Prova finale (Progetto di Reti Logiche)

Adriano Mundo [10524163 – 844112]

Dario Miceli [10503043 – 843733]

Prof. Ferrandi

1 Introduzione

Lo scopo del progetto consiste nella realizzazione di un componente HW descritto in linguaggio VHDL, e sintetizzato con il tool XILINK VIVADO su una FPGA target (xc7a200tfbg484-1). Il componente, data l'immagine in scala di grigi, calcola l'area del rettangolo minimo che circoscrive totalmente tale figura.

L'interfaccia del progetto realizzato è la seguente:

```
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;
```

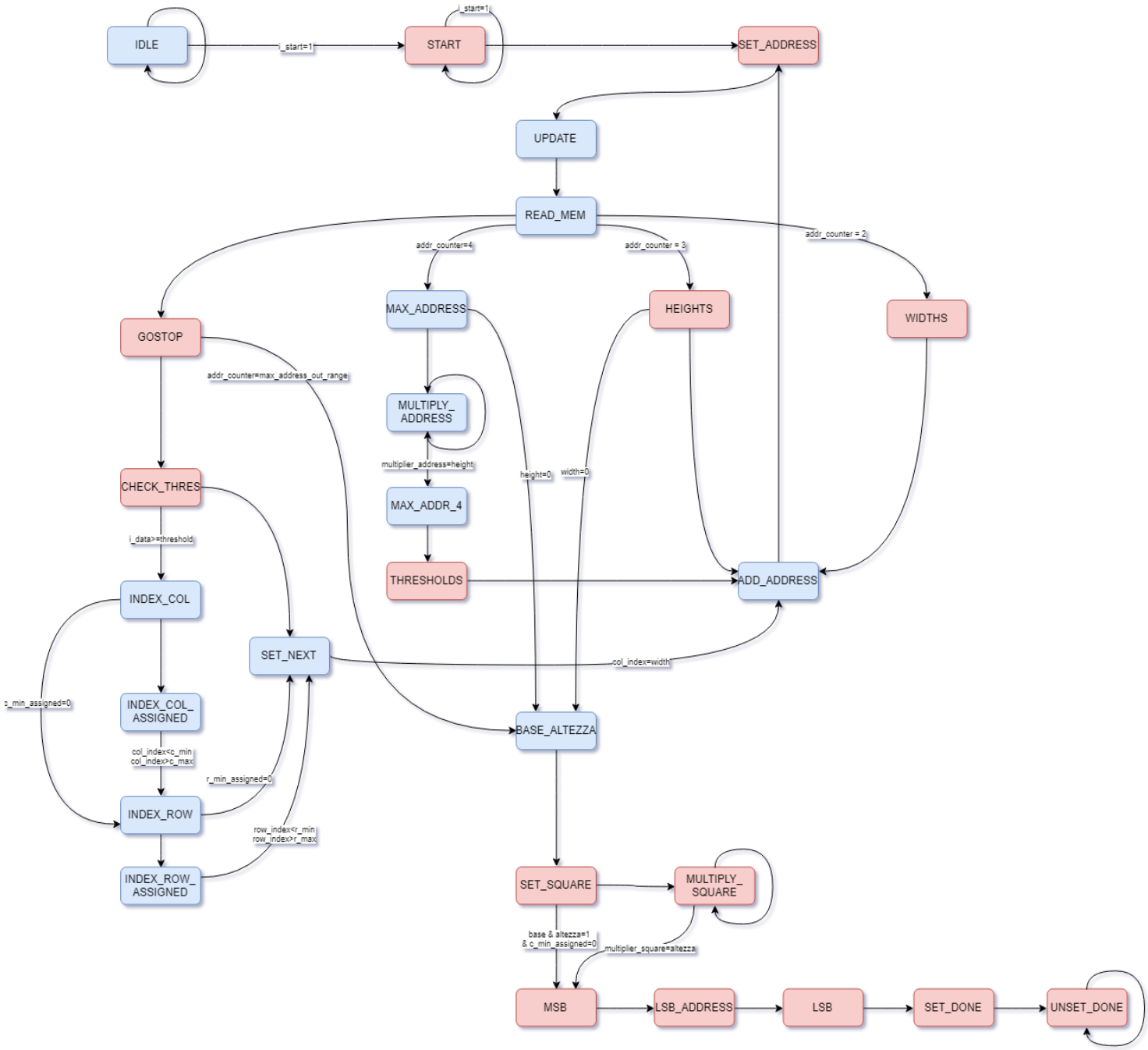
Il componente si interfaccia con una memoria RAM, nel quale è descritta l'immagine sulla quale effettuare l'elaborazione. La memoria è suddivisa in due parti:

- *header*: tre campi di dimensione 1 byte, gli indirizzi *0x0002, 0x0003, 0x0004* rappresentano rispettivamente il numero di colonne, numero di righe, ed il valore di soglia.
- *contenuto*: codificato come una matrice, di cui ogni elemento rappresenta un pixel, codificato con un byte; esso appartiene all'immagine nel caso in cui abbia un valore maggiore o uguale alla soglia contenuta nell' *header*. Il *contenuto* è presente in memoria a partire dall'indirizzo *0x0005.*

Il modulo parte nell'elaborazione alla ricezione del segnale *i_ start,* ed al termine alza ad 1 il valore del segnale di *o_ done* per comunicare alla memoria la fine dell'elaborazione. Dopo tale settaggio, il segnale di *o_ done* è riportato a 0, in attesa di un nuovo segnale *i_ rst* che riporti la macchina nello stato iniziale e ricominci con una nuova elaborazione.

2 Component Design

Data la natura del problema presentato, e la presenza di un segnale di *i_ rst* ed uno di *i_ start,* il design più intuitivo e semplice da implementare per ottenere il risultato finale è stato quello di una Finite State Machine (FSM) con una *architecture behavioral* costituita da un solo *process* con una sensivity list di due segnali, il clock ed il reset. Le motivazioni di tale scelta implementativa sono argomentate nel paragrafo dell'ottimizzazione. Di seguito viene fornito un diagramma a stati di alto livello che rappresenta la macchina implementata.



Per comprendere meglio il diagramma, adesso verrà spiegato in dettaglio ogni singolo stato con le sue operazioni ed il relativo scopo.

IDLE: lo stato iniziale della macchina, nel quale si resta in attesa del primo segnale che verrà alzato dalla memoria(*i_ start*), dopo che *i_ start* è stato portato a 1 per un ciclo di clock, viene riportato a 0 ed a quel punto il componente passa nello stato successivo per prepararsi all'elaborazione.

START: lo stato di start è quello che prepara il componente, settando il segnale di *o_ en* a 1 per poter comunicare con la memoria e *o_ we* a 0 per entrare nella modalità di lettura dalla memoria, dopo il quale passa nello stato che setta l'indirizzo in memoria.

ADD_ADDRESS: stato che aumenta il valore del segnale *addr_ counter*, usato per tenere conto dell'indirizzo corrente sul quale si stanno svolgendo le operazioni.

SET_ADDRESS: stato che setta in *o_ address,* quindi in memoria, il valore corrente dell'*addr_ counter* in modo che al ciclo di clock successivo si possa operare sul valore dell'indirizzo assegnato.

UPDATE: stato che attende un ciclo di clock in attesa dell'effettiva scrittura in memoria.

READ_MEM: stato che controlla il valore dell'*addr_ counter* corrente e decide in quale stato spostarsi per proseguire l'elaborazione.

WIDTHS: stato che legge il valore giunto in ingresso dalla memoria sul segnale *i_ data*, e lo memorizza nel segnale di *width,* necessario per memorizzare il numero di colonne.

HEIGHTS: stato che legge il valore di *i_ data* in ingresso e lo memorizza nel segnale *height* per memorizzare il numero di righe, e nel caso in cui il valore letto precedentemente su *width* sia zero precedentemente allora passa direttamente allo stato che inizia l'elaborazione per l'area, in caso contrario diminuisce il valore di *width* per gestire gli indici a partire da zero.

MAX_ADDRESS: stato che in primis controlla che il valore del numero di righe letto nel ciclo precedente sia uguale a zero ed in tal caso passa allo stato che inizia l'elaborazione per il calcolo dell'area, altrimenti prepara i segnali per il calcolo della condizione di stop, l'indirizzo massimo fino al quale deve proseguire la lettura della matrice.

MULTIPLY_ADDRESS: stato che simula moltiplicazione facendo un totale di N somme del numero di colonne, dove N è il numero di righe letto dalla memoria. Raggiunto il valore N cambia stato.

MAX_ADDR_4: stato in cui viene memorizzato il valore del segnale del massimo indirizzo raggiungibile più lo shift dato dai valori necessari a memorizzare l'msb, l'lsb ed i valori dati pre-elaborazione.

THRESHOLDS: stato in cui si legge il valore giunto in ingresso su *i_ data* e lo si memorizza nel segnale *threshold* che memorizza il valore di soglia, necessario per poter decidere se un pixel appartiene all'immagine o meno.

GOSTOP: stato che controlla se l'$addr\_\ counter$ ha raggiunto il valore massimo ed in tal caso aggiorna il valore di $o\_\ we$ ad 1 per entrare nella modalità scrittura, altrimenti passa allo stato di controllo del valore di soglia.

CHECK_THRES: stato che controlla se il valore giunto in ingresso su i_data appartiene all'immagine o meno confrontandolo con il valore del segnale $threshold$ memorizzato.

INDEX_COL: stato che controlla se un valore appartenente all'immagine è stato trovato o meno, nel caso non sia stato trovato assegna i valori di $c\_\ min,\ c\_\ max$ e passa allo stato di controllo delle righe, altrimenti passa allo stato che effettua il controllo successivo

INDEX_COL_ASSIGNED: nel caso un valore fosse già stato trovato, questo stato controlla se effettivamente il nuovo pixel appartenente all'immagine ha degli indici di colonna che siano minori o maggiori di quelli attualmente memorizzati in $c\_\ min$ e $c\_\ max$ e, se necessario ne aggiorna i valori.

INDEX_ROW: stato che controlla se un valore appartenente all'immagine è già stato trovato o meno, e nel caso non sia stato trovato assegna i valori di $r\_\ min,\ r\_\ max$ e passa allo stato che si occupa di andare avanti al prossimo pixel, altrimenti passa nello stato di controllo successivo per le righe.

INDEX_ROW_ASSIGNED: nel caso un valore fosse già stato trovato, questo stato controlla se il nuovo pixel ha degli indici di riga maggiori o minori di quelli salvati in $r\_\ min$ e $r\_\ max$ in quel momento, ed in caso ne aggiorna i valori.

SET_NEXT: stato che aggiorna i valori degli indici di riga e di colonna e passa allo stato di aggiornamento dell'$addr\_\ counter$.

BASE_ALTEZZA: stato che calcola la base e l'altezza con le formule riportate nel paragrafo dell'algoritmo.

SET_SQUARE: stato che aggiorna il valore dell'indirizzo a quello dell'msb, controlla il caso in cui l'area sia 0, altrimenti passa nello stato di gestione della moltiplicazione

MULTIPLY_SQUARE: : stato che simula moltiplicazione facendo un totale di N somme della base, dove N l'altezza del rettangolo minimo. Raggiunto il valore N cambia, si passa allo stato di scrittura in memoria.

MSB: stato che scrive in $o\_\ data$, e quindi in memoria il valore del bit più significativo dell'area del rettangolo minimo precedentemente calcolata.

LSB_ADDRESS: stato che scrive in $o\_\ address$ l'indirizzo in cui si scriverà in memoria il valore del bit meno significativo.

LSB: stato che scrive in $o\_\ data$, e quindi in memoria il valore del bit meno significativo dell'area del rettangolo minimo precedentemente calcolata.

SET_DONE: stato che alza ad 1 il valore del segnale di $o\_\ done$ per segnalare il termine dell'elaborazione.

UNSET_DONE: stato che riporta a 0 il valore del segnale di $o\_\ done$ dopo un ciclo.

2.1 Ottimizzazioni

Il design presentato nel diagramma precedente, non è quello inizialmente pensato della FSM, ma frutto di una serie di modifiche ed ottimizzazioni, il cui obiettivo è stato quello di cercare di aumentare la frequenza massima del componente, ovvero una riduzione della velocità di clock, a discapito di un aumento del numero di cicli totali (il quale resta comunque un numero costante), di area e complessità circuitale della RTL. I risultati ottenuti e il come sono spiegati partendo dal primo design iniziale fino a raggiungere il design rappresentato ed implementato.

Il primo design pensato, era costituito da due process, uno sincronizzato con il segnale di clock, la cui sensitivity list era costituita da i segnali di clock e reset, ed uno di logica combinatoria sulla base dello stato corrente e dei dati in input. Questa scelta è stata poi scartata per varie ragioni: la versione con un process è sicuramente di più facile lettura, comprensione e scrittura, avere un unico process permette di poter sincronizzare tutto attraverso il ciclo di clock, evita di avere una sensitivity list molto lunga nel process combinatorio ed evita inteferenze di latch non desiderate.

Al termine della prima realizzazione il progetto costituito da un solo process, aveva un numero di stati inferiore ed aveva al suo interno operazioni complesse e computazionalmente pesanti, le moltiplicazioni. Nonostante l'FPGA target sia dotato di blocchi DSP in grado di svolgere operazioni complesse, queste comunque risultavano ancora molto lente. In questo step di ottimizzazione il programma otteneva attraverso un timing constraint una velocità di clock di 7.7 ns.

Il passo successivo è stato quello di minimizzare il logic delay, in particolare il local datapath delays, ovvero il ritardo della logica tra elementi di memorizzazione, perché il ritardo del più lento datapath determina la massima frequenza di clock, ed è stato fatto ridotto aggiunto altri registri, nella pratica riducendo stati più complessi e ricchi di operazioni in sottostati più semplici e con meno operazioni , il che si traduce in una maggiore velocità ottenibile per stato. Le moltiplicazioni venivano svolte in uno stato dedicato, e la velocità raggiunta era di 6.2 ns.

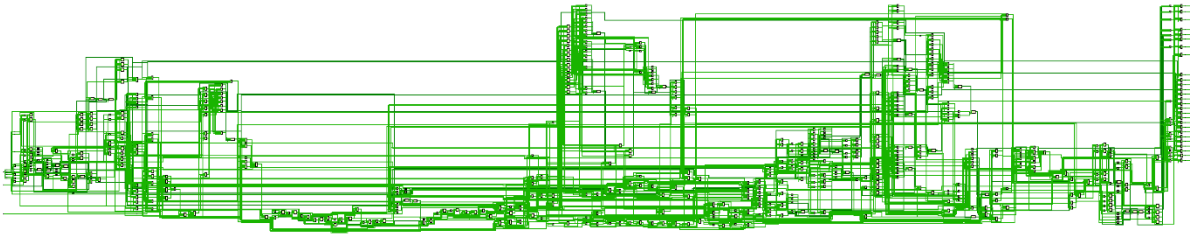
Successivamente si è pensato di eliminare l'operazione di moltiplicazione, con uno stato in grado di fare le N somme richieste per ottenere il medesimo risultato della moltiplicazione. Con tale ottimizzazione: stati piccoli e con poche operazioni ed eliminazione della moltiplicazione, si è riusciti ad ottenere una velocità di clock di 4.3 ns.

Infine si è pensato di modificare nel settings del tool di sintesi, la codifica degli stati effettuata nel processo di sintesi, cambiando la proprietà fsm-extaction, modificando quindi la codifica degli stati al momento della sintesi da auto a gray, il che ha permesso di guadagnare con il design implementato di guadagnare alcuni ns e portare la velocità di clock a 4.1 ns.

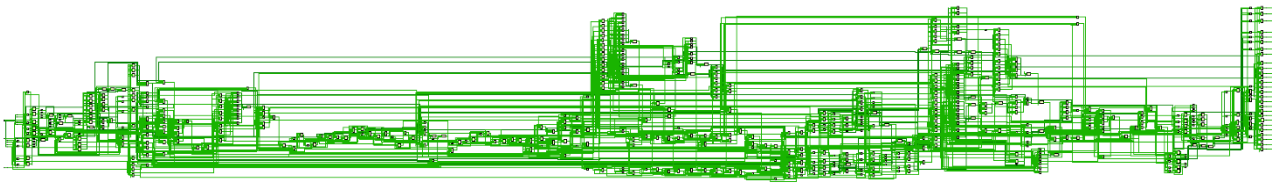
Di seguito i risultati ottenuti

2.1.1 RTL Schematic

Schematic generato Post-Synthesis



Schematic generato Post-Implementation



La complessità dello schematic è stata sacrificata a favore di una maggiore velocità di clock, ed un conseguente aumento dell'area del circuito, in seguito alle numerose piccole operazioni svolte.

2.1.2 Timing

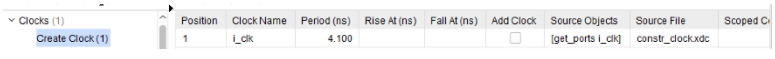
N. Process	Moltiplicazione	Numero e logica stati	fsm-extraction	clock
1	Si	Pochi – complessa	Auto	7.7 ns
1	Si	Parecchi - semplice	Auto	6.2 ns
1	No	Parecchi – semplice	Auto	4.3 ns
1	No	Parecchi – semplice	Gray	4.1 ns

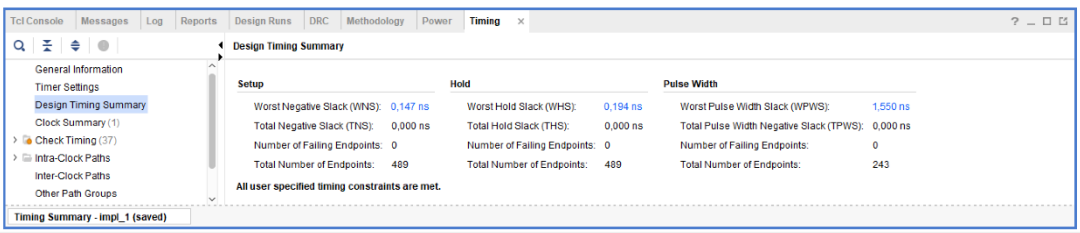
La codifica gray risulta essere la migliore per l'obbiettivo posto inizialmente.
Confrontandola con le altre:

- Auto 4.3 ns
- One_hot 4.2 ns
- Johnson 4.2 ns
- Gray 4.1 ns

Il clock del circuito è stato verificato tramite un timing constraint:

create_clock -period 4.1 -name i_clk [get_ports i_clk]





3 Algoritmo

L'algoritmo usato per la realizzazione del componente è un semplice algoritmo di scansione lineare della matrice, dal primo indirizzo valido, fino all'indirizzo massimo, il cui valore corrisponde a quello dell'area massima del rettangolo ottenibile con i valori del numero di colonne e di righe assegnati dalla memoria, quindi nel caso pessimo il componente analizzerà un numero di pixel pari al valore dell'area massima del rettangolo.

Il semplice algoritmo, ad ogni ciclo, legge il valore presente nell' indirizzo di memoria assegnato, controlla se tale pixel appartiene all'immagine o meno e nel caso il valore sia minore del valore di soglia, il componente salta il resto dei passaggi e si prepara a leggere il pixel successivo, altrimenti comincia l'analisi del pixel.

L'analisi si divide in due fasi:

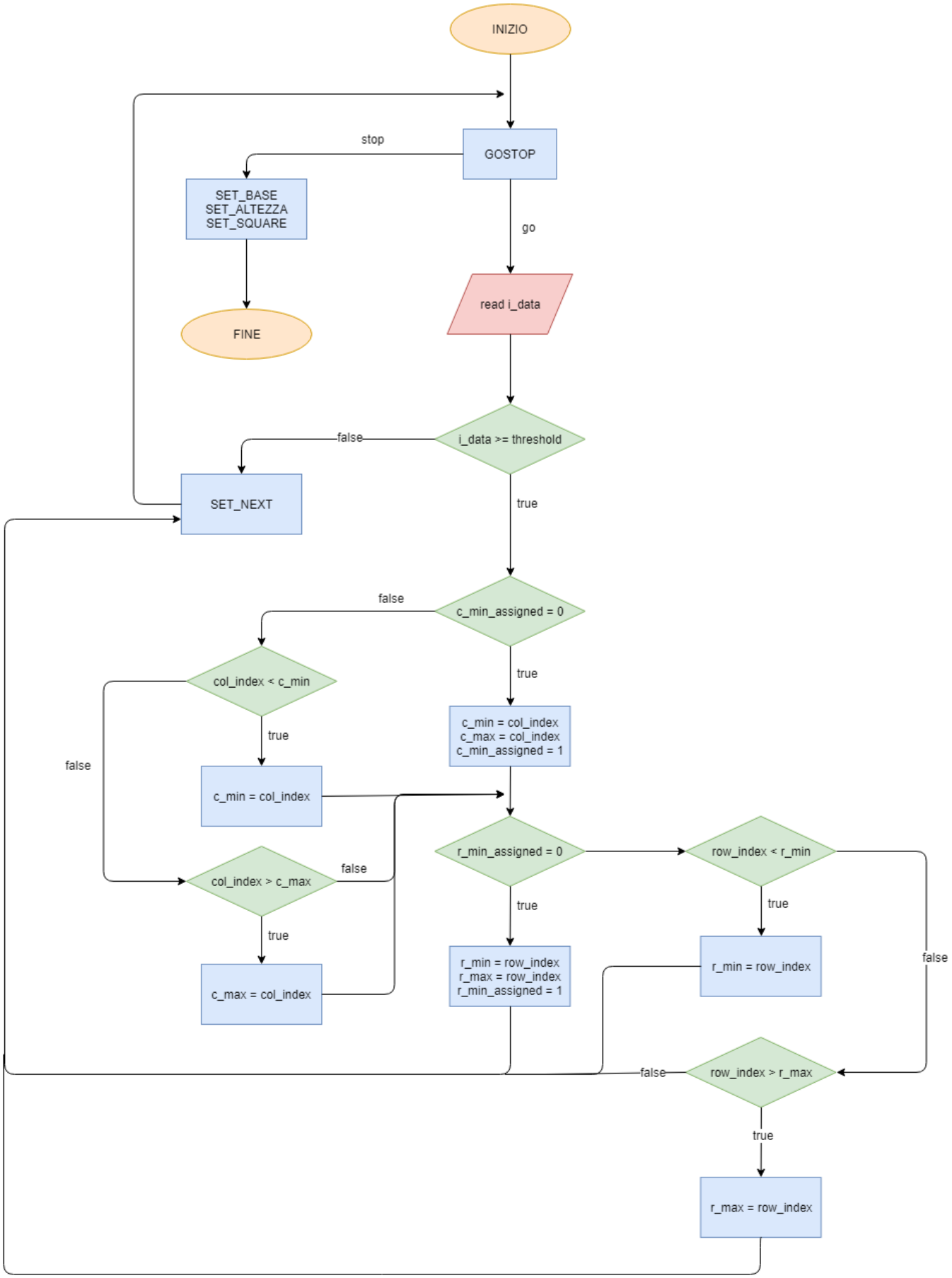
- prima fase, non è stato trovato nessun pixel appartenente all'immagine e quindi in questo caso i valori di c_min, c_max, r_min, r_max vengono assegnati agli indici di riga e di colonna del primo elemento trovato con valore maggiore o uguale al valore soglia;
- seconda fase, avviene la lettura di qualsiasi altro elemento appartenente all'immagine dopo che sia stato trovato il primo. In questo caso, vengono confrontati gli indici di colonna e di riga dell'elemento corrente con quelli assegnati ai valori di minimo e di massimo. Se vengono trovati dei valori minori o maggiori, questi ultimi vengono aggiornati e si prosegue con il pixel successivo, fino a raggiungere la fine dei pixel in lettura.

Al termine del controllo della matrice viene calcolata l'area del rettangolo minimo che circoscrive la figura, il cui calcolo è dato da:

$$base = c\_max - c\_min + 1$$
$$altezza = r\_max - r\_min + 1$$
$$square = base * altezza$$

Un flowchart dell'algoritmo è presente qui di seguito.

N.B. Il flowchart rappresentato tralascia alcuni dettagli riguardanti l'implementazione, come la gestione degli indici colonna e riga, ed il passo finale del settaggio del segnale di o_done. Per tali dettagli guardare il codice ampiamente commentato. Lo scopo è quello di fornire un diagramma ad alto livello, che permettesse un'agevole comprensione dell'algoritmo usato, seppur semplice.


INIZIO
GOSTOP
stop
go
SET_BASE
SET_ALTEZZA
SET_SQUARE
FINE
read i_data
i_data >= threshold
false
true
SET_NEXT
c_min_assigned = 0
false
true
col_index < c_min
true
false
c_min = col_index
c_min = col_index
c_max = col_index
c_min_assigned = 1
r_min_assigned = 0
row_index < r_min
col_index > c_max
false
true
true
c_max = col_index
r_min = row_index
r_max = row_index
r_min_assigned = 1
r_min = row_index
false
false
row_index > r_max
true
r_max = row_index

4 Testbench e Risultati

Al termine dell'ottimizzazione del componente, si è passati ad un'adeguata fase di testing per verificarne la correttezza delle operazioni svolte. In primis sono stati usati gli otto testbench forniti per l'autovalutazione, sia i primi quattro test con i quali alcuni colleghi avevano riscontrato problemi, sia i quattro test inseriti successivamente e nominati delay.
Al fine di verificare l'effettivo funzionamento, sono stati creati altri test, parte dei quali sono stati creati tramite una funzione Java, il cui codice è riportato nel paragrafo successivo, ed il cui scopo era quello di creare dei test casuali e verificarne il risultato. Ai test generati casualmente, ne sono stati aggiunti altri creati ad hoc per testare alcuni casi corner o ritenuti di particolare rilevanza. Tutti i test sono stati provati sia in pre-sintesi che post-sintesi, e sono spiegati in dettaglio qui di seguito.

Test effettuati:

- File di test consegnati per l'autovalutazione
- File di test contenente 30 test consecutivi, per verificare che il componente al termine di una elaborazione, fosse in grado di ripartire e concludere una nuova elaborazione.
- 20 file di test singoli generati random dallo script.
- Test particolare rilevanza:
    - Caso matrice vuota e piena
    - Casi soglia zero, numero righe e colonne zero
    - Casi soglia massima, numero righe e colonne massimo
    - Caso solo ultimo bit della matrice pieno, resto vuoto
    - Casi diagonale, triangolo, e croce per ulteriore verifica

I risultati ottenuti al termine della fase di testing ha confermato la capacità del componente di superare correttamente la simulazione *Behavioral*, la simulazione *Post-Synthesis Functional* e *Post-Synhtesis Timing* con periodo di clock a 4.1 ns.

5 Code

class script {

```
    private static final String ZERO = "00000000";

    private int randomInt() {
        Random random = new Random();
        return random.nextInt(256);
    }


    //conversione binaria per 8 bit
    private String bin(int decimal) {
        String bin = Integer.toBinaryString(decimal);
        if (bin.length() == 8) {
            return bin;
        } else if (bin.length() == 7) {
            return "0" + bin;
        } else if (bin.length() == 6) {
            return "00" + bin;
        } else if (bin.length() == 5) {
            return "000" + bin;
        } else if (bin.length() == 4) {
            return "0000" + bin;
        } else if (bin.length() == 3) {
            return "00000" + bin;
        } else if (bin.length() == 2) {
            return "000000" + bin;
        } else {
            return "0000000" + bin;
        }
    }

    // conversione binaria per 16 bit utile per l'area
    String binSquare(String bin) {

        if (bin.length() == 16) {
            return bin;
        } else if (bin.length() == 15) {
            return "0" + bin;
        } else if (bin.length() == 14) {
            return "00" + bin;
        } else if (bin.length() == 13) {
            return "000" + bin;
        } else if (bin.length() == 12) {
            return "0000" + bin;
        } else if (bin.length() == 11) {
            return "00000" + bin;
        } else if (bin.length() == 10) {
```

```
            return "000000" + bin;
        } else if (bin.length() == 9) {
            return "0000000" + bin;
        } else if (bin.length() == 8) {
            return "00000000" + bin;
        } else if (bin.length() == 7) {
            return "000000000" + bin;
        } else if (bin.length() == 6) {
            return "0000000000" + bin;
        } else if (bin.length() == 5) {
            return "00000000000" + bin;
        } else if (bin.length() == 4) {
            return "000000000000" + bin;
        } else if (bin.length() == 3) {
            return "0000000000000" + bin;
        } else if (bin.length() == 2) {
            return "00000000000000" + bin;
        } else {
            return "000000000000000" + bin;
        }
    }


    //riempe la matrice con valori casuali
    void fillMatrix(String[] matrix) {

        matrix[0] = ZERO;
        matrix[1] = ZERO;

        int squareToFill = Integer.parseInt(matrix[2], 2) *
Integer.parseInt(matrix[3], 2);

        for (int i = 5; i < matrix.length; i++) {
            if (i <= squareToFill + 4) {
                matrix[i] = bin(randomInt());
            } else {
                matrix[i] = ZERO;
            }
        }
    }

    void setLength(String[] matrix) {
        matrix[2] = bin(randomInt());
    }

    void setHeight(String[] matrix) {
        matrix[3] = bin(randomInt());
    }

    void setThreshold(String[] matrix) {
        matrix[4] = bin(randomInt());
    }
```

```
    List<Integer> values(String[] matrix) {
        List<Integer> values = new ArrayList<>();
        values.add(Integer.parseInt(matrix[2], 2));
        values.add(Integer.parseInt(matrix[3], 2));
        values.add(Integer.parseInt(matrix[4], 2));
        return values;
    }

    List<Integer> minRectCalculator(String[] matrix) {

        int width = Integer.parseInt(matrix[2],2)-1;
        int height = Integer.parseInt(matrix[3],2);
        int xMin = 0;
        int xMax = 0;
        int yMin = 0;
        int yMax = 0;
        int square = 0;
        boolean xFounded = false;
        boolean yFounded = false;
        int threshold = Integer.parseInt(matrix[4], 2);
        int    squareToFill    =    Integer.parseInt(matrix[2],2)    *
Integer.parseInt(matrix[3], 2);
        int indexRow = 0;
        int indexCol = 0;

        for (int i = 5; i <= squareToFill + 4; i++) {

            if (Integer.parseInt(matrix[i], 2) >= threshold) {
                if (!xFounded) {
                    xMin = indexCol;
                    xMax = indexCol;
                    xFounded = true;
                }
                else {
                    if (indexCol < xMin)
                        xMin = indexCol;
                    if (indexCol > xMax)
                        xMax = indexCol;
                }
                if (!yFounded) {
                    yMin = indexRow;
                    yMax = indexRow;
                    yFounded = true;
                }
                else {
                    if (indexRow < yMin)
                        yMin = indexRow;
                    if (indexRow > yMax)
                        yMax = indexRow;
                }
            }

            if (indexCol == width) {
```

```
            indexCol = 0;
            indexRow++;
        }
        else {
            indexCol++;
        }

    }

    int base = xMax - xMin + 1;
    int altezza = yMax - yMin + 1;

    if (width == 0 || height == 0) {
        square = 0;
    }
    else if ((base == 1) && (altezza == 1) && !xFounded) {
        square = 0;
    }
    else {
        square = base * altezza;
    }


    List<Integer> valori = new ArrayList<>();
    valori.add(square);
    valori.add(xMin);
    valori.add(xMax);
    valori.add(yMin);
    valori.add(yMax);
    return valori;

 }
```

```
static String fillMatrixTest(String[] matrix, int squareToFill) {

    String str = "";

    for(int i = 2; i < matrix.length; i++) {
        if (i <= squareToFill) {
            str = str + (toStringMatrix(matrix[i], i));
        }
        else {
            break;
        }
    }
    str = str + (toStringOthers());

    return str;
 }
```

```
static String toStringMatrix(String matrixValue, int index) {

        String str = index + " => " + "\"" + matrixValue + "\"" + ",";
        return str;
    }

        return "others     static String toStringOthers() {
=> (others =>'0')";
    }


 static void testBench(List<Integer> list, String[] matrix, int square)
throws IOException {

    // funzione di cui è stato omesso il codice poiché non è altro che
    // la stampa di un testbench con i valori sostituiti in seguito ai
    // calcoli delle altre funzioni, in particolare dopo aver sostituito
    // i valori interni della matrice, l'msb, l'lsb, crea un file di
   //formato vhd che è stato dato in pasto al tool di sintesi Vivado
```

```
public class Main {

    public static void main(String[] args) {

        String msb;
        String lsb;
        int squareToFill;
        String[] matrix;
        script script = new script();
        matrix = new String[65536];

        script.setHeight(matrix);
        script.setLength(matrix);
        script.setThreshold(matrix);
        script.fillMatrix(matrix);


        squareToFill      =      Integer.parseInt(matrix[2],2)      *
Integer.parseInt(matrix[3], 2);
         List<Integer> val  = script.minRectCalculator(matrix);

        String bin = Integer.toBinaryString(val.get(0));
        String binArea = script.binSquare(bin);
        msb = binArea.substring(0, 8);
        lsb = binArea.substring(8, 16);

        matrix[1] = msb;
        matrix[0] = lsb;

        com.me.script.fillMatrixTest(matrix, squareToFill);

        try {
            Main.testBench(val, matrix, squareToFill);
        } catch (IOException e) {
            e.printStackTrace();
        }

    }


}
```