Prova Finale
Progetto di Reti Logiche

Professor Gianluca Palermo – anno 2021/2022

Enrico Alessandro Maria Vento – 10680917, 935358

Federico Valentino – 10679810, 934581

Indice

1

Introduzione *3*
1.1 Specifiche Generali *3*
1.2 Dati e Memoria *4*
1.3 Interfaccia del Componente *5*

2

Progettazione e Design *6*
2.1 Descrizione Generale e Scelte Progettuali *6*
2.2 Stati *7*
2.2.1 WAITFORSTART *7*
2.2.2 RESTART *7*
2.2.3 SETREAD *8*
2.2.4 READTOTALWORDS *8*
2.2.5 READWORD *8*
2.2.6 FIRSTBIT *8*
2.2.7 ELABORATEBIT *8*
2.2.8 SETBIT *8*
2.2.9 WRITEWORD *9*
2.2.10 DECREMENTTOTAL *9*
2.2.11 SETWRITE *9*
2.2.12 DONE *9*

3

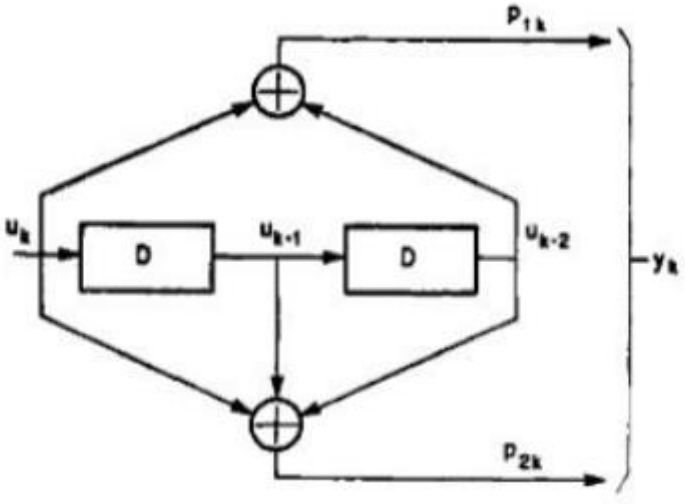
Testing e Ottimizzazioni *12*
3.1 Test Effettuati e Risultati *12*
3.1.1 tre_reset *12*
3.1.2 tre_bis *13*
3.1.3 random testbench *13*
3.2 Ottimizzazione *13*

1. Introduzione

1.1 Specifiche Generali

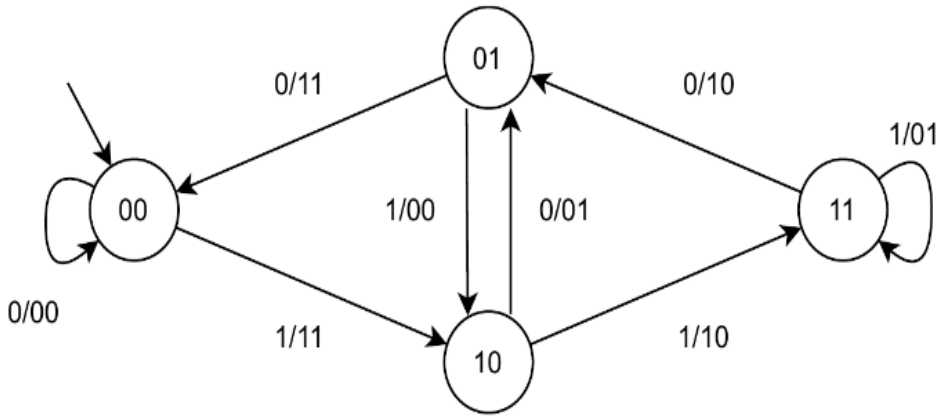
Scopo del progetto è l'implementazione di un modulo *hardware*, descritto in VHDL, che, interfacciandosi con la memoria, esegua le seguenti operazioni.

Data una sequenza di parole (8 *bit* ciascuna) in ingresso, ciascuna di esse viene serializzata, *bit* a *bit*, come specificato dallo schema del convolutore qui rappresentato (dove *Uk* rappresenta un singolo *bit* della parola in ingresso e *Yk* la concatenazione dei risultati delle operazioni dettagliate in figura, *P1k* e *P2k*).



La successiva concatenazione, in ordine, degli *Yk* formerà il flusso d'uscita *Y*, il quale sarà composto anch'esso da parole di 8 *bit* ciascuna (ogni parola corrisponderà alla codifica di metà parola d'ingresso).

Il funzionamento generale del componente, con particolare riferimento ai rapporti ingresso – uscita, è dettagliato dalla seguente macchina a stati



1.2 Dati e Memoria

Il modulo leggerà la sequenza da codificare da una memoria con indirizzamento al *byte*. In particolare:

- All'indirizzo 0 è memorizzata la quantità di parole da codificare
- Agli indirizzi successivi allo 0 sono memorizzate le parole in ingresso
- Le parole codificate vengono scritte a partire dall'indirizzo 1000

1.3 Interfaccia del Componente

Di seguito viene riportata l'interfaccia del componente.

```
entity project_reti_logiche is
      port (
            i_clk     : in std_logic;
            i_rst     : in std_logic;
            i_start   : 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;
```

In particolare:

- i_clk è il segnale di CLOCK in ingresso generato dal *TestBench*;
- 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 *TestBench*;
- i_data è il vettore che arriva dalla memoria in seguito ad una richiesta di lettura;
- o_address è il vettore di uscita che manda l'indirizzo alla memoria;
- o_done è il segnale di uscita che comunica la fine dell'elaborazione e il dato di uscita scritto in memoria;
- o_en è il segnale di ENABLE da inviare alla memoria per poter comunicare;
- o_we è il segnale di WRITE ENABLE da mandare alla memoria per poter scriverci. Per leggere da memoria esso deve essere 0;
- o_data è il vettore di uscita dal componente verso la memoria.

2. Progettazione e Design

2.1 Descrizione Generale e Scelte Progettuali

Il componente è stato progettato secondo il formalismo della macchina a stati: avremo uno stato di *WAITFORSTART*, che funge da *idle*, e uno stato *RESTART*, il nostro *reset*, seguiti da una serie di stati di lettura, elaborazione e scrittura.

Per quanto riguarda la realizzazione pratica, vengono utilizzati tre processi, tutti *clock dependent*.

Il primo codifica il comportamento della macchina: in caso di *reset* alto, lo stato sarà commutato a *WAITFORSTART*, mentre in ogni altra situazione, lo stato viene commutato, durante il *rising_edge* del *clock*, a qualunque sia lo stato successivo, definito nella variabile di stato *nextState*.

Il secondo processo contiene la logica pertinente al cambio stato: è qui che la variabile *nextState* viene manipolata, durante il *falling_edge* del *clock*, per definire lo stato in cui la macchina dovrà trovarsi durante il prossimo ciclo di *clock*.

Il terzo processo, invece, contiene la logica che manipola i vari segnali e vettori ausiliari e che svolge le effettive elaborazioni di cui il componente si fa carico.

Si è deciso di adottare uno *shift register*, chiamato *currentWord*, per scorrere i *bit* delle varie parole in ingresso: le elaborazioni avverranno sempre sul *bit* più significativo di *currentWord*, il quale verrà fatto "shiftare" dopo ogni *bit* elaborato.

Inoltre, per facilitare la scrittura in memoria, si è scelto di modellare la parte di convolutore come un'elaborazione di solamente metà parola in ingresso: infatti, da specifica, da quattro *bit* di ingresso si ottiene una parola in uscita. Un vettore ausiliario, detto *bitCounter*, tiene traccia di quanti *bit* stanno venendo elaborati, permettendoci di capire quando un'intera parola in ingresso è stata elaborata.

La relativa abbondanza di stati è una scelta che verrà approfondita nella sezione 3.2 – Ottimizzazioni.

Segue la descrizione dettagliata del funzionamento dei singoli stati.

2.2 Stati

In totale la macchina si compone di quindici stati: vediamoli nel dettaglio.

2.2.1 *WAITFORSTART*

Si occupa di portare *o_done* a zero se *i_start* è anch'esso a zero.
Conduce in *RESTART* non appena giunge un nuovo segnale di *start*, altrimenti l'esecuzione rimane ferma (è il nostro stato di *idle*).

2.2.2 RESTART

Il nostro stato di *reset*, si occupa di riportare la macchina e i segnali da essa usati allo stato "di fabbrica", approntando il componente ad una nuova elaborazione. Conduce in *SETREAD*.

2.2.3 SETREAD

Prepara la macchina alla lettura, alzando il segnale *o_en*, abbassando quello di *o_we* e settando l'*o_address* all'indirizzo indicato dal vettore *readAddress*, che viene settato da *RESTART* al giusto valore.
In base al valore del segnale booleano *readTotal*, il quale viene usato per capire se la parola da leggere rappresenta uno degli ingressi oppure è il numero totale di parole da leggere (valorizzato come falso in *RESTART*), lo stato conduce in *READTOTALWORDS* (se il booleano ha valore falso) o in *READWORD* (se il booleano ha valore vero).

2.2.4 READTOTALWORDS

Salva nel segnale *totalWords* il numero di parole da leggere attualmente presenti nella memoria, incrementa il *readAddress* e setta il valore di *readTotal* a vero.
Quando in lettura avremo una parola nulla (*i_data* sarà uguale a zero), l'esecuzione dovrà terminare, e la macchina commuta verso lo stato di *DONE*. In caso contrario, si passa allo stato di *SETREAD*.

2.2.5 READWORD

Salva la parola corrente nello *shift register currentWord*, dopodiché incrementa il *readAdress*.
Conduce nello stato chiamato *FIRSTBIT*.

2.2.6 FIRSTBIT

Disabilita la comunicazione con la memoria e inserisce il primo *bit* da analizzare nel segnale ausiliario *currentBit*.
Conduce nello stato indicato dalla variabile di stato *convoNextState*, inizializzato a *ELABORATEBIT_0* da *RESTART*.

2.2.7 ELABORATEBIT

Sono quattro stati di convoluzione, ognuno con opportuna logica. Fanno scorrere lo *shift register* e, in base al valore di *currentBit*, valorizzano Pk1 e Pk2 (come suggerisce la figura della macchina a stati nella sezione 1.1 – Specifiche Generali) e settano il prossimo stato di elaborazione da raggiungere, utilizzando *convoNextState*.
Tutti conducono sempre in *SETBIT*.

2.2.8 SETBIT

La sua azione è determinata dal valore assunto dal vettore ausiliario *outPosition,* dal quale si evince in quale posizione nella parola di uscita (rappresentata dal vettore ausiliario *elaborate)* Pk1 e Pk2 andranno scritti. Inoltre, incrementa *outPosition* stesso, *bitCounter*, e setta il nuovo *currentBit* da analizzare, utilizzando *currentWord*, fatto "shiftare" in precedenza dai vari stati di elaborazione.
Se *outPosition* raggiunge il valore massimo, dunque se metà parola in ingresso è già stata elaborata, conduce in *SETWRITE*, altrimenti conduce nel prossimo stato di elaborazione, indicato da *convoNextState*.

2.2.9 SETWRITE

Prepara la macchina alla scrittura della parola ottenuta e ora salvata in *elaborate*, alzando entrambi i segnali *o_en* e *o_we* e salvando in *o_address* l'indirizzo di scrittura, valorizzato a inizio esecuzione in *writeAddress*.
Conduce in *WRITEWORD*.

2.2.10 WRITEWORD

Scrive la parola in uscita attraverso *o_data*, resetta *outPosition* al valore minimo, per preparare la macchina all'elaborazione di nuovi quattro bit, e incrementa il *writeAddress*.
Se *bitCounter* è al suo valore minimo, allora la prima parola in ingresso è stata elaborata, e la macchina viene condotta nello stato di *DECREMENTTOTAL*; altrimenti si procede alle successive

elaborazioni, commutando verso il corretto stato di convoluzione, indicato da *convoNextState*.

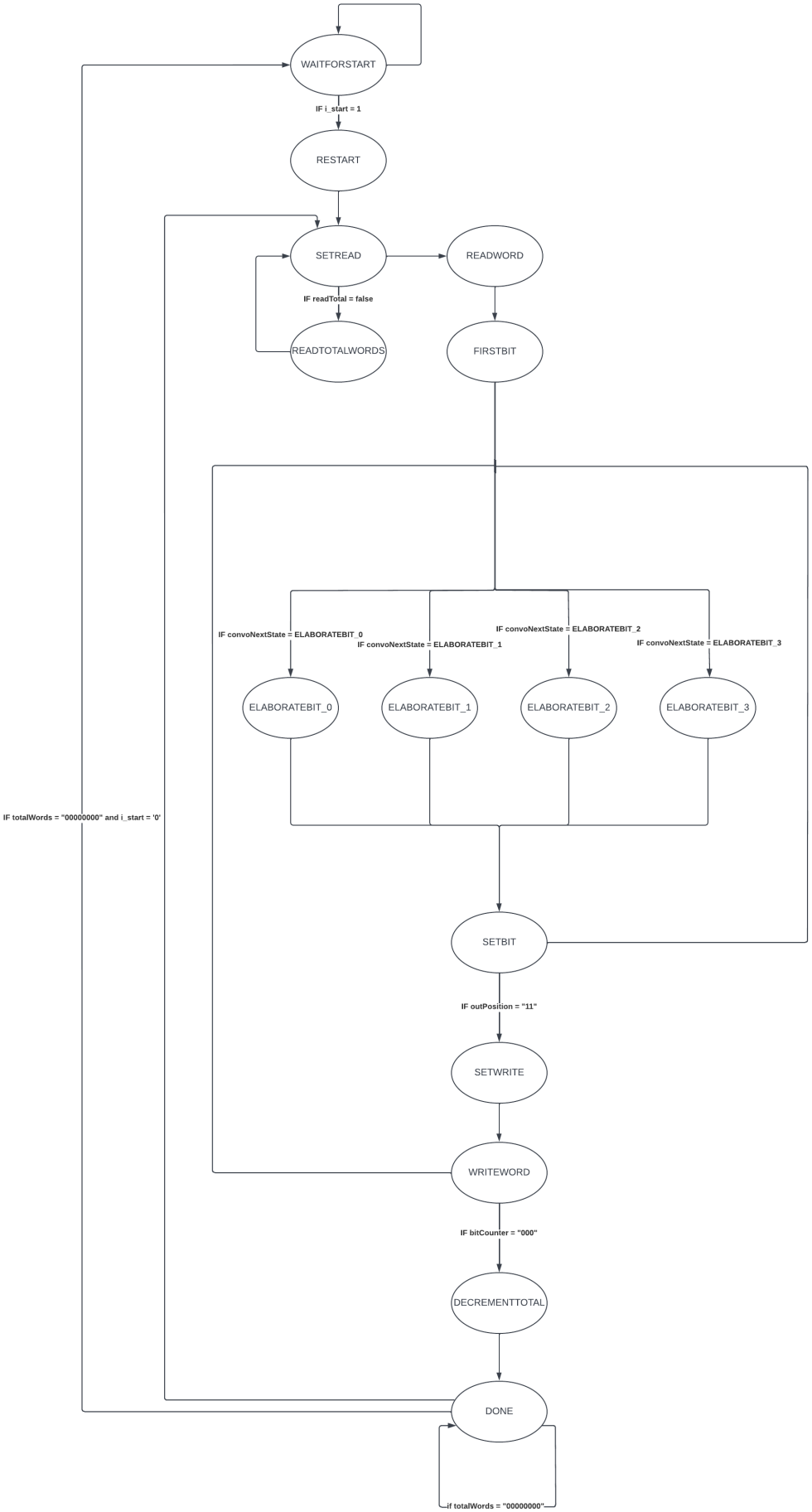
2.2.11 DECREMENTTOTAL

Siccome una parola in ingresso è stata completamente elaborata, decrementa *totalWords* e riporta *bitCounter* al valore minimo, preparando la macchina alla prossima (eventuale) parola in ingresso.
Inoltre, disabilita la comunicazione con la memoria.
Conduce allo stato *DONE*.

2.2.12 ***DONE***

Se *totalWords* ha raggiunto il valore zero, ergo non sono presenti nuove parole in ingresso da elaborare, alza il segnale *o_done*, per segnalare la fine dell'elaborazione. Alternativamente, lo abbassa (o lo lascia invariato).
La transizione da questo stato è la più ramificata. Assumendo che l'elaborazione sia finita (dunque *totalWords* è a valore zero), rimane in *DONE* finché *i_start* rimane alto; appena *i_start* viene abbassato, conduce in *WAITFORSTART*.
Invece, se l'elaborazione non è terminata, conduce in *SETREAD*.

Nella seguente figura, viene rappresentata la macchina con i suoi stati e le relative transizioni.


WAITFORSTART
IF i_start = 1
RESTART
SETREAD
READWORD
IF readTotal = false
READTOTALWORDS
FIRSTBIT
IF convoNextState = ELABORATEBIT_0
IF convoNextState = ELABORATEBIT_1
IF convoNextState = ELABORATEBIT_2
IF convoNextState = ELABORATEBIT_3
ELABORATEBIT_0
ELABORATEBIT_1
ELABORATEBIT_2
ELABORATEBIT_3
IF totalWords = "00000000" and i_start = '0'
SETBIT
IF outPosition = "11"
SETWRITE
WRITEWORD
IF bitCounter = "000"
DECREMENTTOTAL
DONE
if totalWords = "00000000"

3. Testing e Ottimizzazioni

3.1 Test Effettuati e Risultati

Il componente è stato ampiamente testato, utilizzando due metodi differenti.

In primis sono stati svolti test mirati a monitorare il funzionamento della macchina in casi particolari di esecuzione, come la ricezione di numerosi segnali di *reset* durante la normale elaborazione, oppure lo svolgimento di *testbench* con velocità di *clock* molto inferiore ai cento nanosecondi target.

In secondo luogo, è stata analizzata la risposta del componente sotto sforzo, fornendo numerose memorie contenenti grandi numeri di parole da codificare, sia usando casi di test creati *ad hoc*, sia utilizzando una *testbench* "randomica".

Diversi aspetti sono stati scelti come indicatori di performance del componente:

- numero di *flip-flop* utilizzati dalla sintesi
- numero di cicli di *clock* adoperati per terminare l'esecuzione (raffrontati al test standard *tb_example*)
- velocità di *clock* minima (rapportata allo standard di cento nanosecondi)

Segue la trattazione dettagliata di alcuni casi di *test* ritenuti fra i più significativi.

3.1.1 tre_reset

Il *test* in questione permette di verificare il corretto comportamento della macchina quando interrotta da diversi segnali di *reset*: in diversi momenti dell'elaborazione, il segnale di *reset* verrà alzato dalla *testBench*, costringendo la macchina a tornare nello stato *RESTART*.

Il *test* è stato superato con successo in *Behavioural* e *PostSynthesis* rispettivamente in 33600 e 34300 nanosecondi.

3.1.2 tre_bis

Attraverso questo test possiamo monitorare due aspetti della macchina: in primo luogo, possiamo controllare che l'interazione fra i segnali di *start* e *done* sia corretta, in quanto viene chiesto al componente di codificare parole da tre diverse memorie, in cascata.

Inoltre, il *test* viene eseguito ad una velocità di quindici nanosecondi di *clock*. Questo ha comportato grossi problemi alle prime iterazioni del componente da noi progettato, ed ha stimolato un notevole lavoro di ottimizzazione.

Il *test* è stato superato con successo in *Behavioural* e in *PostSynthesis* rispettivamente in 5215 e 5275 nanosecondi.

3.1.3 random testbench

Tra la moltitudine di *test* generati casualmente a cui il componente è stato sottoposto vale la pena di citarne uno in particolare: questo comporta l'utilizzo di 500 memorie da 255 parole ciascuna.

Si tratta dunque di un caso limite, tramite il quale abbiamo verificato la robustezza della macchina.

Si è deciso di omettere la descrizione di ulteriori casi di *test* in quanto ricadono tutti in casi già verificati dai tre test sopracitati (in modo molto più semplice e meno "stressante" per la macchina, come vari reset, codifiche di varie memorie in cascata e così via) ad eccezione di uno soltanto, che verifica il funzionamento della macchina nel caso di ricezione in ingresso di una "memoria minima".

Tutti i test sono stati superati, sia in *Behavioural* che in *PostSynthesis*; in particolare, il *test tb_example* è stato superato in 62000 nanosecondi (*Behavioural*).

La macchina viene sintetizzata utilizzando 103 *flip-flop* e 82 *Look Up Tables*, ed è in grado di funzionare correttamente fino ad una velocità di *clock* di cinque nanosecondi.

3.2 Ottimizzazione

Lo svolgimento dei *test* ha portato, nel tempo, a numerose opere di ottimizzazione, volte a semplificare il *design* della macchina e a velocizzarne l'esecuzione.

Il primo ostacolo affrontato è stato quello della velocità: la prima iterazione del componente completava il *test tb_example* in ben 88 desolanti cicli di *clock*, e veniva sintetizzato con 108 *flip flop*. Riducendo gli stati della macchina (da venti stati originali fino a dodici), il componente era stato portato verso un funzionamento molto più rapido, con 55000 nanosecondi per completare lo stesso *test* e 103 *flip flop* necessari alla sintesi.

Questa seconda versione ha presentato, a sua volta, problemi ben più onerosi da risolvere. Innanzitutto, un'implementazione non molto "robusta" comportava il fallimento di alcuni *test* in *PostSynthesis*; in secondo luogo, seppur veloce, il componente non era in grado di agire con un clock inferiore ai trenta nanosecondi.

Per risolvere questi problemi si è deciso di sacrificare in parte la velocità di esecuzione (salendo a 62000 nanosecondi per *tb_example*) in funzione di una progettazione più solida e attenta a una corretta sintetizzazione della macchina, e volta al permettere l'operatività a velocità di *clock* molto piccole.

In quest'ottica sono stati inseriti dei nuovi stati (come *WAITFORSTART* e *DECRMENTTOTAL* per esempio), distribuendo maggiormente la logica per evitare, come in precedenza, stati ricolmi di funzioni logiche molto differenti fra loro. Questo approccio ha consentito di raggiungere un duplice obiettivo: in primis la risoluzione di ogni problema di sintesi e la possibilità di poter lavorare con velocità di *clock* fino a cinque nanosecondi, ben 1/20 dello standard di partenza. Infine, la stesura di un *design* più facilmente leggibile, qualità che comporta una maggior semplicità nella manutenzione del codice.