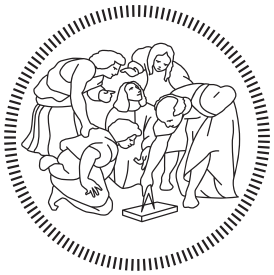

Politecnico di Milano

Prova Finale (Progetto di Reti Logiche)

Jacopo Sacramone (Codice Persona: 10752157 - Matricola: 983764)
Federico Pinto (Codice Persona: 10766381 - Matricola: 987201)

supervisore
Prof. Gianluca Palermo

Indice

1 Introduzione

1.1 Descrizione

Il progetto si propone di sviluppare un modulo hardware, descritto in **VHDL**, che mira a risolvere una specifica sfida di calcolo combinatorio e di gestione delle risorse. In particolare l'obiettivo è l'implementazione di un sistema che, attraverso l'uso di un'interfaccia ben definita, gestisce l'ingresso e la trasformazione di segnali digitali secondo parametri predeterminati.

1.2 Specifiche generali

Il sistema deve gestire un array di parole, ciascuna di dimensione 16 bit e memorizzata consecutivamente in memoria a partire da un indirizzo iniziale specificato (ADD). Ogni parola è composta da due byte: il primo byte rappresenta il valore della parola W e il secondo byte un valore di credibilità C.

La sequenza è costituita da K parole W con valori che variano da 0 a 255. Un valore di 0 in W non rappresenta un valore effettivo ma indica che "il valore non è specificato". Questo richiede una logica di completamento in cui gli zeri sono sostituiti dall'ultimo valore valido (non zero) precedente nella sequenza. Il valore di credibilità C associato a ogni parola W è inizializzato a 31 quando W è diverso da zero. Se W è zero, C viene decrementato di uno rispetto al valore precedente fino a un minimo di 0, e non viene ulteriormente decrementato una volta raggiunto tale minimo.

L'obiettivo principale del modulo è di leggere la sequenza di dati, eseguire la sostituzione degli zeri con l'ultimo valore valido letto e aggiornare il valore di credibilità in modo conforme alle regole stabilite.

Figura 1: Sequenza iniziale

Figura 2: Sequenza finale

1.3 Interfaccia del componente

Il componente da implementare ha un'interfaccia così definita:

```
entity project_reti_logiche is
PORT(
    i_clk       : IN  std_logic;
    i_rst       : IN  std_logic;
    i_start     : IN  std_logic;
    i_add       : IN  std_logic_vector(15 DOWNTO 0);
    i_k         : IN  std_logic_vector(9 DOWNTO 0);

    o_done      : OUT std_logic;

    o_mem_addr  : OUT std_logic_vector(15 DOWNTO 0);
    i_mem_data  : IN  std_logic_vector(7 DOWNTO 0);
    o_mem_data  : OUT std_logic_vector(7 DOWNTO 0);
    o_mem_we    : OUT std_logic;
    o_mem_en    : OUT std_logic
  );
end project_reti_logiche;
```

- `i_clk` è il segnale di `CLOCK` in ingresso generato dal Test Bench
- `i_rst` è il segnale di `RESET` che inizializza la macchina pronta per ricevere il primo segnale di `START`;
- `i_start` è il segnale di `START` generato dal Test Bench
- `i_k` è il segnale (vettore) K generato dal Test Bench rappresentante la lunghezza della sequenza;
- `i_add` è il segnale (vettore) ADD generato dal Test Bench che rappresenta l'indirizzo dal quale parte la sequenza da elaborare;
- `o_done` è il segnale `DONE` di uscita che comunica la fine dell'elaborazione;
- `o_mem_addr` è il segnale (vettore) di uscita che manda l'indirizzo alla memoria;
- `i_mem_data` è il segnale (vettore) che arriva dalla memoria e contiene il dato in seguito ad una richiesta di lettura;
- `o_mem_data` è il segnale (vettore) che va verso la memoria e contiene il dato che verrà successivamente scritto;
- `o_mem_en` è il segnale di `ENABLE` da dover mandare alla memoria per poter comunicare (sia in lettura che in scrittura);
- `o_mem_we` è il segnale di `WRITE ENABLE` da dover mandare alla memoria (=1) per poter scriverci. Per leggere da memoria esso deve essere 0.

2 Architettura

2.1 Macchina a stati

La macchina è composta da 14 stati (sono omessi tutte le transizioni che da qualsiasi stato tornano allo **stato di reset** quando `i_rst = 1 o i_start = 0`).

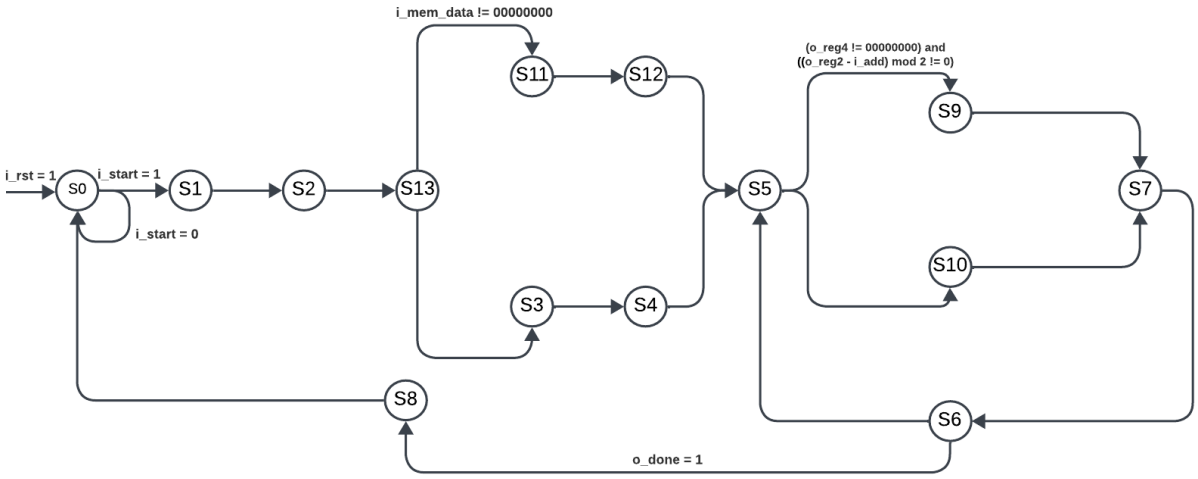

Figura 3: Macchina a stati con le condizioni di transizione

2.1.1 `Reset` state (`S0`)

Stato iniziale in attesa del segnale di `i_start`. Se `i_rst` viene portato a uno o `i_start` viene portato a zero durante il processo, si ritorna a questo stato.

2.1.2 `Initial reading` states (`S1/S2/S13`)

Gestiscono la **lettura del primo indirizzo di memoria**. `S13` funge da **"buffer"**, dando il tempo al dato letto dalla memoria al dato indirizzo di arrivare al componente .

2.1.3 `First zero` states (`S11/S12`)

Gestiscono il caso in cui il primo dato letto sia **zero**, un `edge case` che necessita di un trattamento diverso.

2.1.4 `Normal processing` states (`S3/S4`)

Gestiscono il flusso dei dati quando il primo carattere letto dalla memoria è **diverso da zero**.

2.1.5 `Continuous read` state (`S5`)

Gestisce la **lettura continua dei dati dalla memoria**. Il dato corrente viene caricato e controllato per determinare l'azione successiva, basata su specifici criteri di valutazione.

2.1.6 `Credibility and Address` states (`S6/S10`)

Gestiscono l'incremento dell'indirizzo di memoria, per la lettura successiva di dati, e l'indicatore di credibilità.

2.1.7 `Data writing` state (`S7`)

E' responsabile della **scrittura in memoria** dei dati elaborati.

2.1.8 `Counter reset` state (`S9`)

Resetta la credibilità, necessario dopo la lettura di un numero nuovo diverso da zero, La condizione per entrare in questo stato è che il registro dove passeranno tutte le parole sia diverso da zero (questo perchè il valore di credibilità viene resettato in quel caso) e per gestire l'edge case in cui nella stringa di ingresso sia presente un valore diverso da zero nel byte di credibilità è stata aggiunta in AND una condizione che verifica se lo spiazzamento tra indirizzo corrente e iniziale sia dispari, garantendo che la transizione venga fatta quando viene letto un dato (una parola effettiva) in posizione per la quale la transizione può avvenire.

2.1.9 `Done` state (`S8`)

Stato che da il tempo ai segnali di tornare allo stato per cui una volta arrivati con il ciclo di clock successivo in `S0` non si abbiano problemi nel caso in cui una nuova sequenza debba partire.

2.2 Datapath

Il circuito logico è diviso in 4 blocchi. Nella progettazione degli `Address` block e `Data` block, è stata adottata una specifica strategia implementativa che consiste nell'includere un percorso diretto dall'ingresso fino all'ultimo multiplexer. Questa decisione è motivata dalla necessità di migliorare l'efficienza temporale durante operazioni critiche, come la prima lettura dei dati. La scelta di avere un collegamento diretto permette di bypassare il caricamento del dato nel registro quando non serve, rendendo il dato immediatamente disponibile in uscita.

2.2.1 `Address` block

Gestisce l'indirizzamento della memoria. Il circuito include vari componenti per determinare l'indirizzo di memoria corretto dove verranno scritti i dati elaborati dal `Data` **block**

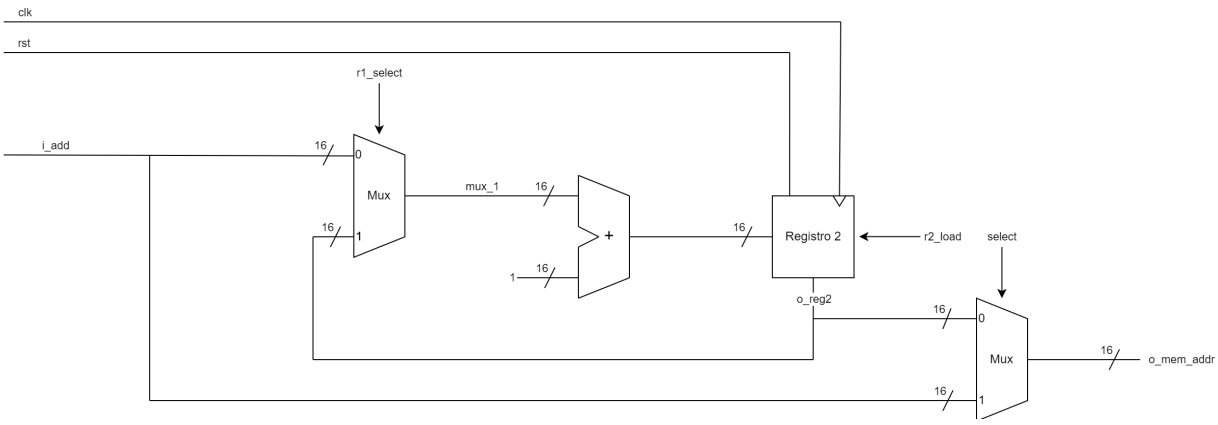

Figura 4: Address block

2.2.2 `Counter` block

Progettata per gestire e tracciare la lunghezza della stringa di numeri data come input. L'output di questa sezione del circuito è il segnale `o_done`, che viene impostato su 1 esclusivamente al completamento dell'elaborazione dell'intera sequenza di numeri.

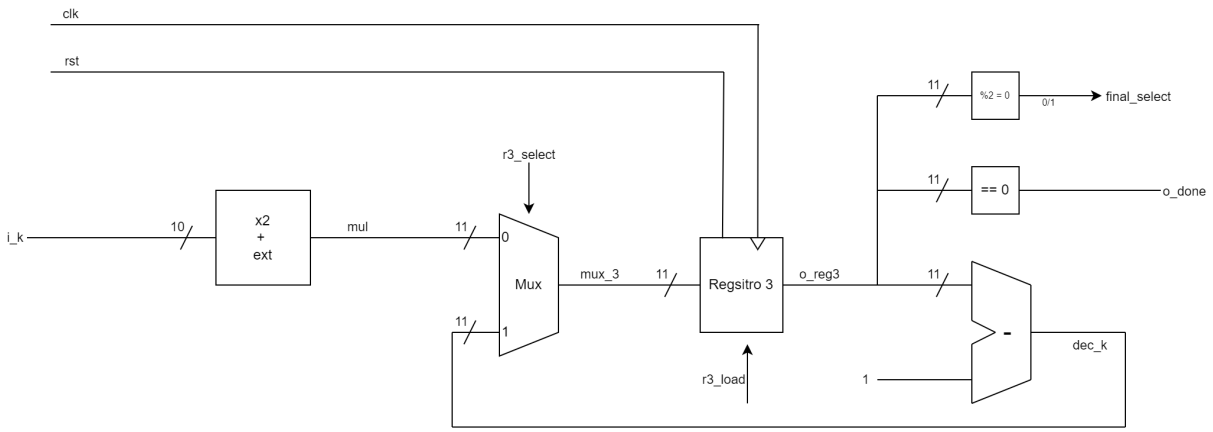

Figura 5: Counter block

Il blocco produce anche il segnale `final_select` che modifica il suo valore in base alla parità o disparità di k. Questo segnale viene impiegato nel `Data` **block** perchè permette di sapere dove si sta leggendo (se in una posizione dove deve essere presente una parola o un valore di credibilità) e questo è fondamentale per compiere le giuste operazioni sul dato .
Il segnale k è stato esteso a 11 bit immediatamente dopo il suo ingresso. Questa modifica, per come è stata realizzata la struttura del `Counter` block, permette di gestire il caso in cui il numero di parole k supera 512, in questo caso infatti non basterebbero 10 bit.

2.2.3 `Data` and `Credibilty` block

Progettato per manipolare e processare i dati provenienti dalla memoria insieme al valore di credibilità, al fine di stabilire l'informazione finale che verrà scritta in una specifica cella di memoria elaborata dall'`Address` **block**
Il demultiplexer è stato utilizzato per ottimizzare il salvataggio e la riscrittura delle parole. Usando un demultiplexer, è possibile controllare il processo di gestione delle parole, rendendolo più diretto.

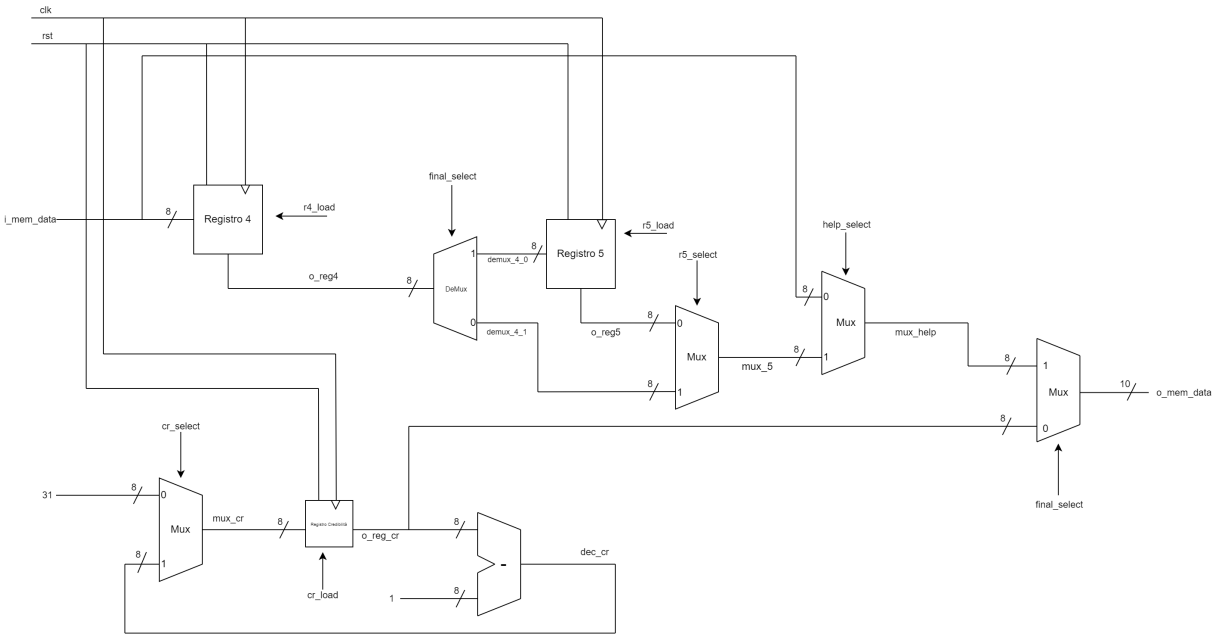

Figura 6: Data and Credibilty block

3 Risultati sperimentali

3.1 Sintesi

Il progetto è stato testato con la versione **Vivado 2018.3.1**, utilizzando come FPGA target **xc7a12ticsg325-1L**, e ha generato il seguente report di sintesi:

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	94	0	8000	1.18
LUT as Logic	94	0	8000	1.18
LUT as Memory	0	0	5000	0.00
Slice Registers	55	0	16000	0.34
Register as Flip Flop	55	0	16000	0.34
Register as Latch	0	0	16000	0.00
F7 Muxes	0	0	7300	0.00
F8 Muxes	0	0	3650	0.00

3.2 Risultati dei test

Per assicurare l'efficienza e la correttezza del componente sintetizzato, dopo averlo sottoposto a una prima valutazione con il test bench di esempio, abbiamo elaborato ulteriori casi di test. Questi test includono scenari che mettono alla prova la simulazione negli edge case, mirando a coprire tutti i possibili cammini che la macchina potrebbe seguire durante l'esecuzione.

3.2.1 `Data in credibility cell` test bench

Il test verifica che, in presenza di una stringa il cui primo valore è zero e con successivi valori diversi da zero nel byte di credibilità, il processo di scrittura nel byte di credibilità sia eseguito correttamente.

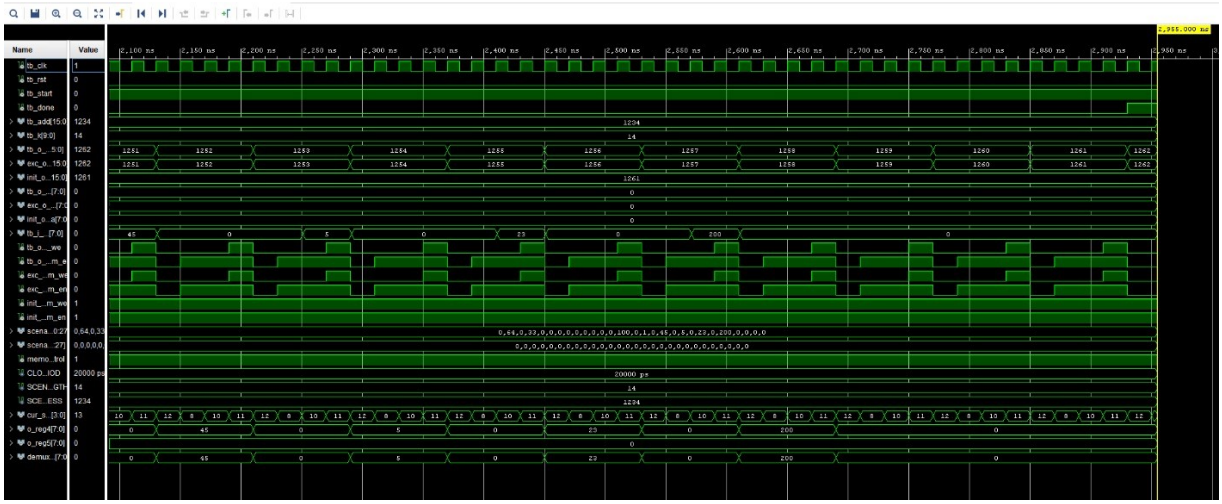

Figura 7: Data in credibility cell test bench

3.2.2 `Mid reset` test bench

Test per verificare la corretta funzionalità del sistema quando un segnale di reset viene inviato tra la prima e la seconda sequenza di parole.

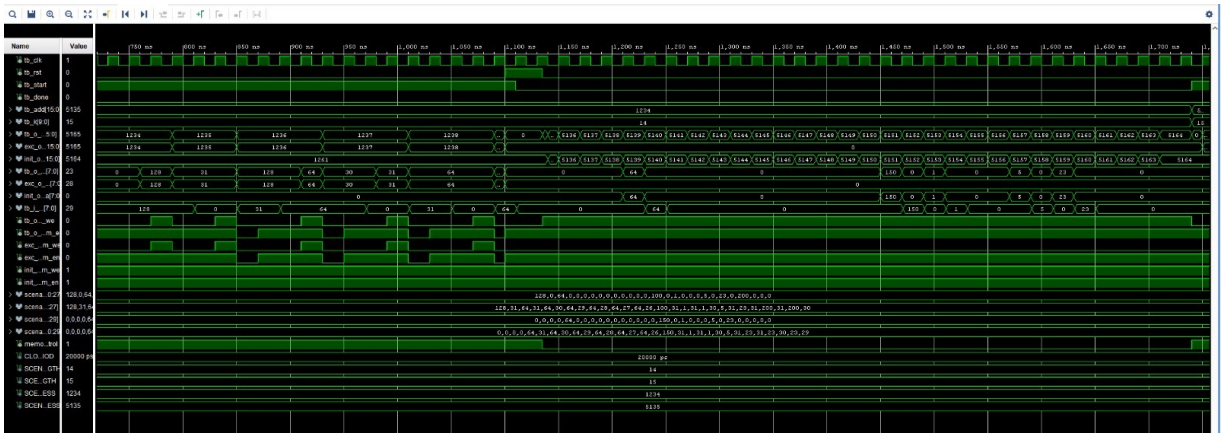

Figura 8: Mid reset test bench (istante in cui viene mandato il segnale di reset)

3.2.3 `K max value` test bench

Test per verificare la corretta funzionalità del sistema quando il segnale `i_k`, rappresentato come un valore a 10 bit, raggiunge il suo valore massimo di 1023. Questo scenario è critico poiché comporta la gestione di 2046 celle, richiedendo un contatore interno esteso a 11 bit per adeguarsi al numero di byte.

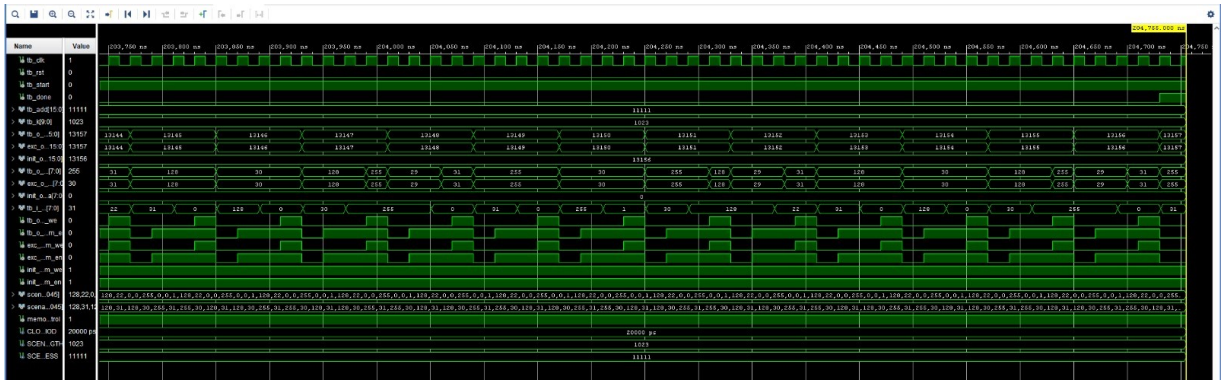

Figura 9: K max value test bench

3.2.4 `Edge` case

Sono stati testati i seguenti casi limite:

- `K zero` **test bench**: verifica la corretta gestione del sistema quando riceve in input una stringa di lunghezza zero parole.
- `All zero` **test bench**: verifica la corretta gestione del sistema nel caso in cui riceva in input una stringa composta interamente da zeri.
- `Multiple start` **test bench**: verifica la corretta gestione del sistema nel caso in cui venga inviato più di un segnale di start.
- `Credibility to zero` **test bench**: verifica il corretto funzionamento del sistema quando una sequenza inizia con una W (diversa da zero) seguita da una serie di zeri sufficienti a ridurre a zero la credibilità, assicurando che non vi siano ulteriori decrementi.
- `Data in credibility cell 2` **test bench**: verifica il corretto funzionamento del sistema nel caso in cui il byte di credibilità contenga un valore diverso da zero, con una parola iniziale anch'essa diversa da zero.
- `Last address` **test bench**: verifica il corretto funzionamento del sistema quando l'ultimo valore viene scritto all'ultimo indirizzo disponibile nella memoria.
- `Reset during done` **test bench**: verifica il corretto funzionamento del sistema quando viene ricevuto un segnale di reset mentre il segnale `o_done` è alto.

In tutti questi casi di test il progetto ha ottenuto esiti positivi.

3.3 Schematic

Per questione di impaginazione, lo schematic si trova in fondo al documento.

4 Conclusioni

La specifica è stata rispettata mediante la creazione di un'architettura performante ed efficiente, come dimostrato dai test e dalle simulazioni effettuate. L'architettura si presenta infatti modulare e migliorata grazie all'interazione tra il datapath e la macchina a stati finiti astratta che ne regola il funzionamento. Inoltre, la sintesi del progetto ha permesso un'analisi dettagliata delle prestazioni e dell'efficacia delle soluzioni adottate.

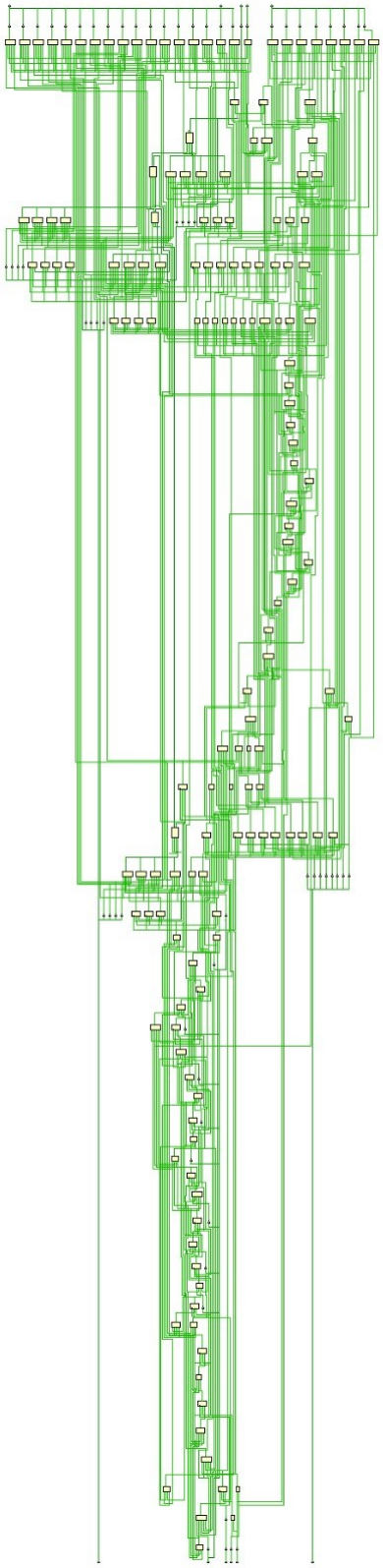

Figura 10: Schematic