Politecnico di Torino

Dipartimento di elettronica e telecomunicazioni

Ingegneria Elettronica

Report Laboratorio n.6

Gruppo n.18

Elettronica Dei Sistemi Digitali

Autori:
Balaara Angelo (257582)
Bellettini Leonardo (260347)
Caddeo Gabriele Mario (263414)
Valloni Melissa (261799)

Prof:
Maurizio Zamboni

5 Maggio 2019

CAPITOLO 1

Progettazione

1.1 INTRODUZIONE

L'obiettivo di tale laboratorio è la realizzazione di una rete neurale artificiale semplificata, rappresentabile tramite connessioni di più neuroni. In particolar modo, considerando una memoria come primo neurone in cui vengono caricati 1024 dati in ingresso, questi ultimi vengono processati per poi essere caricati in una seconda memoria della stessa grandezza, restituendo, inoltre, in uscita, il conteggio dei segnali processati che assumono un valore positivo.

1.2 PSEUDO CODICE

```
int8_t dataIn [1024];
int8_t memB [1024];
int8_t memA [1024];
uint16_t positiveNum;
bool done = 0;
for (int i=0; i < 1024; i++){
  memA[i] = dataIn[i];
}
for (int i=0; i < 1024; i++){
  if (i == 0){
    memB[i] = memA[i]*0.5;
  }else if (i == 1){
    memB[i] = memA[i]*0.5 + memA[i-1]*3.75;
  }else{
    memB[i] = memA[i]*0.5 + memA[i-1]*3.75 - memA[i-2];
  }
  if (memB[i] > 127){
    memB[i] = 127;
  }else if (memB[i] < -128){
    memB[i] = -128;
  }
  if (memB[i] > 0){
    positiveNum++;
  }
}
done = 1;
```

Come da specifica di progetto, una volta ricevuto il segnale START dall'esterno, si inizia a riempire la memoria A (memA) con i dataIn presenti nel bus di ingresso nella locazione di memoria puntata dall'indirizzo corrente (i); una volta processati i dati in ingresso si andranno a riempire con i rispettivi risultati le celle della memoria B (memB) nei rispettivi indirizzi. Poichè le memorie utilizzate sono di 1 KByte, l'operazione va iterata 1024 volte (0-1023). L'algoritmo, che è in grado di implementare l'equazione, può essere diviso in diverse parti, per i primi due indirizzi occorrono meno operazioni, infatti, quando i=0, allora memB[i] = memA[i] $*$ 0.5, quando invece l'address è uguale a '1' memB[i] = memA[i] $*$ 0.5 + memA[i−1] $*$ 3.75, per tutti gli altri casi fino a i $<$ 1024 le celle di memoria della memB vengono riempite mediante le seguenti operazioni memB[i] = memA[i] $*$ 0.5 + memA[i−1] $*$ 3.75 − memA[i−2].
Inoltre, poichè la memoria B, come la memoria A, può contenere solo dati esprimibili al massimo con 8 bit, nel caso in cui le operazioni diano come risultato numeri maggiori, si ricorre, mantenendo il giusto segno, alla tecnica dell'aritmetica saturata, in modo che il minor numero negativo rappresentabile sia -128 e il maggior numero positivo rappresentabile sia 127 (entrambi con 8 bit).
Infine, occorre anche conteggiare di volta in volta il numero dei valori positivi in ingresso alla memB e una volta che quest'ultima è stata riempita completamente, in uscita si avrà il segnale DONE=1 e il numero dei valori positivi che sono stati rilevati.

1.3 DATAPATH

Il datapath doveva assolvere il seguente algoritmo:

$$
\begin{aligned}
&Y(0) = 0.5X(0)\\
&Y(1) = 0.5X(1) + 3.75X(0)\\
&Y(2) = 0.5X(2) + 3.75X(1) - X(0)\\
&...
\end{aligned}
$$

Come richiesto da specifica, il datapath non prevede l'utilizzo di un moltiplicatore e dipende da un sommatore per il calcolo della uscita $Y(n)$ ed uno per per il calcolo delle $Y(n)$ maggiori di zero.
Essendo le costanti delle moltiplicazioni combinazioni di potenze di 2, si è riuscito a sfruttare lo shift dei bit e la somma/sottrazione, evitando l'utilizzo di un moltiplicatore. Per effettuare la moltiplicazione per 2^{-1} si è utilizzato uno shift, che shiftasse il dato di uno verso destra. La moltiplicazione per 3.75 è stata eseguita come se fosse una moltiplicazione per $\frac{15}{4}$, a sua volta suddiviso in $\frac{16-1}{4}$ (shift a sinistra 4, sottrazione con se stesso, shift a destra di 2). Per via di quest'ultima operazione il parallelismo interno è maggiore di quello dei dati in input. Il minor numero di bit d'espansione del segno che permette di eseguire la moltiplicazione per 16 senza perdita di informazione è 4. Il parallelismo interno diventa quindi di 12bit.

Nella realizzazione del datapath sono state vagliate diverse varianti. Quella reputata più soddisfacente per numero di component necessari e per l'assenza di ridondanze è quella che è stata poi implementata (figura 1.1).

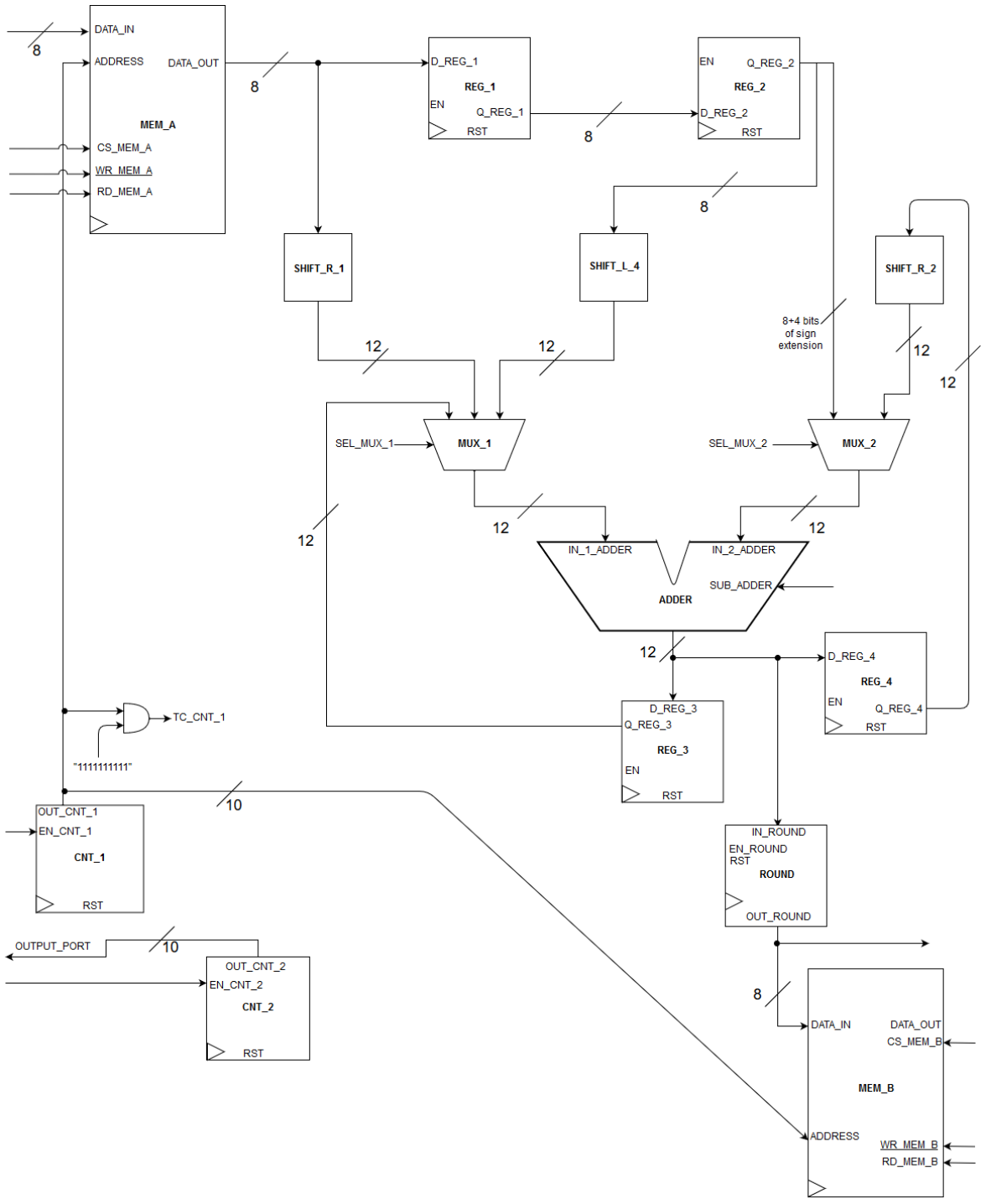

Figura 1.1: Datapath

Di seguito un'analisi più dettagliata:

- I registri REG_1 e REG_2 vengono utilizzati per ritardare i dati in uscita dalla memA, in modo da averli disponibili al momento opportuno, e per salvarli per le operazioni successive.
- I componenti SHIFT_R_1, SHIFT_L_4 e SHIFT_R_2 sono utilizzati per gli shift sopra citati; sono stati realizzati con una semplice traslazione dei segnali in maniera combinatoria.
- I multiplexer MUX_1 e MUX_2 occorrono per selezionare, al momento opportuno, i dati da fornire al sommatore.
- Il componente ADDER permette di eseguire sia addizioni che sottrazioni semplicemente sfruttando il segnale SUB_ADDER (0 per le addizioni e 1 per le sottrazioni).
- Sfruttando i registri REG_3 e REG_4 si salvano le operazioni intermedie ($0.5X(n-2) - X(n)$ e $16X(n-1) - X(n-1)$).
- I contatori CNT_1 e CNT_2 sono usati, rispettivamente, per scorrere gli indirizzi delle memorie e contare le uscite positive della rete neurale.
- Il componente ROUNDER permette di arrotondare i dati con parallelismo di 12 bit per ottenere dati con parallelismo di 8 bit (come in ingresso). Da specifica, il rounder lavora in logica saturata, quindi se il dato in complemento a due è rappresentabile in 8 bit non viene modificato, in caso contrario viene rappresentato con il massimo valore possibile sugli 8 bit (rispettando il suo segno). L'arrotondamento è implementato osservando i 4 MSB del dato in ingresso al rounder: se questi sono tutti uguali (la XOR tra di essi restituisce 0) allora non vi è stato overflow e si effettua un semplice troncamento, mentre se anche un bit di questi non fosse uguale agli altri allora siamo in presenza di overflow, pertanto saturiamo a "10000000" o "01111111".

1.4 ASM CHART ALGORITMICA

L'ASM chart dell'Algoritmo nasce dall'evoluzione dello pseudo codice. All'inizio la macchina si trova in uno stato (IDLE) in cui tutti i registri vengono resettati, ovvero settati a 0, e sia il contatore degli indirizzi sia quello dei numeri positivi non incrementano il loro valore. Quando viene dato il segnale di START dall'esterno, inizia l'esecuzione dell'algoritmo. In primo luogo vi è il riempimento della memoria A con i dati forniti dall'esterno. La fine di tale operazione viene sancita dal segnale di TC (attivo quando il counter raggiunge 1023, ovvero quando viene puntata l'ultima locazione della memoria).
Al colpo di clock successivo, nello stato ENABLE_MEM_A, il contatore riparte da 000000000 senza la necessità del reset e la memoria A viene abilitata. Si procede poi nello stato successivo (OPERATION_1), dove ha inizio la prima operazione. Si va a inserire in REG_3 il dato puntato dall'indirizzo corrente MEM_A(ADDRESS) moltiplicato per 0.5, a cui si sottrae poi (MEM_A_MINUS_2); per quest'ultimo occorre fare un'ulteriore considerazione, infatti, sarà uguale a 0 se ADDRESS<2, altrimenti sarà uguale a MEM_A(ADDRESS−2).
Si procede poi con OPERATION_2 in cui si va a inserire nel REG_4 il dato (MEM_A_MINUS_1) moltiplicato per 16, a cui si sottrae poi MEM_A_MINUS_1 stesso; quest'ultimo sarà uguale a 0 se ADDRESS<1, altrimenti sarà uguale a MEM_A(ADDRESS−1).
Lo stato successivo (OPERATION_3) permette di ottenere l'ingresso del ROUNDER attraverso la somma del contenuto di REG_3 e di (REG_4/4).
Nello stato WRITE_MEM_B si ha il troncamento del risultato delle varie operazioni precedenti in modo tale da poter caricare nella memoria B gli 8 bit più significativi, allocandoli nell'indirizzo corrente.
A questo punto, se l'uscita del ROUNDER è positiva, si andrà nello stato POSITIVE_OUT, il contatore dei numeri positivi verrà incrementato di uno (POSITIVES++) e si andrà a all'indirizzo successivo, altrimenti, se l'uscita del ROUNDER è negativa, si andrà nello stato NOT_POSITIVE_OUT e si incrementerà solo il contatore dell'indirizzo (ADDRESS++).
Infine, se ADDRESS++ non ha raggiunto l'ultimo valore (ovvero 1023), la macchina ritorna nello stato ENABLE_MEM_A, altrimenti va in END_STATE dove DONE=1 in uscita indica la fine del riempimento di MEM_B.
A questo punto, se viene dato il segnale di START si ritorna nello stato di IDLE, altrimenti si rimane in END_STATE.

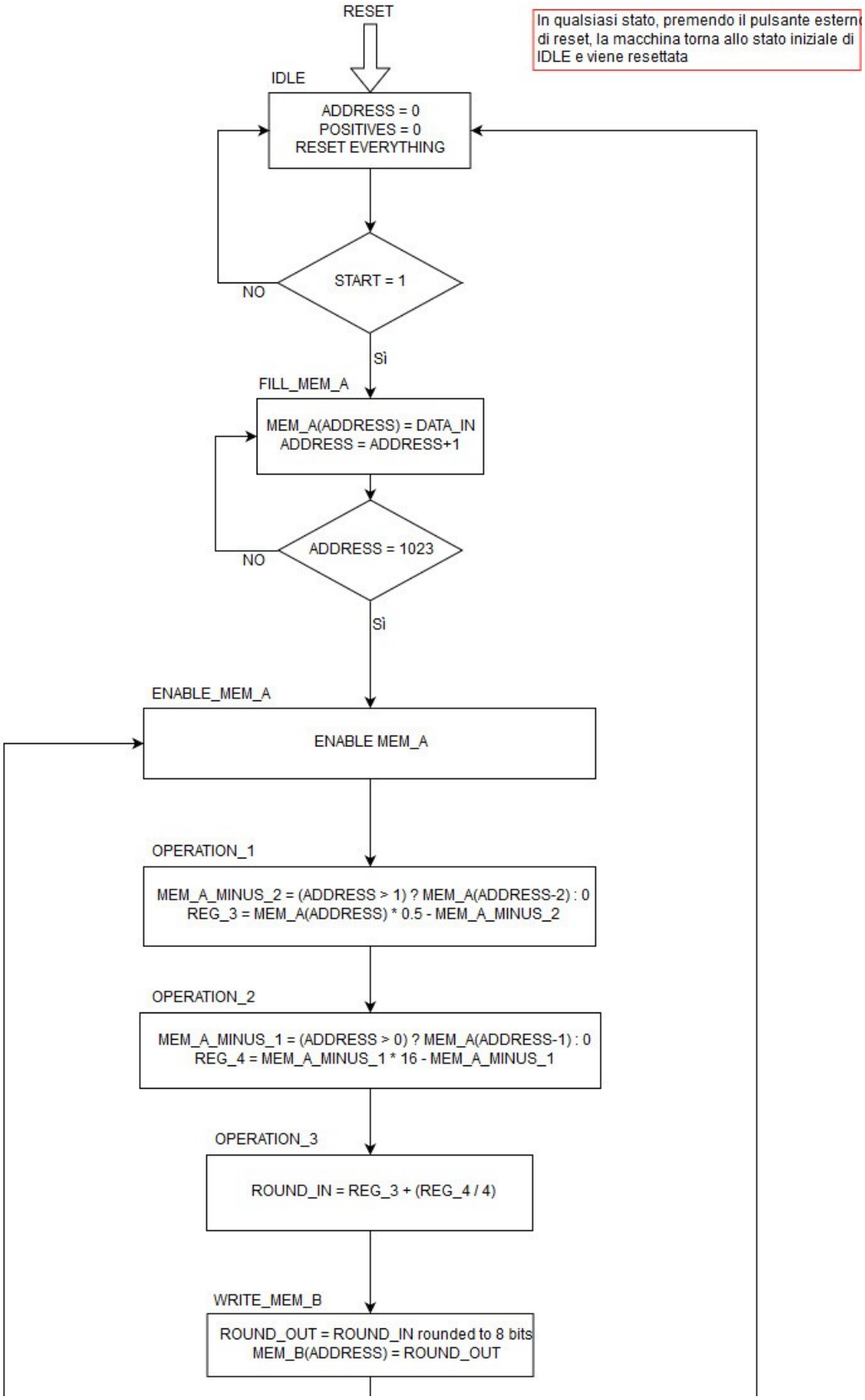

Figura 1.2: Algorithm ASM part 1

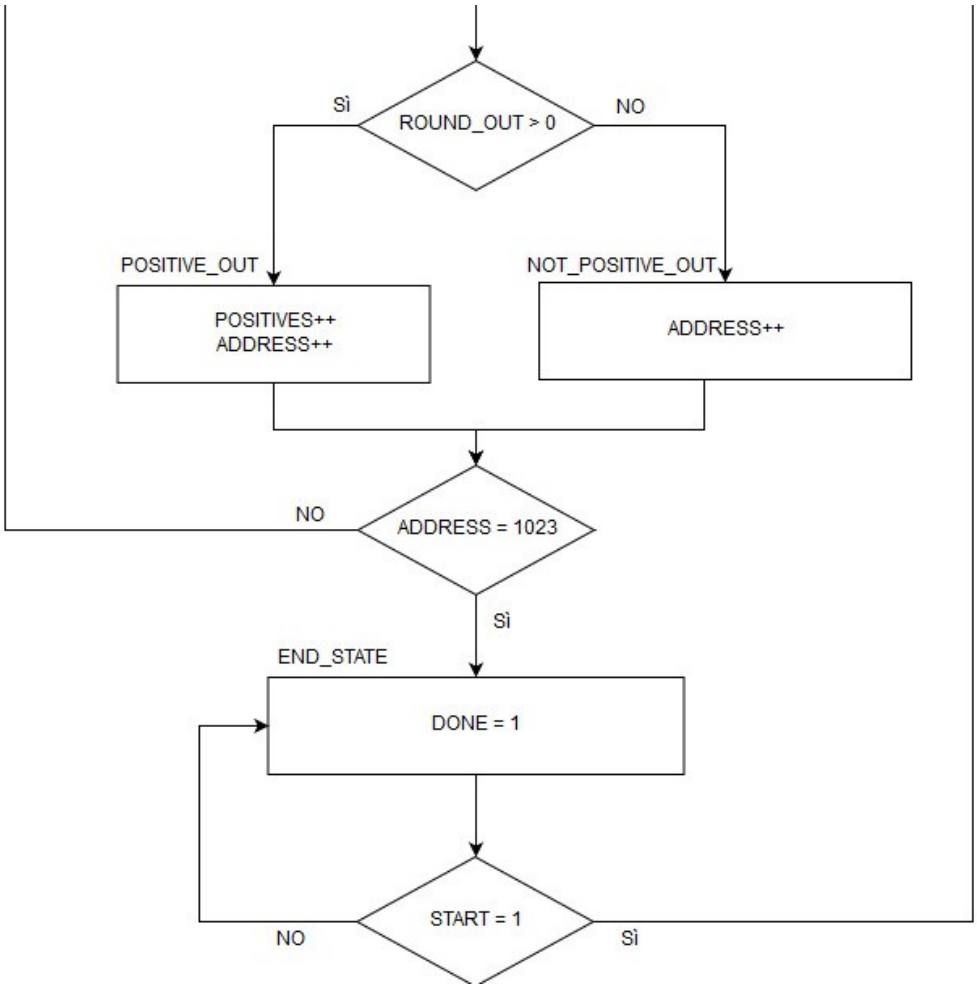

Figura 1.3: Algorithm ASM part 2

1.5 ASM CHART DELLA CONTROL UNIT

Il corretto funzionamento del sistema è garantito solo con un opportuno scambio di segnali tra il datapath e la control unit. Le tipologie di segnali che vengono scambiati si distinguono in segnali di stato e di controllo.
Per quanto riguarda i primi citati, si può individuare:

- TC_CNT_1: permette di capire se il contatore degli indirizzi ha raggiunto il valore finale 1023.
- OUT_ROUND: permette di conteggiare i numeri positivi.

Per quanto riguarda invece i segnali di controllo:

- Segnali per la gestione delle memorie: CS_MEM_A, CS_MEM_B che permettono di abilitare le memorie, WR_MEM_A, RD_MEM_A, WR_MEM_B, RD_MEM_B che permettono di leggere e scrivere nelle memorie.

- Segnali per la gestione dei registri: riguardano i segnali di abilitazione dei registri EN_REG_1, EN_REG_2, EN_REG_3, EN_REG_4.

- segnali per la gestione dei multiplexer : si tratta di tutti i segnali di selezione dei MUX: SEL_MUX_1, SEL_MUX_2.

- Segnale per la gestione del counter : ovvero il segnale che ne gestisce l'incremento EN_CNT_1, EN_CNT_2;

- Segnale per la gestione dell'adder : SUB_ADDER_1 tramite il quale si controlla se eseguire un'addizione o una sottrazione.

- Segnale per la gestione del rounder: EN_ROUNDER che permette l'abilitazione di quest'ultimo.

- RESET_n tramite cui si possono resettare tutti i componenti.
  Dati i segnali di stato tramite cui la CU deve prendere le decisioni, quelli di controllo con cui si gestisce correttamente il datapath e l'ASM chart dell'algoritmo, il passaggio dall'ASM chart dell'algoritmo all'asm chart della control unit è immediato.

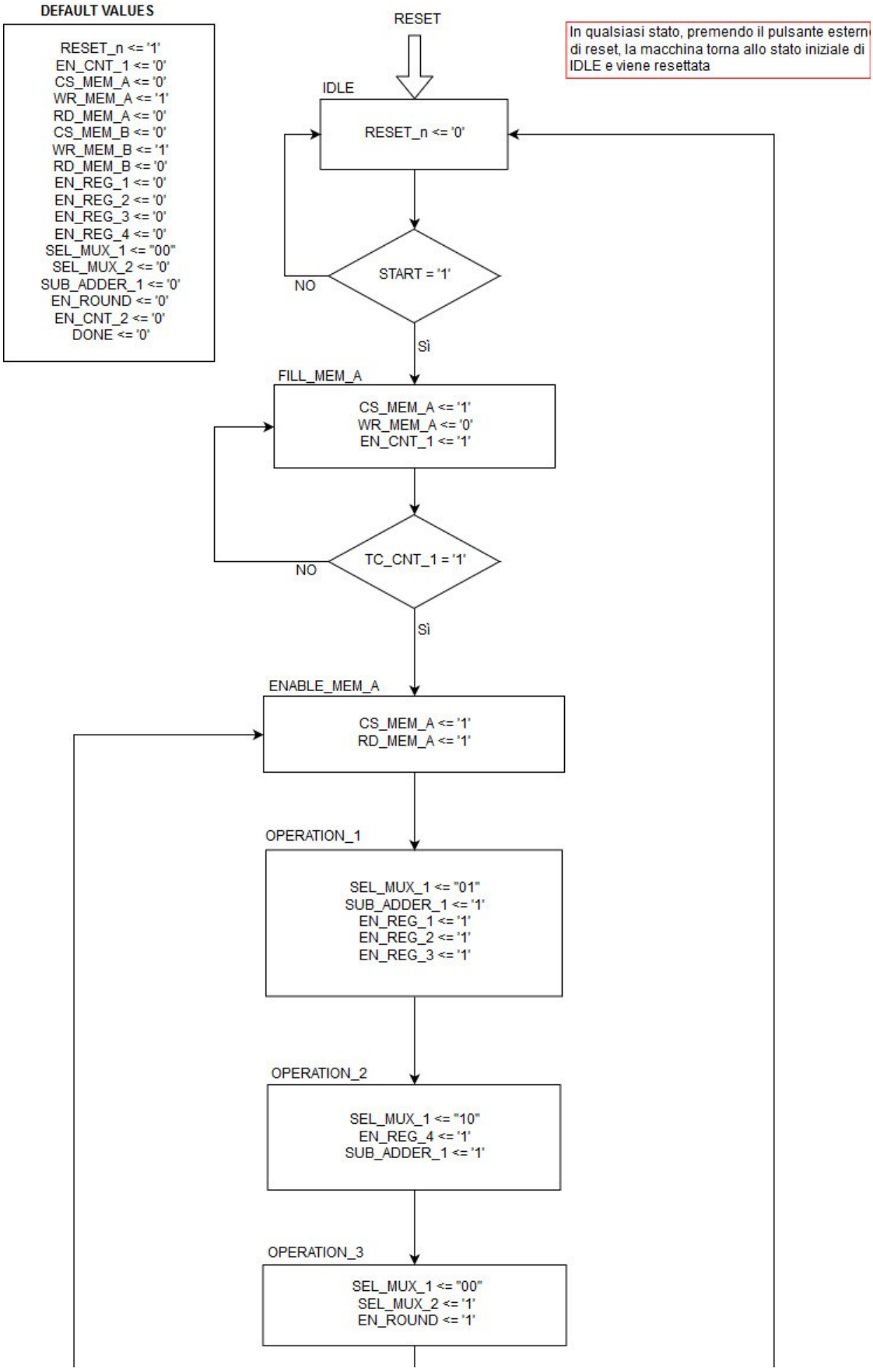

Figura 1.4: Control ASM part 1

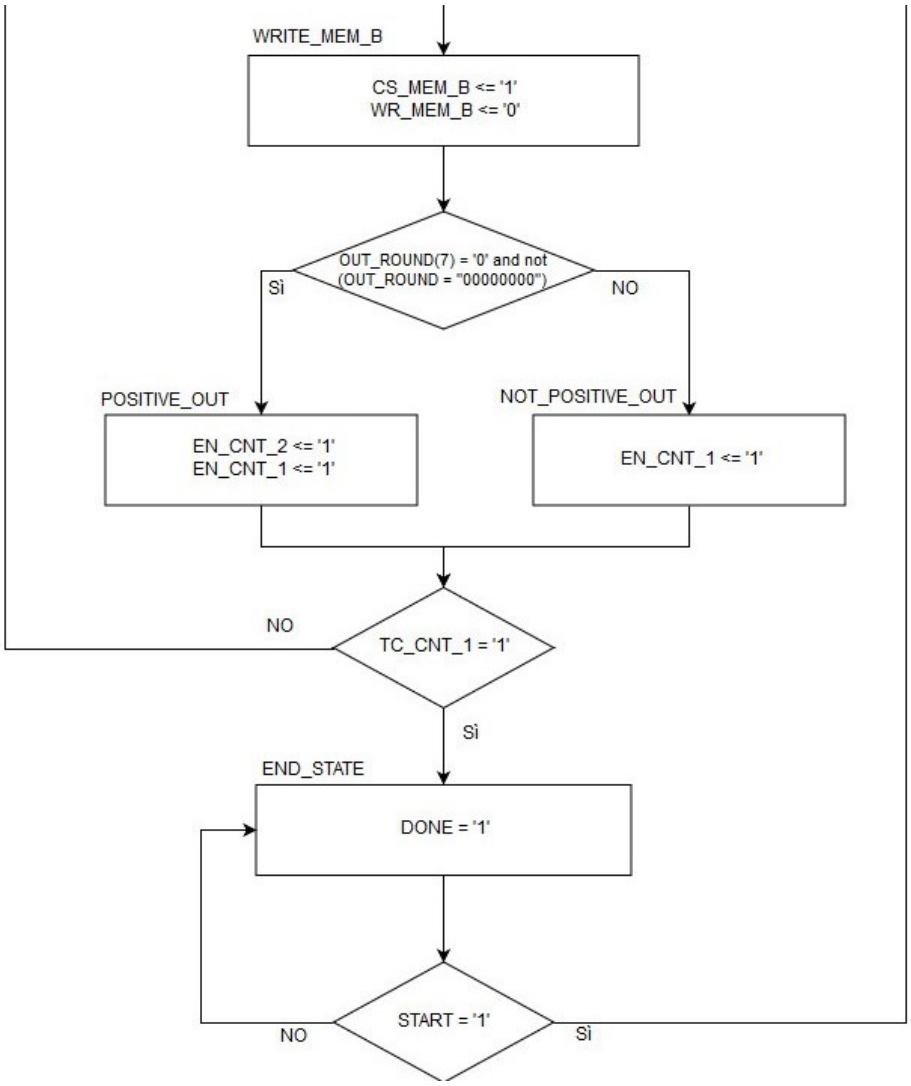

Figura 1.5: Control ASM part 2

CAPITOLO 2

Timing e simulazione

Successivamente si è ritenuto necessario verificare che l'algoritmo potesse funzionare in accordo con il timing diagram deciso in fase di progetto (riportato in figura 2.1).

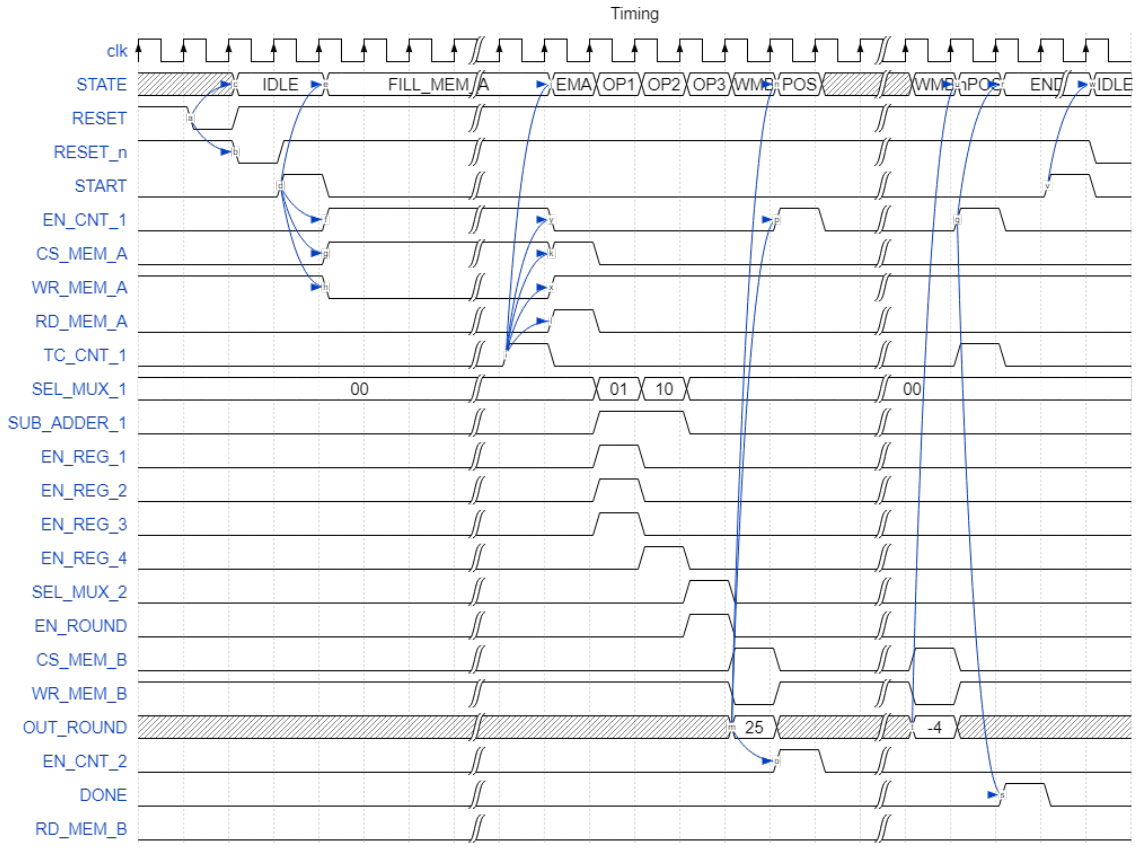

Figura 2.1: Timing Diagram

In primo luogo, il reset asincrono esterno resetta la macchina a stati, si entra così nello stato di IDLE, dove la FSM invia un reset sincrono a tutti i registri del datapath, inizializzandoli a 0. Una volta ricevuto dall'esterno il segnale START,

si esce dallo stato di IDLE passando allo stato FILL_MEM_A nel clock successivo. In questo stato la Memoria A viene caricata con gli inputs esterni e i segnali CS_MEM_A, WR_MEM_A, EN_CNT_1 vengono attivati. In figura 2.2 il confronto con la simulazione in Modelsim.

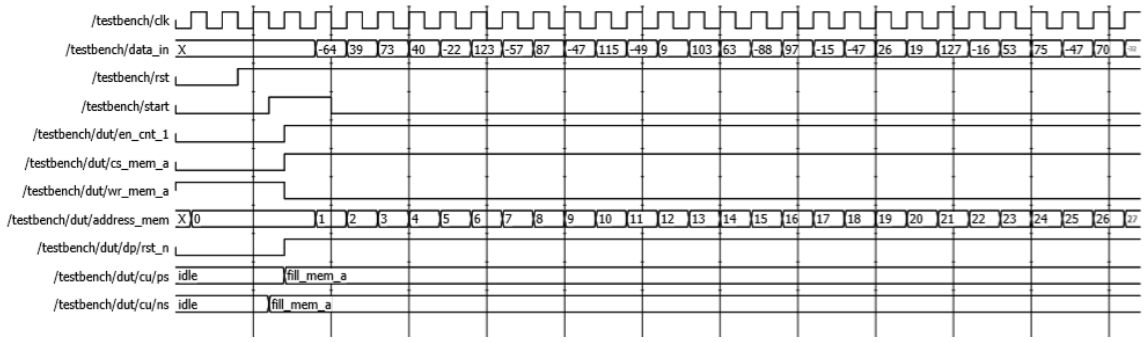

Figura 2.2: IDLE e FILL_MEM_A da simulazione in Modelsim

Una volta che il contatore ha finito di contare fino a 1023 e ha mandato un segnale di fine conta attraverso il segnale TC_CNT_1, si passa dallo stato FILL_MEM_A allo stato ENABLE_MEM_A (EMA in figura 2.1). Qui si attiva il segnale di RD_MEM_A e si disattivano i segnali WR_MEM_A e EN_CNT_1 (perchè il contatore deve rimanere fermo a 0 fino alla fine della scrittura del risultato in MEM_B). In questo stato si abilita la memoria indicando l'indirizzo opportuno da cui si vuole leggere il dato e che sarà disponibile in uscita al clk successivo.
Nel colpo di clock successivo, si entra nello stato OPERATION_1. Qui si effettua la prima operazione stabilita dall'algoritmo e già descritta precedentemente. In questo stato vengono attivati gli enable dei registri EN_REG_1, EN_REG_2, EN_REG_3, il SEL_MUX_1 viene portato al valore "01", viene reso attivo il segnale SUB_ADDER_1 che abilita la sottrazione e vengono disabilitati RD_MEM_A, CS_MEM_A. Nel colpo di clock successivo si passa allo stato OPERATION_2, in cui si effettua la seconda operazione; perciò rimane attivo il segnale SUB_ADDER_1 ma SEL_MUX_1 avrà la codifica "10" e si abilita EN_REG_4 per salvare il risultato di quest'ultimo calcolo.
Il colpo di clock dopo si effettua l'ultima operazione algebrica OPERATION_3, la somma tra il risultato di OPERATION_1 e OPERATION_2. Per questo motivo viene disabilitato il segnale SUB_ADDER_1, SEL_MUX_1 diventa "00" e viene asserito anche SEL_MUX_2. Inoltre verrà attivato anche il segnale EN_ROUND, che farà passare in ingresso all'arrotondatore il risultato dell'ultimo calcolo.
Col colpo di clock dopo si entra nello stato WRITE_MEM_B (WMB in figura 2.1), in cui si salva il dato in uscita dall'arrotondatore nella memoria B abilitando i segnali CS_MEM_B, WR_MEM_B.
Il colpo di clock successivo, a seconda che il dato in uscita dall'arrotondatore sia positivo o negativo, si passa allo stato POSITIVE_OUT (POS in figura 2.1) o allo stato NOT_POSITVE_OUT (nPOS in figura 2.1). Nel caso della simulazione modelsim (fi-

gura 2.3) si è preso in considerazione il caso negativo, mentre nel timing diagram creato in fase di progettazione 2.1 si è preso in esame il caso positivo. La differenza consiste nell'abilitare o meno il contatore dei numeri positivi (segnale EN_CNT_2). In entrambi i casi si abilita EN_CNT_1, in modo da far avanzare il contatore che punta agli indirizzi delle memorie. A questo punto il ciclo dovrebbe continuare fino a che non si sono elaborati tutti i dati della MEM_A.

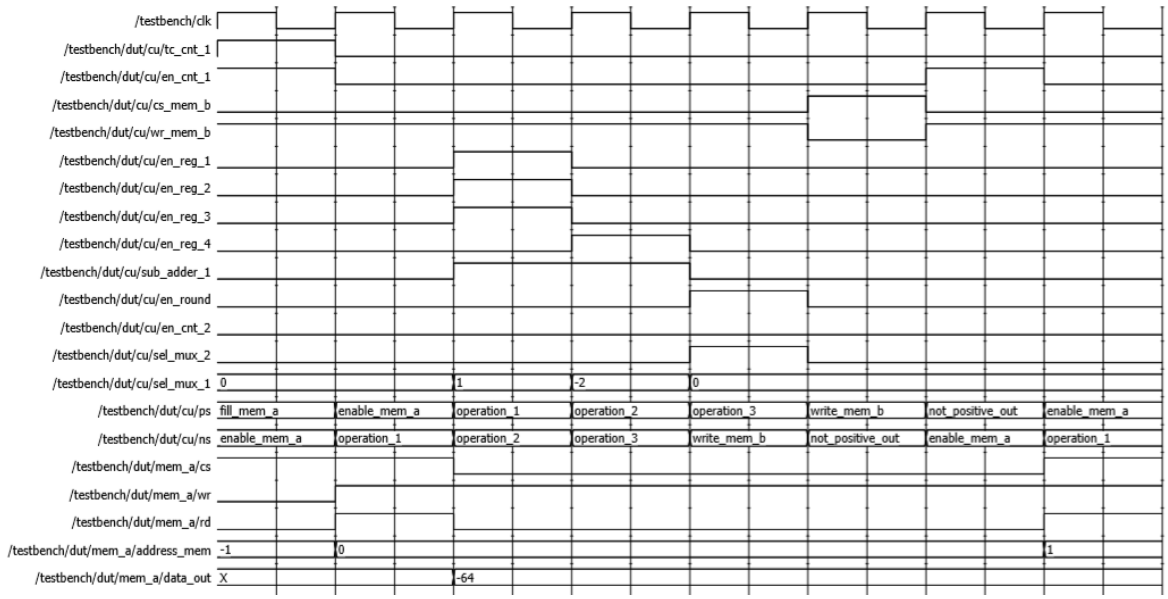

Figura 2.3: Stati ENABLE_MEM_A, OPERATIONS, WRITE_MEM_B e NOT_POSITIVE_OUT da simulazione in Modelsim

Il segnale TC_CNT_1 avverte se si è arrivati alla fine dei dati in ingresso e si finirà così nello stato END_STATE in cui è attivo il segnale DONE. Per ripartire, a questo punto, è necessario un nuovo START che riporti la macchina allo stato di IDLE.

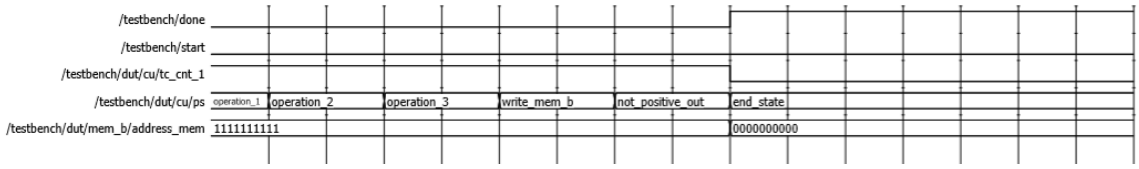

Figura 2.4: Stato END_STATE da simulazione in Modelsim

CAPITOLO 3

Testbench

Per una simulazione più accurata e che prendesse in esame più casi possibili, si è scritto un codice nel linguaggio Python che ha lo scopo di generare un file con ingressi casuali e un file con le uscite che ci si aspetta dalla simulazione in Modelsim dati gli ingressi appena generati. Il codice è riportato qui di seguito:

```
import random
import math
inputs = [0,0,0]
def main():
  random.seed()
  with open("inputs.txt", "w+") as f_in:
    with open("outputs.txt", "w+") as f_out:
      for i in range(1024):
        inputs[0] = inputs[1]
        inputs[1] = inputs[2]
        inputs[2] = random.randint(-128, 127)
        f_in.write(str(inputs[2]) + '\n')
        op1 = math.floor(0.5 * inputs[2])
        op2 = math.floor(3.75 * inputs[1])
        output = op1 + op2 - inputs[0]
        if output > 127:
          output = 127
        elif output < -128:
          output = -128
        if output >= 0:
          binary_output = format(output, 'b').zfill(8)
        else:
          binary_output = format(((-output)^255)+1, 'b')
        f_out.write(str(output) + ' ' + binary_output + '\n')
main()
```

Listing 1: Codice per la generazione di ingressi e uscite casuali

Si riportano, per completezza, anche le prime righe di un esempio di file generati:

```
-64
39
73
40
-22
123
-57
87
-47
115
...
```

Listing 2: Prime righe del file degli ingressi

```
-32 11100000
-128 10000000
127 01111111
127 01111111
66 01000010
-62 11000010
127 01111111
-128 10000000
127 01111111
-128 10000000
...
```

Listing 3: Prime righe del file delle uscite

Il file di testbench è riportato nel listing 4 e mostra l'utilizzo della libreria std.textio.all per l'apertura e lettura del file inputs.txt precedentemente generato. Ogni riga di quest'ultimo file viene usata per riempire un data_in che entrerà successivamente in MEM_A. Il file di testbench è stato commentato riga per riga per una più semplice consultazione.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use std.textio.all;

ENTITY testbench IS
END testbench;

ARCHITECTURE behavioural OF testbench IS
  SIGNAL CLK : STD_LOGIC;
  SIGNAL DATA_IN : SIGNED(7 DOWNTO 0);
  SIGNAL DONE : STD_LOGIC;
  SIGNAL OUTPUT_PORT : UNSIGNED(10 DOWNTO 0);
  SIGNAL RST : STD_LOGIC;
  SIGNAL START : STD_LOGIC;
  SIGNAL EN_READ : STD_LOGIC;

  component LAB6
    port(CLK,START,RST : in std_logic;
         DATA_IN : in signed(7 downto 0);
         DONE : out std_logic;
         OUTPUT_PORT : out unsigned(10 downto 0));
  end component;

  BEGIN
    DUT: LAB6 PORT MAP (CLK, START, RST, DATA_IN, DONE, OUTPUT_PORT);

  file_read: process(clk)
    variable input_line : line;
    variable input_tmp : integer := 0;
    file input_file : text; --creo un oggetto di tipo text che contiene l'intero file
    begin
      file_open(input_file, "inputs.txt", read_mode);
      if(rising_edge(clk)) then
        if EN_READ = '1' then
          if not endfile(input_file) then  --se non ha finito di leggere il file
            readline(input_file, input_line); --salva una riga del file in input_line
            read(input_line,input_tmp); --salva il numero nella riga in input_tmp
            data_in <= to_signed(input_tmp,8); --converti input_tmp in signed
            -- file_close(input_file);
          end if;
        end if;
      end if;
  end process;

  clock : PROCESS
    BEGIN
      clk <= '0';
      wait for 1 ns;
      clk <= '1';
      wait for 1 ns;
  END PROCESS clock;

  always : PROCESS
    BEGIN
      RST <= '0';
      START <= '0';
      EN_READ <= '0';

      wait for 4 ns;
      RST <= '1';

      wait for 2 ns;
      START <= '1';
      EN_READ <= '1';

      wait for 2ns;
      START <= '0';


      wait;
  END PROCESS;
END behavioural;
```

Listing 4: Testbench.vhd

CAPITOLO 4

Conclusioni

Il progetto è stato realizzato rispettando le specifiche fornite e considerando come fine ultimo il funzionamento complessivo della macchina. Inoltre, si è cercato di creare un algoritmo che minimizzasse il numero di componenti hardware. A titolo d'esempio con questo algoritmo è stato possibile fare a meno di moltiplicatori e shift register sfruttando semplici shift combinatori. Durante la fase di test sono stati riscontrati alcuni problemi a livello di timing che sono stati risolti attraverso uno studio accurato dei segnali nelle simulazioni su Modelsim. Da sottolineare come una progettazione rigorosa e ordinata, sia nella procedura sia nella nomenclatura, abbia aiutato nella risoluzione dei problemi, rendendo il tutto più immediato e gestibile.

CAPITOLO 5

Codici

5.1 Control unit

5.1.1 FSM

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
use ieee.numeric_std.all;

ENTITY FSM IS
  PORT( CLK,RST: IN STD_LOGIC;
        START, TC_CNT_1 : IN STD_LOGIC;
        OUT_ROUND : IN signed(7 DOWNTO 0);
        RESET, EN_CNT_1, CS_MEM_A, WR_MEM_A,
        RD_MEM_A, CS_MEM_B, WR_MEM_B, RD_MEM_B,  EN_REG_1,
        EN_REG_2, EN_REG_3, EN_REG_4, SUB_ADDER_1,
        EN_ROUND, EN_CNT_2, SEL_MUX_2, DONE : OUT STD_LOGIC;
        SEL_MUX_1 : OUT STD_LOGIC_VECTOR(1 DOWNTO 0));
END FSM;

ARCHITECTURE Behavioural OF FSM IS

  TYPE State_type IS (IDLE, FILL_MEM_A, ENABLE_MEM_A, OPERATION_1, OPERATION_2, OPERATION_3,
    ↪   WRITE_MEM_B, POSITIVE_OUT, NOT_POSITIVE_OUT, END_STATE);
  SIGNAL PS, NS: State_type;

  BEGIN

  stateProgression: PROCESS(CLK, START, PS, OUT_ROUND, TC_CNT_1) --individuo il next state
    BEGIN
    CASE PS IS
      WHEN IDLE=>
        IF(START = '1') THEN
          NS <= FILL_MEM_A;
        ELSE
          NS <= IDLE;
        END IF;

      WHEN FILL_MEM_A=>
        IF(TC_CNT_1 = '1') THEN
          NS <= ENABLE_MEM_A;
        ELSE
          NS <= FILL_MEM_A;
        END IF;

      WHEN ENABLE_MEM_A=>
        NS <= OPERATION_1;

      WHEN OPERATION_1=>
        NS <= OPERATION_2;

      WHEN OPERATION_2=>
        NS <= OPERATION_3;

      WHEN OPERATION_3=>
        NS <= WRITE_MEM_B;

      WHEN WRITE_MEM_B=>
        IF(OUT_ROUND(7) = '0' and not (OUT_ROUND = "00000000")) THEN
          NS <= POSITIVE_OUT;
        ELSE
          NS <= NOT_POSITIVE_OUT;
        END IF;

      WHEN POSITIVE_OUT=>
```

```
      IF(TC_CNT_1 = '1') THEN
        NS <= END_STATE;
      ELSE
        NS <= ENABLE_MEM_A;
      END IF;

    WHEN NOT_POSITIVE_OUT=>
      IF(TC_CNT_1 = '1') THEN
        NS <= END_STATE;
      ELSE
        NS <= ENABLE_MEM_A;
      END IF;

    WHEN END_STATE=>
      IF(START = '1') THEN
        NS <= IDLE;
      ELSE
        NS <= END_STATE;
      END IF;

    WHEN OTHERS =>
      NS <= IDLE;

  END CASE;
END PROCESS;

stateUpdate: PROCESS(CLK,RST) -- aggiorno present state
  BEGIN
    IF RST = '0' then
      PS <= IDLE;
    ELSIF CLK'EVENT AND CLK='1' THEN
      PS<=NS;
    END IF;
END PROCESS;

outputUpdate: PROCESS(PS, CLK) -- aggiorno le uscite
  BEGIN

  RESET <= '1';
  EN_CNT_1 <= '0';
  CS_MEM_A <= '0';
  WR_MEM_A <= '1';
  RD_MEM_A <= '0';
  CS_MEM_B <= '0';
  WR_MEM_B <= '1';
  RD_MEM_B <= '0';
  EN_REG_1 <= '0';
  EN_REG_2 <= '0';
  EN_REG_3 <= '0';
  EN_REG_4 <= '0';
  SUB_ADDER_1 <= '0';
  EN_ROUND <= '0';
  EN_CNT_2 <= '0';
  SEL_MUX_1 <= "00";
  SEL_MUX_2 <= '0';
  DONE <= '0';

  CASE PS IS
    WHEN IDLE=>
      RESET <= '0';

    WHEN FILL_MEM_A=>
      CS_MEM_A <= '1';
      WR_MEM_A <= '0';
      EN_CNT_1 <= '1';

    WHEN ENABLE_MEM_A=>
      CS_MEM_A <= '1';
      RD_MEM_A <= '1';

    WHEN OPERATION_1=>
      SEL_MUX_1 <= "01";
      SUB_ADDER_1 <= '1';
      EN_REG_1 <= '1';
      EN_REG_2 <= '1';
      EN_REG_3 <= '1';

    WHEN OPERATION_2=>
      SEL_MUX_1 <= "10";
      EN_REG_4 <= '1';
      SUB_ADDER_1 <= '1';

    WHEN OPERATION_3=>
      SEL_MUX_1 <= "00";
      SEL_MUX_2 <= '1';
      EN_ROUND <= '1';

    WHEN WRITE_MEM_B=>
      CS_MEM_B <= '1';
      WR_MEM_B <= '0';

    WHEN POSITIVE_OUT=>
      EN_CNT_2 <= '1';
      EN_CNT_1 <= '1';

    WHEN NOT_POSITIVE_OUT=>
      EN_CNT_1 <= '1';

    WHEN END_STATE=>
      DONE <= '1';

    WHEN OTHERS =>

  END CASE;
END PROCESS;
```

```
END Behavioural;
```

5.2 Datapath

5.2.1 Adder

```
-- Sommatore e sottratore
-- SUB_ADDER seleziona quale delle due operazioni effettuare
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity Adder is
  generic(N : integer:=11);
  port(SUB_ADDER: in std_logic;
       IN_1,IN_2 : in signed(N-1 downto 0);
       DATA_OUT : out signed(N-1 downto 0));
end entity;

architecture behaviour of Adder is

begin
  DATA_OUT <= IN_1 + IN_2 when SUB_ADDER = '0' else
              (IN_1 - IN_2);
end architecture;
```

5.2.2 Counter_Nbit

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

entity counter_Nbit is
  generic(N : integer := 16);
  port(EN, CLK, RST_n : in std_logic;
       OUT_CNT : out std_logic_vector(N-1 downto 0));
end entity;

architecture structural of counter_Nbit is
  component tflipflop
    port (t, clock, clear : in std_logic;
          Q_neg, Q: buffer std_logic);
  end component;

  signal Q, Q_neg, T : std_logic_vector(N-1 downto 0);

  begin
  T(0) <= EN;
  GEN_CNT: for i in 0 to N-1 generate
    GEN_IF: if i > 0 generate
      T(i) <= T(i-1) and Q(i-1);
    end generate;
    TFF_X : tflipflop port map(T(i), CLK, RST_n, Q_neg(i), Q(i));
  end generate;
  OUT_CNT <= Q;
end architecture;
```

5.2.3 Counter_positivi

```
-- Composto da un sommatore ed un registro
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
```

```
entity Counter_positivi is
  port(CLK,RST,EN : in std_logic;
       OUT_CNT : out unsigned(10 downto 0));
end entity;

architecture behaviour of Counter_positivi is
  component reg_unsigned
    GENERIC ( N : integer:=11);
    PORT (EN : in std_logic;
          R : IN UNSIGNED(N-1 DOWNTO 0);
          Clock, Resetn : IN STD_LOGIC;
          Q : OUT UNSIGNED(N-1 DOWNTO 0));
  end component;

  component HA
    generic(N : integer:=11);
    port(IN_1_HA,IN_2_HA : in unsigned(N-1 downto 0);
         OUT_HA : out unsigned(N-1 downto 0));
  end component;

  signal OUT_REG, OUT_HA : unsigned(10 downto 0);

  begin

  HalfAdder : HA generic map(N=>11)
                 port map("00000000001", OUT_REG, OUT_HA);
  reg : reg_unsigned generic map(N =>11)
             port map(EN, OUT_HA, CLK, RST, OUT_REG);

  OUT_CNT <= OUT_REG;
end architecture;
```

5.2.4 D_FF

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY dflipflop IS
  PORT (D, Clock, Resetn : IN STD_LOGIC;
        Q : OUT STD_LOGIC);
END ENTITY;

ARCHITECTURE Behavior OF dflipflop IS
BEGIN
  PROCESS (Clock, Resetn)
    BEGIN
      IF (Clock'EVENT AND Clock = '1') THEN
        IF (Resetn = '0') THEN -- synchronous clear
          Q <= '0';
        ELSE
          Q <= D;
        END IF;
      END IF;
  END PROCESS;
END Behavior;
```

5.2.5 T_FF

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

entity tflipflop is
  port (t, clock, clear : in std_logic;
        Q_neg, Q: buffer std_logic);
end entity;

architecture structural of tflipflop is
  signal D : std_logic;
  component dflipflop
    port (D, Clock, Resetn : in STD_LOGIC;
          Q : out STD_LOGIC);
  end component;

  begin
  D <= (not(t) and Q) or (t and Q_neg);
  ff : dflipflop port map(D, clock, clear, Q);
```

```
  Q_neg <= not(Q);
end architecture;
```

5.2.6 HA

```
-- RST -> Reset attivo basso
-- EN -> Enable
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity HA is
  generic(N : integer:=11);
  port(IN_1_HA,IN_2_HA : in unsigned(N-1 downto 0);
       OUT_HA : out unsigned(N-1 downto 0));
end entity;

architecture behaviour of HA is

begin
  OUT_HA <= IN_1_HA + IN_2_HA;
end architecture;
```

5.2.7 mux2to1

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

entity Mux2to1 is
  port(IN1, IN2 : in signed(11 downto 0);
       SEL : in std_logic;
       OUT_DATA : out signed(11 downto 0));
end entity;

architecture behavioural of Mux2to1 is
begin
    OUT_DATA <= IN1 when SEL = '0' else
                IN2;
end architecture;
```

5.2.8 mux4to1

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

entity Mux4to1 is
  port(IN1, IN2, IN3, IN4 : in signed(11 downto 0);
       SEL : in std_logic_vector(1 downto 0);
       OUT_DATA : out signed(11 downto 0));
end entity;

architecture behaviour of Mux4to1 is
begin
    OUT_DATA <=  IN1 when SEL = "00" else
                 IN2 when SEL = "01" else
                 IN3 when SEL = "10" else
                 IN4;
end architecture;
```

5.2.9 Reg_signed

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY reg_sig IS
  GENERIC ( N : integer:=11);
  PORT (CLK, EN, RST : IN STD_LOGIC;
        D : IN SIGNED(N-1 DOWNTO 0);
        Q : OUT SIGNED(N-1 DOWNTO 0));
END reg_sig;

ARCHITECTURE Behavior OF reg_sig IS
BEGIN
  PROCESS (CLK, RST)
  BEGIN
    IF (RST = '0') THEN
      Q <= (OTHERS => '0');
    ELSIF (CLK'EVENT AND CLK = '1') THEN
      IF EN = '1' THEN
        Q <= D;
      END IF;
    END IF;
  END PROCESS;
END Behavior;
```

5.2.10 Reg_unsigned

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY reg_unsigned IS
GENERIC ( N : integer:=11);
PORT (EN : in std_logic;
      R : IN UNSIGNED(N-1 DOWNTO 0);
      Clock, Resetn : IN STD_LOGIC;
      Q : OUT UNSIGNED(N-1 DOWNTO 0));
END reg_unsigned;

ARCHITECTURE Behavior OF reg_unsigned IS
BEGIN
  PROCESS (Clock, Resetn)
  BEGIN
    IF (Resetn = '0') THEN
      Q <= (OTHERS => '0');
    ELSIF (Clock'EVENT AND Clock = '1') THEN
      IF EN = '1' THEN
        Q <= R;
      END IF;
    END IF;
  END PROCESS;
END Behavior;
```

5.2.11 Rounder

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

entity rounder is
  port(CLK, EN, RST : in std_logic;
       IN_ROUND : in signed(11 downto 0);
       OUT_ROUND : out signed(7 downto 0));
end entity;

architecture behaviour of rounder is
signal Comparison : std_logic;
begin
  Comparison <= (IN_ROUND(11) xor IN_ROUND(10)) xor (IN_ROUND(9) xor IN_ROUND(8));
  process(CLK,EN,RST,Comparison,IN_ROUND)
    begin

    if RST = '0' then
```

```
      OUT_ROUND <= (others =>'0');
    elsif CLK'event and CLK = '1' then
      if EN = '1' then
        if(Comparison = '1') then
          if IN_ROUND(11) = '1' then
            OUT_ROUND <= "10000000";
          else
            OUT_ROUND <= "01111111";
          end if;
         else
          OUT_ROUND <= IN_ROUND(11) & IN_ROUND(6 downto 0);
        end if;
      end if;
     end if;
  end process;
end architecture;
```

5.2.12 SHIFT_L_4

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

entity shift_l_4 is
port (in_shift_l_4: in signed (7 downto 0);
                        out_shift_l_4: out signed (11 downto 0));
end entity;

architecture structural of shift_l_4 is
begin
out_shift_l_4(11)<= in_shift_l_4(7);
out_shift_l_4(10)<= in_shift_l_4(6);
out_shift_l_4(9)<= in_shift_l_4(5);
out_shift_l_4(8)<= in_shift_l_4(4);
out_shift_l_4(7)<= in_shift_l_4(3);
out_shift_l_4(6)<= in_shift_l_4(2);
out_shift_l_4(5)<= in_shift_l_4(1);
out_shift_l_4(4)<= in_shift_l_4(0);
out_shift_l_4(3)<= '0';
out_shift_l_4(2)<= '0';
out_shift_l_4(1)<= '0';
out_shift_l_4(0)<= '0';
end architecture;
```

5.2.13 SHIFT_R_1

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

entity shift_r_1 is
        port (in_shift_r_1: in signed (7 downto 0);
                          out_shift_r_1: out signed (11 downto 0));
end entity;

architecture structural of shift_r_1 is
begin
out_shift_r_1(11)<=in_shift_r_1(7);
out_shift_r_1(10)<=in_shift_r_1(7);
out_shift_r_1(9)<=in_shift_r_1(7);
out_shift_r_1(8)<=in_shift_r_1(7);
out_shift_r_1(7)<=in_shift_r_1(7);
out_shift_r_1(6)<=in_shift_r_1(7);
out_shift_r_1(5)<=in_shift_r_1(6);
out_shift_r_1(4)<=in_shift_r_1(5);
out_shift_r_1(3)<=in_shift_r_1(4);
out_shift_r_1(2)<=in_shift_r_1(3);
out_shift_r_1(1)<=in_shift_r_1(2);
out_shift_r_1(0)<=in_shift_r_1(1);
end architecture;
```

5.2.14 SHIFT_R_2

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

entity shift_r_2 is
        port(in_shift_r_2: in signed (11 downto 0);
                           out_shift_r_2: out signed (11 downto 0));
end entity;

architecture structural of shift_r_2 is
begin
out_shift_r_2(11)<= in_shift_r_2(11);
out_shift_r_2(10)<= in_shift_r_2(11);
out_shift_r_2(9)<= in_shift_r_2(11);
out_shift_r_2(8)<= in_shift_r_2(10);
out_shift_r_2(7)<= in_shift_r_2(9);
out_shift_r_2(6)<= in_shift_r_2(8);
out_shift_r_2(5)<= in_shift_r_2(7);
out_shift_r_2(4)<= in_shift_r_2(6);
out_shift_r_2(3)<= in_shift_r_2(5);
out_shift_r_2(2)<= in_shift_r_2(4);
out_shift_r_2(1)<= in_shift_r_2(3);
out_shift_r_2(0)<= in_shift_r_2(2);

end architecture;
```

5.2.15 Datapath

```
--RESET attivo basso
--ENABLE attivi alti

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

entity Datapath is
 port(CLK, RST_n: in std_logic; --CLOCK,RESET
    DATA_OUT_MEM_A : in signed(7 downto 0);  --Dato in uscita dalla memoria A
   EN_REG_1, EN_REG_2, EN_REG_3, EN_REG_4, SEL_MUX2 : in std_logic;
   SEL_MUX1 : in std_logic_vector(1 downto 0); --Selettori MUX1 e MUX2 (rispettivamente in IN1 ed IN2)
    SUB_ADDER : in std_logic;
   EN_CNT_1, EN_CNT_2 : in std_logic;
   EN_ROUND : in std_logic;

   TC_CNT_1 : out std_logic;  --Terminal counter a 1023
   OUT_ROUND : out signed(7 downto 0);
   ADDRESS_MEM : out std_logic_vector(9 downto 0);  --Uscita per l'indirizzo sia della Memoria A che B
   OUTPUT_PORT : out unsigned(10 downto 0) --Uscita che indica le Y positive calcolate
    );
end entity;

architecture behaviour of Datapath is

component shift_r_2
 port(in_shift_r_2: in signed (11 downto 0);
    out_shift_r_2: out signed (11 downto 0));
end component;

component shift_r_1
 port (in_shift_r_1: in signed (7 downto 0);
     out_shift_r_1: out signed (11 downto 0));
end component;

component shift_l_4
 port (in_shift_l_4: in signed (7 downto 0);
    out_shift_l_4: out signed (11 downto 0));
end component;

component Adder
 generic(N : integer:=11);
 port(SUB_ADDER: in std_logic;
      IN_1,IN_2 : in signed(N-1 downto 0);
      DATA_OUT : out signed(N-1 downto 0));
end component;

component counter_Nbit
 generic(N : integer := 16);
 port(EN, CLK, RST_n : in std_logic;
     OUT_CNT : out std_logic_vector(N-1 downto 0));
end component;

component Counter_positivi
 port(CLK,RST,EN : in std_logic;
```

```
     OUT_CNT : out unsigned(10 downto 0));
end component;

component Mux4to1
 port(IN1, IN2, IN3, IN4 : in signed(11 downto 0);
      SEL : in std_logic_vector(1 downto 0);
      OUT_DATA : out signed(11 downto 0));
end component;

component Mux2to1
 port(IN1, IN2 : in signed(11 downto 0);
      SEL : in std_logic;
      OUT_DATA : out signed(11 downto 0));
end component;

component reg_sig
 GENERIC ( N : integer:=11);
 PORT (CLK, EN, RST : IN STD_LOGIC;
    D : IN SIGNED(N-1 DOWNTO 0);
    Q : OUT SIGNED(N-1 DOWNTO 0));
end component;

component rounder
 port(CLK, EN, RST : in std_logic;
     IN_ROUND : in signed(11 downto 0);
    OUT_ROUND : out signed(7 downto 0));
end component;

signal OUT_CNT_1 : std_logic_vector(9 downto 0);
signal Q_REG_3, Q_REG_4, OUT_SHIFT_R_1, OUT_SHIFT_L_4, OUT_SHIFT_R_2, IN_1_MUX_2, IN_2_MUX_2 :
 ↪   signed(11 downto 0);
signal OUT_DATA_MUX1, OUT_DATA_MUX2, OUT_ADDER: signed(11 downto 0);
signal D_REG_1, Q_REG_1, Q_REG_2 : signed(7 downto 0);
signal OUT_CNT_2 : unsigned(10 downto 0);

begin

 shift_right_2 : shift_r_2 port map (Q_REG_4, out_shift_r_2);
 shift_right_1 : shift_r_1 port map(data_out_mem_a, out_shift_r_1);
 shift_left_4 : shift_l_4 port map(Q_REG_2, out_shift_l_4);

 D_REG_1 <= DATA_OUT_MEM_A;
 REG_1 : reg_sig generic map(N=>8) port map(CLK, EN_REG_1, RST_n, D_REG_1, Q_REG_1);
 REG_2 : reg_sig generic map(N=>8) port map(CLK, EN_REG_2, RST_n, Q_REG_1, Q_REG_2);

 mux_1: mux4to1 port map(Q_REG_3, OUT_SHIFT_R_1, OUT_SHIFT_L_4, "000000000000", SEL_MUX1,
  ↪   OUT_DATA_MUX1);
 IN_2_MUX_2 <= Q_REG_2(7) & Q_REG_2(7) & Q_REG_2(7) & Q_REG_2(7) & Q_REG_2;
 mux_2: mux2to1 port map(IN_2_MUX_2, OUT_SHIFT_R_2, SEL_MUX2, OUT_DATA_MUX2);

 sommatore : adder generic map(N=>12) port map(SUB_ADDER, OUT_DATA_MUX1, OUT_DATA_MUX2, OUT_ADDER);
 REG_3 :  reg_sig generic map(N=>12) port map(CLK, EN_REG_3, RST_n, OUT_ADDER, Q_REG_3);
 REG_4 :  reg_sig generic map(N=>12) port map(CLK, EN_REG_4, RST_n, OUT_ADDER, Q_REG_4);

 rounding: rounder port map(CLK, EN_ROUND, RST_n, OUT_ADDER, OUT_ROUND);

 cnt_1 : counter_Nbit generic map(N=>10) port map(EN_CNT_1, CLK, RST_n, OUT_CNT_1);
 cnt_2: counter_positivi port map(CLK, RST_n, EN_CNT_2, OUT_CNT_2);

 OUTPUT_PORT <= OUT_CNT_2;
 ADDRESS_MEM <= OUT_CNT_1;

 --SEGNALE DEL TERMINAL COUNTER
 TC_CNT_1 <= OUT_CNT_1(9) and OUT_CNT_1(8) and OUT_CNT_1(7) and OUT_CNT_1(6) and
       OUT_CNT_1(5) and OUT_CNT_1(4) and OUT_CNT_1(3) and OUT_CNT_1(2) and
       OUT_CNT_1(1) and OUT_CNT_1(0);
end architecture;
```

5.3 Memoria

```
-- Memoria di signed (1024 righe 8 colonne)
-- CLK -> Clock
-- RST -> Reset
-- CS -> Enable
-- WR -> Write (active low)
-- RD -> Read
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity Memory is
        port(CLK, CS, WR, RD : in std_logic;
             ADDRESS_MEM : in std_logic_vector(9 downto 0);
             DATA_IN : in signed(7 downto 0);
             DATA_OUT : out signed(7 downto 0));
```

```
end entity;

architecture behaviour of Memory is

type MEM is array(0 to 1023) of signed(7 downto 0);
signal MEMORIA : MEM;

begin
        RW : process(CLK, CS, RD, WR)
                begin
                if CLK'event and CLK = '1' then
                        if CS = '1' then
                                if WR = '0' then
                                        MEMORIA(to_integer(unsigned(ADDRESS_MEM))) <= DATA_IN;
                                end if;
                                if RD = '1' then
                                        DATA_OUT <= MEMORIA(to_integer(unsigned(ADDRESS_MEM)));
                                end if;
                        end if;
                end if;
        end process;
end architecture;
```

5.4 Sistema Completo

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

entity LAB6 is
 port(CLK, START, RST : in std_logic;
    DATA_IN : in signed(7 downto 0);
    DONE : out std_logic;
    OUTPUT_PORT : out unsigned(10 downto 0));
end entity;

architecture behavioural of LAB6 is
 component Datapath
 port(CLK, RST_n: in std_logic; --CLOCK,RESET
   DATA_OUT_MEM_A : in signed(7 downto 0);  --Dato in uscita dalla memoria A
   EN_REG_1, EN_REG_2, EN_REG_3, EN_REG_4, SEL_MUX2 : in std_logic;
   SEL_MUX1 : in std_logic_vector(1 downto 0); --Selettori MUX1 e MUX2 (rispettivamente in IN1 ed IN2)
   SUB_ADDER : in std_logic;
   EN_CNT_1, EN_CNT_2 : in std_logic;
   EN_ROUND : in std_logic;

   TC_CNT_1 : out std_logic;  --Terminal counter a 1023
   OUT_ROUND : out signed(7 downto 0);
   ADDRESS_MEM : out std_logic_vector(9 downto 0);  --Uscita per l'indirizzo sia della Memoria A che B
   OUTPUT_PORT : out unsigned(10 downto 0)); --Uscita che indica le Y positive calcolate
 end component;

 component FSM
  PORT(CLK,RST: IN STD_LOGIC;
    START, TC_CNT_1 : IN STD_LOGIC;
    OUT_ROUND : IN signed(7 DOWNTO 0);
    RESET, EN_CNT_1, CS_MEM_A, WR_MEM_A,
    RD_MEM_A, CS_MEM_B, WR_MEM_B, RD_MEM_B, EN_REG_1,
    EN_REG_2, EN_REG_3, EN_REG_4, SUB_ADDER_1,
    EN_ROUND, EN_CNT_2, SEL_MUX_2, DONE : OUT STD_LOGIC;
    SEL_MUX_1 : OUT STD_LOGIC_VECTOR(1 DOWNTO 0));
 end component;

 component Memory
  port(CLK, CS, WR, RD : in std_logic;
     ADDRESS_MEM : in std_logic_vector(9 downto 0);
     DATA_IN : in signed(7 downto 0);
     DATA_OUT : out signed(7 downto 0));
 end component;

 signal DATA_OUT_MEM_A, OUT_ROUND, DATA_OUT_MEM_B: signed (7 downto 0);
 signal RESET, EN_REG_1, EN_REG_2, EN_REG_3,
     EN_REG_4, SUB_ADDER, EN_CNT_1, EN_CNT_2, EN_ROUND, TC_CNT_1, CS_MEM_A,
     WR_MEM_A, RD_MEM_A, CS_MEM_B, WR_MEM_B, RD_MEM_B, SEL_MUX2: std_logic;
 signal SEL_MUX1: std_logic_vector(1 downto 0);
 signal ADDRESS_MEM: std_logic_vector(9 downto 0);

 begin
  DP: Datapath port map(CLK, RESET, DATA_OUT_MEM_A, EN_REG_1, EN_REG_2, EN_REG_3, EN_REG_4, SEL_MUX2,
    ↪   SEL_MUX1, SUB_ADDER, EN_CNT_1, EN_CNT_2, EN_ROUND, TC_CNT_1, OUT_ROUND, ADDRESS_MEM,
    ↪   OUTPUT_PORT);
  CU: FSM port map(CLK, RST, START, TC_CNT_1,OUT_ROUND, RESET, EN_CNT_1, CS_MEM_A, WR_MEM_A, RD_MEM_A,
    ↪   CS_MEM_B, WR_MEM_B, RD_MEM_B, EN_REG_1, EN_REG_2, EN_REG_3, EN_REG_4, SUB_ADDER, EN_ROUND,
    ↪   EN_CNT_2, SEL_MUX2, DONE, SEL_MUX1);
  MEM_A: Memory port map (CLK, CS_MEM_A, WR_MEM_A, RD_MEM_A, ADDRESS_MEM, DATA_IN, DATA_OUT_MEM_A);
```

```
  MEM_B: Memory port map (CLK, CS_MEM_B, WR_MEM_B, RD_MEM_B, ADDRESS_MEM, OUT_ROUND, DATA_OUT_MEM_B);
end architecture;
```