# **INTRODUZIONE**

**Il progetto è stato realizzato congiuntamente da ELISABETTA LOLLINO e FRANCESCO MARCHI.**

# **Scopo del progetto**

Il progetto consiste nella lettura dalla RAM di un messaggio costituito da k parole, ognuna delle quali può assumere un valore compreso tra 0 e 255. Ogni parola viene separata da un bit con valore pari a 0 che viene poi sostituito da un *valore di credibilità*, settato inizialmente a 31.  
In presenza di parole pari a 0, il valore di credibilità viene decrementato di uno ad ogni zero letto e il valore 0 in questione viene sostituito dall’ultimo valore letto diverso da zero.  
 Se invece viene letta una parola diversa da 0 il valore di credibilità ritorna nuovamente al suo valore di partenza, e la parola viene salvata come *ultimo valore valido*.

Esempio:  
Messaggio *in lettura*: **128** 0 **64** 0 **0** 0 **0** 0 **0** 0 **0** 0 **0** 0 **100** 0 **1** 0 **0** 0 **5** 0 **23** 0 **200** 0 **0** 0

Messaggio *dopo l’elaborazione*: **128** 31 **64** 31 **64** 30 **64** 29 **64** 28 **64** 27 **64** 26 **100** 31 **1** 31 **1** 30 **5** 31 **23** 31 **200** 31 **200** 30

Il modulo ha tre ingressi principali:

* **START (1 bit)**: un segnale che indica l’inizio dell’elaborazione
* **ADD (16 bit):** contiene indirizzo iniziale dell’elaborazione
* **K (10bit)**: contiene il numero di parole da elaborare

Per le uscite quella principale è:

* **il DONE (1bit)**: un segnale che indica la fine dell’elaborazione e che viene impostato dal circuito programmato

Inoltre, c’è un unico segnale di CLK, utilizzato per sincronizzare tutti i segnali, ad eccezione del segnale RESET, che è *asincrono.*

Scendendo più nel dettaglio: il modulo attende che **il segnale di START diventi 1** prima di iniziare a processare i dati nella RAM. Una volta che il segnale di START diventa alto, **rimane alto per tutta la durata dell'elaborazione**, compresa la scrittura dei dati in memoria. Il segnale di **DONE si alzerà al termine dell’elaborazione** e rimane alto fino a quando il segnale di START non verrà riportato a 0, inoltre per poter iniziare una nuova elaborazione è necessario che prima il segnale di DONE venga abbassato. Si dà per assunto che la prima elaborazione **sarà preceduta da un reset**.

Quando il segnale di START viene impostato a 1, e per tutto il tempo in cui rimane 1, sugli ingressi ADD e K vengono impostati il primo indirizzo e la dimensione della sequenza da elaborare, e rimangono tali finché START rimane alto. Nel caso in cui **il primo dato della sequenza sia pari a zero**, il suo valore rimane tale e il valore di credibilità deve essere impostato a zero. Lo stesso procedimento si applica **fino al raggiungimento del primo dato della sequenza con valore diverso da zero**.

## **Approccio al progetto**

Nel corso della pianificazione di questo progetto, abbiamo deliberatamente optato per l'impiego di componenti preesistenti, non limitandoci solo a quelli appresi durante le lezioni ma anche oltre, al fine di avvicinarci il più possibile a un'implementazione circuitale realistica.

**ARCHITETTURA**

## **Interfaccia del componente**

Il componente presenta la seguente interfaccia

|  |  |
| --- | --- |
| * **i\_clk**: segnale di CLOCK; * **i\_rst**: segnale di RESET asincrono; * **i\_start**: segnale di START che notifica al componente quando iniziare ad elaborare la stringa in ingresso; * **i\_k**: segnale che rappresenta la lunghezza della sequenza in numero di bit; * **i\_add**: segnale che rappresenta l’indirizzo di memoria dal quale parte la sequenza da elaborare; * **o\_done**: segnale generato dal componente che comunica la fine dell’elaborazione; * **o\_mem\_addr**: segnale che specifica su quale indirizzo di memoria eseguire operazioni di lettura e scrittura; * **i\_mem\_data**: segnale ricevuto dalla memoria che contiene la parola della sequenza nell’indirizzo specificato da o\_mem\_addr; * **o\_mem\_data**: segnale generato dal componente, è il dato che verrà scritto in memoria nell’indirizzo specificato da o\_mem\_addr; * **o\_mem\_en**: segnale di ENABLE che abilita le operazioni di lettura e scrittura in memoria; * **o\_mem\_we**: segnale che abilita le operazioni di scrittura se e solo se anche il segnale o\_mem\_en è alto. |  |

## **Descrizione dei componenti**

### **Registri**

I registri sono formati da una serie di **Flip Flop tipo D** con ingressi e uscite parallele (quindi tecnicamente dei registri di tipo **PIPO**)**.** Questa scelta è stata motivata dalla necessità di gestire in modo efficiente sia i valori di input che quelli di output, lavorando in parallelo.

#### ***Register Address (RA) – 16 bit***

Il registro in questione ha lo scopo di memorizzare l'indirizzo iniziale della prima parola e successivamente il valore elaborato dal sommatore. Il RESET imposta il valore salvato a 0 (cioè 16 zeri in fila).

#### ***Register Credibility (RC) – 8 bit***

Registro dedicato al valore di credibilità, è destinato a memorizzare il valore che varia in base alle operazioni del sottrattore.  
Esso ha come input il valore calcolato dal sottrattore (che decrementa sempre il valore salvato di 1). Il RESET imposta il valore salvato a 31, ciò è fondamentale per il comportamento del componente.

#### **Register Data (RD) – 8 bit**

Questo registro salva gli ingressi validi (cioè diversi da zero), al fine di utilizzarli per sostituire gli zeri. Per poter eseguire questa operazione, questo registro ha come ingressi:

* **i\_mem\_data,** il valore da memorizzare;
* **RD\_RST**, segnale di RESET generato dal FSM. Imposta il valore salvato a 0 (cioè 8 zeri in fila)
* **RD\_EN**, segnale generato dal FSM. Quando èalto, indica che il **registro deve memorizzare il valore di i\_mem\_data**; se è basso allora **non lo memorizza.**

#### **Register K (RK) – 10 bit**

Un registro utilizzato per salvare le operazioni effettuate sul segnale **i\_k**. Ha come input:

* **RK\_RST**, segnale di RESET generato dall’FSM. Quando è alto imposta il valore salvato a 0.
* **SUM\_RK**, segnale di uscita del sommatore SUM\_RK che appunto si occupa di eseguire le operazioni su **i\_k.**

### **Sommatori**

Sarebbe più corretto definirli *incrementatori* visto che tutto ciò che fanno è solo incrementare di 1, cosa che ci ha permesso di semplificare la loro struttura.  
Sappiamo che un sommatore *ripple carry* è formato da una serie di *Full Adder*, e che l’i-esimo FA produce come risultati:

* Si = ai ⊕ bi ⊕ ci
* Ci = ai bi + ai ci +bi ci

Notiamo però che il bit rappresentato da bi **sarà sempre pari a zero**, di conseguenza possiamo riscrivere le formule come:

* Si = ai ⊕ ci
* Ci = ai ci

Cioè, **abbiamo ottenuto un Half Adder**, possiamo dunque utilizzare un ripple carry formato solo da Half Adder (fa eccezione il primo elemento del ripple carry che in teoria può avere b0 = 1, in ogni caso però la struttura non cambia visto che **il primo elemento di un ripple carry è sempre un Half Adder**).

#### **Sommatore del K (SUM\_K)**

Questo sommatore ha come obiettivo quello di incrementare il valore del registro RK, serve a contare il numero di parole che sono state elaborate.  
Ha come input:

* **RK**, ovvero valore da incrementare proveniente dal register\_k;
* **INC\_EN**, segnale di ENABLE generato dalla FSM. Abilita l’incremento.

L’uscita sovrascrive il valore salvato nel registro.

#### **Sommatore del RA (SUM\_RA)**

Il sommatore è stato progettato per aumentare il valore del parametro in ingresso ***i\_add*,** che rappresenta l'indirizzo della prima parola da leggere nella RAM, consentendo così la lettura (e scrittura) delle parole successive.  
Ha come input:

* **RA**, ovvero valore da incrementare proveniente dal register\_ra;
* **ADD\_EN**, segnale di ENABLE generato dalla FSM. Abilita l’incremento.

L’uscita sovrascrive il valore salvato nel registro.

### **Sottrattore del RC (SUB\_RC)**

Vale lo stesso ragionamento fatto i sommatori, dunque la struttura del sottrattore è formata interamente da *Half Subtractor*.  
È stato concepito per ridurre il valore di credibilità ogni volta che la parola letta è uguale a zero.

Ha come input:

* **RC**, ovvero valore da incrementare proveniente dal register\_rc;
* **SUB\_EN**, segnale di ENABLE generato dalla FSM. Abilita il decremento.

L’uscita sovrascrive il valore salvato nel registro.

### **XNOR del K**

Viene usato per calcolare il valore di **o\_done**, confrontando il valore di **i\_k** con il valore generato dal SUM\_RK.  
Quando l’output è:

* Uguale a 1 vuol dire che **i\_k = RK;**
* Uguale a 0 vuol dire che **i\_k != RK.**

### **OR E**

Calcola il segnale E, si tratta banalmente di una NOR fra tutti i bit del valore letto in ingresso. In particolare:

* Se **E=0**, significa che il valore letto è **diverso da zero;**
* Se **E=1**, significa che il valore letto è **pari a zero**.

### **OR ZERO\_CHECK**

Infine, abbiamo implementato una porta logica OR con lo scopo di segnalare all'FSM quando RC avesse raggiunto il valore di zero e non dovesse più essere decrementato. Quest'azione **impedisce a SUB\_RC di assumere il valore pari a 1**. È una NOR fra tutti i bit di RC:

* Quando **RC è uguale a "00000000"**, **ZERO\_CHECK** assume il **valore 1**, indicando che RC ha raggiunto il valore di zero.  
   In questo caso, il segnale di ENABLE del sottrattore di RC, chiamato SUB\_EN, viene impostato a zero.
* Quando **RC è diverso da "00000000"**, **ZERO\_CHECK** assume il **valore 0**.

### **Multiplexer Register Address (MUX\_RA)**

Seleziona il segnale in ingresso nel register\_address tra i\_add e il risultato di SUM\_RA. Il segnale i\_add deve essere salvato prima di poter fare le necessarie elaborazioni. È controllato dal segnale SEL\_ADD generato dalla FSM: 0 seleziona i\_add, 1 seleziona SUM\_REG\_ADD.

### **Multiplexer Register K (MUX\_RK)**

Durante la fase di testing abbiamo riscontato dei problemi riguardanti l’assegnazione al valore **o\_done**. Poiché prima dell’assegnazione del valore di **i\_k**, questo assumeva un valore non specificato, non eravamo in grado di calcolare XNOR tra esso e il sommatore del registro RK.

Poiché il risultato di questa operazione determinava il valore di o\_done, ottenevamo sempre un risultato indefinito.  
Per risolvere questo problema abbiamo creato un multiplexer con due ingressi:

* Una **costante zero** da un singolo bit;
* Il segnale di DONE vero e proprio.

Per decidere quale valore assegnare a o\_done, l’uscita del multiplexer, abbiamo utilizzato un selettore DONE\_MUX\_SEL, generato dal FSM principale:

* Prima dell'inizio dell'elaborazione, **DONE\_MUX\_SEL è impostato a 0**, in modo che il multiplexer prenda la costante zero
* Dopo l'inizio dell'elaborazione, **DONE\_MUX\_SEL viene impostato su 1**, consentendo al multiplexer di utilizzare il valore calcolato dall’XNOR come input, che sarà sempre zero fino a quando non verrà fatto l’ultima elaborazione, compresa la scrittura in memoria.

### **Multiplexer o\_mem\_data (MUX\_O)**

Per gestire l’uscita dei dati avevamo a disposizione solo un'uscita, **o\_mem\_data.** Avendo due possibili segnali di uscitaabbiamo introdotto un multiplexer.  
Questo componente alterna l'uscita tra due registri che conservano le informazioni sui dati: **RC** e **RD**.  
Il multiplexer viene controllato da un segnale generato dalla FSM, il segnale **SEL\_OUT:**

* **SEL\_OUT uguale a 0**, il multiplexer mette in uscita il valore contenuto nel registro **RC**.
* **SEL\_OUT uguale a 1**, il multiplexer mette in uscita il valore contenuto nel registro **RD**.

## **Descrizione della FSM (MOORE)**

La FSM può essere considerata il fulcro centrale del circuito, poiché gestisce completamente la logica attraverso 14 stati navigati tramite quattro ingressi fondamentali, che sono:

1. **START**: è il segnale fornito esternamente dall'utilizzatore che indica l'avvio dell'elaborazione dopo le operazioni preliminari;
2. **E**: è il segnale calcolato internamente dal componente, che effettua una computazione logica preliminare sul valore letto dalla memoria. Risulta alto se il valore letto corrisponde a 0, mentre è basso se viene un valore diverso da zero.
3. **DONE**: è il segnale che segnala la conclusione dell'elaborazione. Successivamente, nella descrizione degli stati, sarà chiaro il motivo per cui questo segnale è considerato di ingresso;
4. **CHECK\_ZERO**: calcolato internamente, questo segnale agisce come una protezione nel caso in cui il valore memorizzato nel registro di credibilità raggiunga lo zero, garantendo l'integrità del processo di gestione della credibilità.

Ogni stato è invece dotato delle seguenti dieci uscite:

1. **ADD\_EN**: segnale di ENABLE per il sommatore SUM\_RA;
2. **RD\_EN**: segnale di ENABLE per il registro REGISTER\_DATA;
3. **SEL\_OUT**: segnale di selezione per il MULTIPLEXER\_O\_MEM\_DATA:
   * 1 manda in uscita il valore salvato nel REGISTER\_DATA
   * 0 manda in uscita il valore salvato nel REGISTER\_CREDIBILITY
4. **RC\_RST**: segnale di reset del REGISTER\_CREDIBILITY;
5. **RD\_RST**: segnale di reset del REGISTER\_DATA;
6. **SUB\_EN**: segnale di ENABLE per il sottrattore SUB\_RC;
7. **O\_MEM\_E**: segnale di ENABLE per le operazioni sulla memoria;
8. **O\_MEM\_WE**: segnale di ENABLE per la scrittura in memoria (la scrittura richiede che O\_MEM\_E e O\_MEM\_WE siano alti allo stesso tempo);
9. **SEL\_ADD**: segnale di selezione per il MULTIPLEXER\_RA:
   * 1 seleziona il valore in uscita al SUM\_RA, cioè il risultato dell’incremento
   * 0 seleziona il valore i\_add letto dal componente alla sua inizializzazione
10. **DONE\_MUX\_SEL**: segnale di selezione per il MUX\_RK, costituendo un'ulteriore misura di sicurezza per garantire che il segnale DONE non si attivi in momenti dell'elaborazione indesiderati:
    * 1 impone che il DONE sia pari a zero
    * 0 impone che il DONE sia il risultato dell’elaborazione della XNOR\_K

### **Descrizione degli stati**

***Stati iniziali:***

* **S0**: Questo è lo stato di reset della macchina. Rimane in questo stato finché non viene rilevato START=1, indicando l'inizio dell'elaborazione della stringa in ingresso. Durante questo periodo, tutti i segnali di ENABLE sono bassi, mentre i segnali di RESET sono alti. Poiché il reset del componente tra un'elaborazione e l'altra non è garantito, abbiamo implementato una gestione "esplicita" dei segnali di reset che non interferisca con il reset globale del componente. Questo approccio si basa sul fatto che la macchina ritorna sempre a S0 alla fine dell'elaborazione di una stringa. Pertanto, impostando i segnali di RESET alti in S0, possiamo reimpostare i componenti rilevanti senza dover fare affidamento su un reset globale. Allo stesso modo, poiché un reset globale riporta sempre la FSM a S0, i reset espliciti non interferiscono con esso. In questo stato, i segnali di selezione non sono significativi tranne per DONE\_MUX\_SEL, che è basso. Questo è essenzialmente una precauzione di sicurezza per evitare che DONE si alzi prematuramente.
* **S1**: Questo stato rappresenta una fase "intermedia" in cui l'elaborazione non è ancora iniziata ma è necessario per due ragioni:
  + Potremmo trovarci in una situazione particolare in cui la lunghezza della stringa è pari a zero. In tal caso, questo stato ci consente di passare direttamente allo stato di fine elaborazione, SF.
  + Prima di poter eseguire le operazioni necessarie sul REGISTER\_ADDRESS, è essenziale avere la possibilità di salvare l'indirizzo di memoria del primo valore della stringa. Tuttavia, questo segnale, i\_add, viene fornito solo quando i\_start=1. Pertanto, questo stato ci consente di salvare questo valore senza problemi, garantendo una transizione fluida verso lo stato successivo.

***S2-S3-S5 loop:* questa parte della macchina si occupa del caso particolare in cui la stringa da elaborare inizia con almeno uno 0**

* **S2**: Questo è lo stato di lettura. Qui il segnale O\_MEM\_E si alza poiché la lettura richiede due colpi di clock, e nello stato successivo il valore di memoria sarà disponibile. Poiché dopo la lettura potrebbe essere completata l'elaborazione, da S2 possiamo passare direttamente a SF nel caso in cui DONE si sia già alto.
* **S3**: Questo è lo stato di branching. In questo stato, il segnale ADD\_ENABLE si attiva per calcolare l'indirizzo di memoria della cella successiva rispetto a quella da cui è stato letto il valore. A questo punto, ci sono due possibili scenari:
  + Se il valore letto è diverso da 0, l'elaborazione procede normalmente e passiamo a S4
  + Se il valore letto è uguale a 0, ci troviamo nel caso particolare in cui il primo valore letto è 0. Questo potrebbe indicare che la stringa inizia con una serie di 0 o è interamente costituita da 0. In questo caso, entriamo in un loop tra gli stati S2-S3-S5.
* **S5**: Questo è lo stato di scrittura. Ci troviamo in una situazione particolare in cui la stringa inizia con 0, il che implica che il valore di credibilità rimarrà 0 fino a quando non leggeremo un valore valido. Poiché utilizzare il REGISTER\_CREDIBILITY sarebbe stato complicato, abbiamo optato per l'utilizzo del valore salvato in REGISTER\_DATA, che dopo il RESET sarà sicuramente impostato a 0. Per questo motivo, il segnale SEL\_OUT è impostato a 1. È importante notare che i segnali O\_MEM\_EN e O\_MEM\_WE sono alti, consentendo la scrittura. Inoltre, il segnale ADD\_ENABLE alto garantisce che nello stato di lettura successivo avremo a disposizione l'indirizzo di memoria della cella contigua a quella dove è avvenuta la scrittura.
* **S4**: Questo è lo stato di scrittura. Simile a S5, la differenza sta nel fatto che dobbiamo scrivere il valore 31 in memoria, per cui utilizziamo il REGISTER\_CREDIBILITY, quindi il segnale SEL\_OUT è basso. Inoltre, notiamo che il segnale RD\_ENABLE è alto, indicando che stiamo salvando in memoria il valore letto, che è il primo valore valido letto. Una volta raggiunto lo stato S4, è impossibile tornare agli stati precedenti a meno che l'elaborazione non sia completata. Questo perché il loop descritto sopra gestisce solo il caso particolare in cui leggiamo uno o più 0 all'inizio della stringa.

***Gestione del caso generale:* parte della macchina che gestisce l’elaborazione generale della stringa**

* **S6**: Questo stato rappresenta una fase di lettura simile a S2, con l'eccezione che il segnale SUB\_ENABLE viene attivato. Questo indica che, per impostazione predefinita, consideriamo il valore letto come uno 0 fino a prova contraria.
* **S7**: Questo stato è una fase di branching identica a S3, ad eccezione del fatto che il segnale ADD\_ENABLE è basso. Questo perché se il valore letto è zero, non dobbiamo solo scrivere la credibilità, ma anche l'ultimo valore valido.

***Valore letto pari a 0:***

* **S8**: Questo stato rappresenta la fase di scrittura dell'ultimo valore valido. Qui sostituiamo lo 0 letto con il valore salvato in precedenza (SEL\_OUT=1), e solo in questo stato ADD\_ENABLE può essere attivato.
* **S9**: Questo stato rappresenta la fase di scrittura della credibilità. In questo stato, il segnale SEL\_OUT è impostato a 0, indicando che stiamo scrivendo il valore di credibilità nella memoria.

***CHECK\_ZERO=1:* se normalmente ritorneremmo nello stato S6, CHECK\_ZERO alto significhi che dopo svariati loop il valore di credibilità salvato in REGISTER\_CREDIBILITY ha raggiunto lo 0. Per evitare errori di overflow, la macchina va nello stato S12**

* **S12**: stato safeguard. Identico allo stato S6, con la differenza fondamentale che in questo caso il segnale SUB\_ENABLE è basso. Possiamo quindi considerare che S12 sostituisce S6 nel flusso dell'elaborazione.

***Valore letto pari a 1:***

* **S10**: Questo stato è dedicato al salvataggio del valore letto come valido (RD\_ENABLE=1). Inoltre, viene attivato il segnale RC\_RESET per riportare il suo valore a 31, poiché abbiamo detto che fino a prova contraria ci aspettiamo di leggere uno 0.
* **S11**: Questo stato è responsabile della scrittura della credibilità nella memoria.

***Stato finale:***

* **SF**: Questo stato rappresenta la fine dell'elaborazione, raggiunto quando il segnale DONE si alza. La macchina rimane in questo stato fino a quando il segnale START non viene abbassato dall'utilizzatore del componente. Il segnale DONE è un ingresso della macchina. Essendo uno stato di non-elaborazione, tutti i segnali ENABLE sono bassi, i selettori sono indifferenti e i segnali di RESET sono bassi.

## **Descrizione della FSM\_K (MOORE)**

Lo scopo della macchina è semplicemente quello di attivare un segnale di ENABLE ogni due segnali ADD\_ENABLE letti. L’attivazione di ENABLE, a sua volta, attiva il sommatore SUM\_RK che incrementa di 1 il valore salvato nel REGISTER\_K a partire da 0. Il valore del registro è comparato con i\_k attraverso una XNOR (sappiamo bene infatti che dati due segnali logici A e B, (A⊕B)’=0 solo se i due segnali sono equivalenti) per generare il segnale di DONE. La scelta di utilizzare ADD\_ENABLE deriva dal fatto che ad ogni attivazione viene indicato uno spostamento di una posizione nella memoria. Pertanto, se la lunghezza della stringa è K, l'elaborazione sarà completa dopo aver letto 2K posizioni.

Anche se contare utilizzando una macchina a stati è un processo semplice, la complessità della FSM principale ha reso necessario l'utilizzo di una seconda macchina a stati, denominata FSM\_K, con il design descritto di seguito.

**Immagine che contiene schizzo, disegno, Line art, clipart

Descrizione generata automaticamente**L’unico ingresso della macchina è il segnale di ADD\_ENABLE, mentre il segnale di uscita è un INC\_ENABLE che attiva il SUM\_RK. L’uscita è alta ogni due 1 in ingresso, che possono anche essere spezzati da uno o più 0

# **RISULTATI SPERIMENTALI**

In seguito alla realizzazione della sintesi, la TCL console ha prodotto i seguenti report:

## **Utilization report:**

Immagine che contiene testo, schermata, numero, Carattere

Descrizione generata automaticamenteCome possiamo leggere, non è registrato l’utilizzo di alcun LATCH, e visto che non abbiamo usato LATCH nel nostro design, significa che non sono presenti INFERRING LATCHES.  
Gli inferring latches sono dei latch involontari non introdotti dal progettista, ma creati da Vivado per cercare di mantenere stati del componente che normalmente muterebbero in modo autonomo.

## **Timing Report:**

“Slack (MET) : 14.264ns (required time - arrival time)”

Il report indica che i constraint sotto cui abbiamo lavorato sono stati rispettati con un ampio margine. In effetti, si potrebbe affermare che il componente potrebbe operare senza problemi a una frequenza doppia rispetto a quella attuale.

## **Risoluzione dei problemi**

Uno dei primi problemi che abbiamo incontrato è stato nella gestione della scrittura e la lettura.  
Inizialmente, avevamo erroneamente supposto che la lettura richiedesse soltanto 1 stato, mentre la scrittura ne richiedesse 2 (in altre parole l’opposto del funzionamento corretto). Questo errore ci ha costretti a riprogettare completamente la FSM da zero

Nonostante la riprogettazione, abbiamo incontrato ulteriori difficoltà, in particolare abbiamo avuto numerose difficoltà a sincronizzare il calcolo degli indirizzi con i vari stati della macchina. Sebbene avessimo progettato i sommatori e i sottrattori come una versione semplificata del Carry Look Ahead, che di per sé è un componente molto veloce, eravamo convinti che il risultato dei calcoli di questi componenti fosse disponibile solo al clock successivo. Tuttavia, abbiamo scoperto che il calcolo era pressoché istantaneo.  
Di conseguenza, se inizialmente avevamo pensato che ci volessero ben 3 stati della macchina per leggere (uno per il calcolo dell’indirizzo, uno per mandare la richiesta di lettura e uno per ricevere effettivamente il dato) ce ne volevano soltanto 2.

Un altro problema che abbiamo affrontato riguardava il calcolo del segnale di DONE.

Inizialmente, avevamo impostato le transizioni per lo stato SF *negli stati in cui avveniva una scrittura*, poiché eravamo convinti che se avessimo scritto l'ultimo valore di credibilità, avremmo concluso l'elaborazione. Successivamente, abbiamo scoperto che questo non avveniva immediatamente: il segnale di DONE si alzava dopo due stati. Di conseguenza, la macchina poteva rimanere bloccata in uno stato qualunque o tornare allo stato di scrittura senza raggiungere lo stato SF, il che non garantiva il funzionamento del componente in caso di più di una singola elaborazione di fila a causa di gravi problemi di sincronizzazione con il nuovo segnale di start.

Abbiamo risolto facendo due cose:

* Prima di tutto abbiamo collegato direttamente il risultato del SUM\_RK alla porta XNOR\_K. In precedenza, era collegata con il REGISTER\_K, velocizzando la disponibilità del segnale di DONE di 1 stato;
* Abbiamo aggiunto delle transizioni per lo stato SF *subito dopo agli stati di scrittura*, e rimosso quelle vecchie.

## **Verifica delle varie casistiche**

Per verificare che il modulo rispondesse come ci aspettavamo in tutte le possibili casistiche, abbiamo iniziato creando i TEST CASE presenti nelle specifiche:

* Il primo TESTBENCH era una semplice sequenza senza nessun caso limite.

**signal** scenario\_input : scenario\_type := (**51**,**0**,**0**,**0**,**57**,**0**,**24**,**0**,**0**,**0**,**0**,**0**,**126**,**0**,**0**,**0**,**192**,**0**,**0**,**0**);

**signal** scenario\_full : scenario\_type := (**51**,**31**,**51**,**30**,**57**,**31**,**24**,**31**,**24**,**30**,**24**,**29**,**126**,**31**,**126**,**30**,**192**,**31**,**192**,**30**);

* Il secondo TESTBENCH includeva il caso in cui il valore di credibilità viene decrementato fino a zero e rimane tale finché continua a leggere zeri.

**signal** scenario\_input : scenario\_type:=(**51**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**);

**signal** scenario\_full: scenario\_type:=(**51**,**31**,**51**,**30**,**51**,**29**,**51**,**28**,**51**,**27**,**51**,**26**,**51**,**25**,**51**,**24**,**51**,**23**,**51**,**22**,**51**,**21**,**51**,**20**,**51**,**19**,**51**,**18**,**51**,**17**,**51**,**16**,**51**,**15**,**51**,**14**, **51**,**13**,**51**,**12**,**51**,**11**,**51**,**10**,**51**,**9**,**51**,**8**,**51**,**7**,**51**,**6**,**51**,**5**,**51**,**4**,**51**,**3**,**51**,**2**,**51**,**1**,**51**,**0**,**51**,**0**,**51**,**0**,**51**,**0**);

* Il terzo TESTBENCH controllava il caso in cui la stringa leggeva prima uno zero e poi altri numeri diversi da zero.

**signal** scenario\_input : scenario\_type := (**0**,**0**,**255**,**0**,**0**,**0**,**0**,**0**,**137**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**111**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**181**,**0**,**0**,**0**, **0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**246**,**0**,**124**,**0**,**0**,**0**,**93**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**,**0**);

**signal** scenario\_full : scenario\_type := (**0**,**0**,**255**,**31**,**255**,**30**,**255**,**29**,**137**,**31**,**137**,**30**,**137**,**29**,**137**,**28**,**137**,**27**,**137**,**26**,**137**,**25**,**137**,**24**,**137**, **23**,**111**,**31**,**111**,**30**,**111**,**29**,**111**,**28**,**181**,**31**,**181**,**30**,**181**,**29**,**181**,**28**,**181**,**27**,**181**,**26**,**246**,**31**,**124**,**31**,**124**,**30**,**93**,**31**,**93**,**30**,**93**,**29**,**93**,**28**,**93**,**27**,**93**, **26**,**93**,**25**);

Successivamente, abbiamo sviluppato dei TESTBENCH aggiuntivi che comprendevano casi non presenti nelle specifiche:

* Un caso in cui ci sono più sequenze da elaborare.
* Un caso in cui si ha i\_k=0.
* Un caso in cui si verifica un segnale di RESET durante l'elaborazione, sia per una sola sequenza che per due.

In tutti questi casi, siamo riusciti a far funzionare il nostro progetto, e grazie a essi, ci siamo resi conto degli errori discussi in precedenza, contribuendo a rendere il nostro lavoro il più completo possibile.

# **CONCLUSIONI**

La progettazione ha richiesto la combinazione di diversi componenti, come registri, sommatori, sottrattori e multiplexer, il tutto orchestrato da una macchina a stati finiti (FSM) per gestire logicamente le fasi di lettura, elaborazione e scrittura dei dati. Tramite una rigorosa fase di testing, prima dei componenti singoli attraverso strumenti come GHDL e GTKWAVE, e solo successivamente del componente nella sua interezza con gli strumenti forniti da Vivado e la scrittura di testbench, abbiamo raggiunto un design finale che secondo noi rispetta le specifiche richieste; i report generati da Vivado ci hanno dato un’ulteriore conferma di ciò.

In definitiva il progetto ci ha permesso di acquisire nozioni e conoscenze importanti sul funzionamento dei sistemi digitali, sulla progettazione basata su specifiche e una visione sugli strumenti utilizzati da esperti e studiosi del settore.