa entità viene impiegata per descrivere un registro a 4 bit. La sua interfaccia è composta da:

Codice **VHDL**:

```
entity Register4Bit is
   port (
            inputR4 : in unsigned ( 3 downto 0) ;
            clkR4 : in std_logic ;
            rstR4 : in std_logic;
            outputR4 : out unsigned ( 3 downto 0));
end Register4Bit;
```

Questa entità è utile per memorizzare l'indice (**index**) che terrà conto di quale centroide si sta attualmente esaminando.
Dato che l'indice dovrà essere utilizzato come "cursore" della maschera di ingresso sarà necessario memorizzare questo indice come un unsigned al fine di applicare la funzione to_integer e ispezionare il relativo bit sulla maschera di ingresso.
Il funzionamento del registro è descritto nel processo "save_process" dove è presente un reset asincrono che se abilitato pone l'indice uguale a "0111" che corrisponde all'ottavo centroide , cioè il primo punto che si dovrà esaminare. Il salvataggio , come per il registro a 9 bit precedentemente spiegato, avviene sul fronte di discesa del clock. L'istanza di tale registro si chiama "index_register" e si trova nella entity principale.

# 3.Test Bench effettuati

Per verificare il corretto funzionamento del componente sono stati creati diversi test bench. Ogni test è pensato per stressare il componente in condizioni di normale funzionamento oppure in condizioni al contorno. E' importante specificare che il componente ha risposto in maniera positiva a tutti i test bench a cui è stato sottoposto, in **behavioural**, **post-synthesis functional** simulation e in **post_synthesis timing** simulation. Tutti i test sono stati svolti abilitando il segnale di reset prima di porre a 1 il segnale di start.
Di seguito la lista di test con i risultati attesi:

1. **Test Bench di esempio**: è il test fornito dal professore, che verifica la correttezza del componente in una situazione di normale funzionamento.
   **RAM:**

```
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'));
```

**Risultato atteso:** `assert RAM(19) = std_logic_vector(to_unsigned( 17 , 8))`

2. **Mask_in a “00000000”**: in questo test, la maschera di ingresso è formata da soli zeri, quindi ci si aspetta che il risultato sia anch’esso una maschera di soli zeri. Questo test è un caso limite che indica che nessun centroide dovrà essere preso in considerazione.
   **RAM:**

```
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'));
```

**Risultato atteso:** `assert RAM(19) = std_logic_vector(to_unsigned( 0 , 8))`

3. **Mask_in a “11111111”:** in questo test, la maschera di ingresso è formata da soli uno e tutti i centroidi coincidono (sono quindi tutti equidistanti dal punto). Ci si aspetta di ottenere una maschera d’uscita di soli uno.
   **RAM:**

```
signal RAM: ram_type := (0 => std_logic_vector(to_unsigned( 255 , 8)),
                         1 => std_logic_vector(to_unsigned( 111 , 8)),
                         2 => std_logic_vector(to_unsigned( 32 , 8)),
                         3 => std_logic_vector(to_unsigned( 111 , 8)),
                         4 => std_logic_vector(to_unsigned( 32 , 8)),
```

```
                    5 => std_logic_vector(to_unsigned( 111 , 8)),
                    6 => std_logic_vector(to_unsigned( 32 , 8)),
                    7 => std_logic_vector(to_unsigned( 111, 8)),
                    8 => std_logic_vector(to_unsigned( 32 , 8)),
                    9 => std_logic_vector(to_unsigned( 111 , 8)),
                    10 => std_logic_vector(to_unsigned( 32 , 8)),
                    11 => std_logic_vector(to_unsigned( 111 , 8)),
                    12 => std_logic_vector(to_unsigned( 32 , 8)),
                    13 => std_logic_vector(to_unsigned( 111 , 8)),
                    14 => std_logic_vector(to_unsigned( 32 , 8)),
                    15 => std_logic_vector(to_unsigned( 111 , 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'));
```

**Risultato atteso:** `assert RAM(19) = std_logic_vector(to_unsigned( 255 , 8))`

4. **Mask_in con un solo 1**: in questo test, la maschera d'ingresso presenta un solo 1 (quindi un solo centroide da esaminare). Ci si aspetta di ottenere una maschera d'uscita con un solo 1 nella stessa posizione dell'1 nella maschera di ingresso.
   **RAM:**

```
signal RAM: ram_type := (0 => std_logic_vector(to_unsigned( 16 , 8)),
                    1 => std_logic_vector(to_unsigned( 111 , 8)),
                    2 => std_logic_vector(to_unsigned( 32 , 8)),
                    3 => std_logic_vector(to_unsigned( 111 , 8)),
                    4 => std_logic_vector(to_unsigned( 32 , 8)),
                    5 => std_logic_vector(to_unsigned( 111 , 8)),
                    6 => std_logic_vector(to_unsigned( 32 , 8)),
                    7 => std_logic_vector(to_unsigned( 111, 8)),
                    8 => std_logic_vector(to_unsigned( 32 , 8)),
                    9 => std_logic_vector(to_unsigned( 111 , 8)),
                    10 => std_logic_vector(to_unsigned( 32 , 8)),
                    11 => std_logic_vector(to_unsigned( 111 , 8)),
                    12 => std_logic_vector(to_unsigned( 32 , 8)),
                    13 => std_logic_vector(to_unsigned( 111 , 8)),
                    14 => std_logic_vector(to_unsigned( 32 , 8)),
                    15 => std_logic_vector(to_unsigned( 111 , 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'));
```

**Risultato atteso:** `assert RAM(19) = std_logic_vector(to_unsigned( 16 , 8))`

5. **Punto in 0,0 e centroidi in 255,255:** in questo test, ci si è concentrati sul corretto funzionamento nel caso limite del punto (0,0) e di tutti i centroidi (255,255) posizionati a distanza massima. La maschera di ingresso è stata scelta in maniera casuale. La maschera di uscita deve essere coincidente con la maschera di ingresso.
   **RAM:**

```
signal RAM: ram_type := (0 => std_logic_vector(to_unsigned( 139 , 8)),
                         1 => std_logic_vector(to_unsigned( 255 , 8)),
                         2 => std_logic_vector(to_unsigned( 255 , 8)),
                         3 => std_logic_vector(to_unsigned( 255 , 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( 255 , 8)),
                         8 => std_logic_vector(to_unsigned( 255 , 8)),
                         9 => std_logic_vector(to_unsigned( 255 , 8)),
                         10 => std_logic_vector(to_unsigned( 255 , 8)),
                         11 => std_logic_vector(to_unsigned( 255 , 8)),
                         12 => std_logic_vector(to_unsigned( 255 , 8)),
                         13 => std_logic_vector(to_unsigned( 255 , 8)),
                         14 => std_logic_vector(to_unsigned( 255 , 8)),
                         15 => std_logic_vector(to_unsigned( 255 , 8)),
                         16 => std_logic_vector(to_unsigned( 255 , 8)),
                         17 => std_logic_vector(to_unsigned( 0 , 8)),
                         18 => std_logic_vector(to_unsigned( 0 , 8)),
        others => (others =>'0'));
```

**Risultato atteso:** `assert RAM(19) = std_logic_vector(to_unsigned( 139 , 8))`

6. **Centroidi tutti equidistanti dal punto, tranne l'ultimo più vicino**: in questo test, i centroidi da 7 a 1 sono stati posizionati alla stessa distanza dal punto, mentre il centroide 0 è stata collocato più vicino al punto. La maschera d'uscita deve essere la seguente: "00000001".
   **RAM:**

```
signal RAM: ram_type := (0 => std_logic_vector(to_unsigned( 255 , 8)),
                         1 => std_logic_vector(to_unsigned( 1 , 8)),
                         2 => std_logic_vector(to_unsigned( 1 , 8)),
                         3 => std_logic_vector(to_unsigned( 255 , 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( 255 , 8)),
                    8 => std_logic_vector(to_unsigned( 255 , 8)),
                    9 => std_logic_vector(to_unsigned( 255 , 8)),
                    10 => std_logic_vector(to_unsigned( 255 , 8)),
                    11 => std_logic_vector(to_unsigned( 255 , 8)),
                    12 => std_logic_vector(to_unsigned( 255 , 8)),
                    13 => std_logic_vector(to_unsigned( 255 , 8)),
                    14 => std_logic_vector(to_unsigned( 255 , 8)),
                    15 => std_logic_vector(to_unsigned( 255 , 8)),
                    16 => std_logic_vector(to_unsigned( 255 , 8)),
                    17 => std_logic_vector(to_unsigned( 0 , 8)),
                    18 => std_logic_vector(to_unsigned( 0 , 8)),
        others => (others =>'0'));
```

**Risultato atteso:** `assert RAM(19) = std_logic_vector(to_unsigned( 1 , 8))`

7. **Centroidi equidistanti dal punto uno si e uno no**: in questo test, i centroidi dispari (7,5,3,1) sono posti equidistanti dal punto mentre i rimanenti sono collocato ad una distanza maggiore. Ci si aspetta quindi una maschera d'uscita così formata: "10101010".
   **RAM:**

```
signal RAM: ram_type := (0 => std_logic_vector(to_unsigned( 255 , 8)),
                    1 => std_logic_vector(to_unsigned( 1 , 8)),
                    2 => std_logic_vector(to_unsigned( 1 , 8)),
                    3 => std_logic_vector(to_unsigned( 255 , 8)),
                    4 => std_logic_vector(to_unsigned( 255 , 8)),
                    5 => std_logic_vector(to_unsigned( 1 , 8)),
                    6 => std_logic_vector(to_unsigned( 1 , 8)),
                    7 => std_logic_vector(to_unsigned( 255, 8)),
                    8 => std_logic_vector(to_unsigned( 255 , 8)),
                    9 => std_logic_vector(to_unsigned( 1 , 8)),
                    10 => std_logic_vector(to_unsigned( 1, 8)),
                    11 => std_logic_vector(to_unsigned( 255 , 8)),
                    12 => std_logic_vector(to_unsigned( 255 , 8)),
                    13 => std_logic_vector(to_unsigned( 1 , 8)),
                    14 => std_logic_vector(to_unsigned( 1 , 8)),
                    15 => std_logic_vector(to_unsigned( 255 , 8)),
                    16 => std_logic_vector(to_unsigned( 255 , 8)),
                    17 => std_logic_vector(to_unsigned( 0 , 8)),
                    18 => std_logic_vector(to_unsigned( 0 , 8)),
```

```
                others => (others =>'0'));
```

**Risultato atteso:** `assert RAM(19) = std_logic_vector(to_unsigned( 85 , 8))`

# 4. Risultati della sintesi

La sintesi del componente descritto funziona correttamente e , come scritto in precedenza, funzionano sia la post-synthesis functional simulation che la post-synthesis timing simulation con tutti i test bench prodotti.
L'unico warning in post-sintesi è "o_address driven by constant 0" derivante dal fatto che il vettore che manda gli indirizzi in memoria (o_address) è composto da 16 bit e i suoi primi 11 sono sempre 0. Ciò è corretto dato che si usa una porzione piccolissima della RAM confronto a quella realmente a disposizione (si usano solo 20 indirizzi con 16 bit).

## 4.1 Performance:

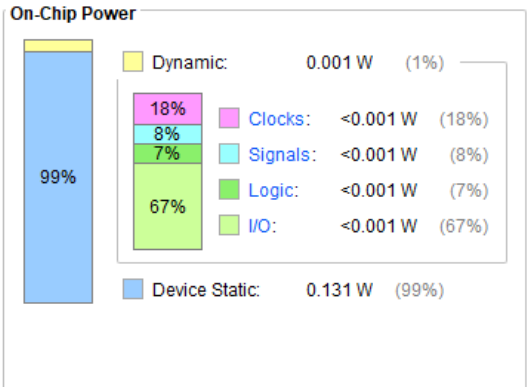
Il numero di componenti della FPGA utilizzati per la sintesi è il seguente:

| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| LUT | 88 | 133800 | 0.07 |
| FF | 108 | 267600 | 0.04 |
| IO | 38 | 285 | 13.33 |
| BUFG | 1 | 32 | 3.13 |

Il consumo di energia del componente implementato è il seguente:

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

| | |
|---|---|
| **Total On-Chip Power:** | **0.131 W** |
| **Design Power Budget:** | **Not Specified** |
| **Power Budget Margin:** | **N/A** |
| **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 |




Infine, è stato generato un file di constraint (clock a 100 ns) per analizzare le prestazioni del componente e il risultato è il seguente:

| Setup | | Hold | | Pulse Width | |
|---|---|---|---|---|---|
| Worst Negative Slack (WNS): | 45,149 ns | Worst Hold Slack (WHS): | 0,118 ns | Worst Pulse Width Slack (WPWS): | 49,500 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: | 174 | Total Number of Endpoints: | 174 | Total Number of Endpoints: | 109 |

**All user specified timing constraints are met.**