Prova Finale - Progetto di Reti Logiche

Prof. Fabio Salice - Anno 2021/2022

Alberto Nidasio - 10665344 / 934129

25 giugno 2022

Indice

1 Introduzione

1.1 Scopo del progetto

Lo scopo del progetto è quello di implementare un modulo hardware, descritto in VHDL, che applichi il codice convoluzionale $\frac{1}{2}$ ad un flusso di bit salvato in memoria. Il flusso di bit generato dovrà essere a sua volta memorizzato.

1.2 Codice convoluzionale $\frac{1}{2}$

Un codice convoluzionale è un tipo di codifica nel quale l'informazione, composta da m bit, viene trasformata in un flusso di n bit, dove m/n è il rapporto del codice o tasso di trasmissione.[1]

Nel caso in esame, il codice convoluzionale ha un rapporto $\frac{1}{2}$, quindi per ogni bit di informazione vengono generati due bit. Per il calcolo del flusso in uscita viene seguito lo schema riportato in figura 1 dove i nodi rettangolari rappresentano dei flip flop e i nodi rotondi delle somme.

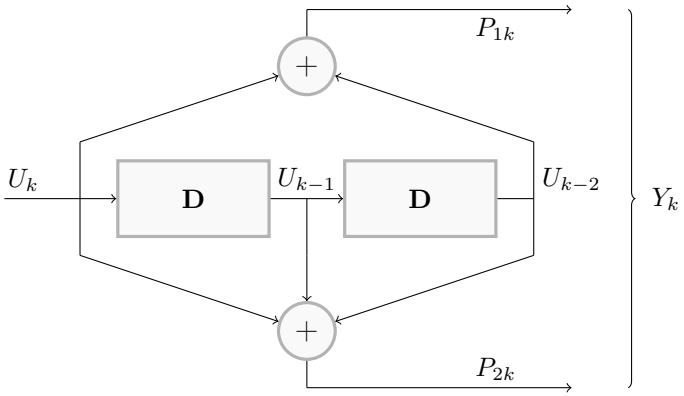

Figura 1: Codificatore convoluzionale con tasso di trasmissione $\frac{1}{2}$

Considerando il flusso U di bit in ingresso e indicando con k l'instante di tempo considerato, la codifica legge il bit U_k e produce i bit P_{1k} e P_{2k} calcolati come mostrato in eq. 1 e 2. È importate sottolineare che il convolutore mantiene in memoria, ad ogni istante di tempo, gli ultimi due bit precedentemente letti, ovvero U_{k-1} e U_{k-2}, i quali sono inizializzati a 0. Inoltre, per produrre i bit di P, viene applicato l'operatore XOR ($\oplus$) in modo tale da ottenere una somma senza riporto.

$$P_{1k} = U_k \oplus U_{k-2} \tag{1}$$

$$P_{2k} = U_k \oplus U_{k-1} \oplus U_{k-2} \tag{2}$$

Il flusso in uscita sarà la concatenazione dei bit P_{1k} e P_{2k}. Il convolutore è quindi una macchina sequenziale sincrona con un clock globale che scandisce l'ingresso dei bit del flusso U e il calcolo dei bit di P.

[1] *Codice convoluzionale*.

In figura 2 è riportato un esempio in cui vengono codificati 8 bit. Il flusso in uscita è quindi composto come: $P_{10}, P_{20}, P_{11}, P_{21}, ..., P_{17}, P_{27}$.

T	0	1	2	3	4	5	6	7
U_k	1	0	1	0	0	0	1	0
P_{1k}	1	0	0	0	1	0	1	0
P_{2k}	1	1	0	1	1	0	1	1

$$U = 10100010$$

$$P = 1101000111001101$$

Figura 2: Esempio di codifica di 8 bit

1.3 Descrizione della memoria

Il modulo da implementare deve leggere il flusso U da una memoria con indirizzamento al byte. All'indirizzo 0 è presente la quantità di parole W da codificare e, a partire dall'indirizzo 1, i byte U_k. La dimensione massima della sequenza di ingesso è di 255 byte. Il flusso in uscita dovrà essere memorizzato a partire dall'indirizzo 1000.

In figura 3 è rappresentato il contenuto della memoria.

Indirizzo	Contenuto
0	W
1	U_0
2	U_1
...	...
W	U_{W-1}
...	...
1000	P_0
1001	P_1
...	...
$1000 + 2W - 1$	P_{2W-1}

Figura 3: Contenuto della memoria

1.4 Specifiche di funzionamento

Il componente da descrivere in VHDL deve implementare l'interfaccia indicata in figura 4 e rispettare le seguenti caratteristiche:

- L'elaborazione parte quando il segnale `i_start` viene portato a 1;
- Il segnale `o_done` deve essere portato a 1 per indicare il termine della computazione;
- Il modulo deve essere in grado di ripartire ogni qual volta il segnale `i_start` viene portato a 1;
- Il modulo deve resettare il proprio stato ogni volta che il segnale `i_rst` viene portato a 1.

Inoltre il segnale `i_start` rimarrà alto finché `o_done` rimane basso e il segnale `i_rst` verrà inviato solamente una volta, successive elaborazione dovranno ripartire solamente con `i_start`.

La memoria è collegata al modulo tramite i segnali `i_data`, `o_address`, `o_en`, `o_we` e `o_data`. Per comunicare con essa è necessario abilitarla portando a 1 il segnale `o_en`, mentre `o_we` indica

```
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;
```

Figura 4: Interfaccia del modulo

la modalità di scrittura, se alto, o la modalità di lettura, se basso. Se abilitata, la memoria si attiva sul rising edge del clock. Nel caso sia impostata la modalità di lettura, essa riporta il valore memorizzato all'indirizzo `o_address` sul segnale `i_data` con un determinato ritardo di tempo. Altrimenti, se in modalità di scrittura, memorizza nell'indirizzo specificato il valore riportato dal segnale `o_data`.

Durante un'operazione di lettura è importante attendere che la memoria abbia avuto sufficiente tempo per riportare sul segnale `i_data` il valore richiesto. A tal proposito è ragionevole assumere che il ritardo che caratterizza l'operazione non sia più grande del periodo di clock. Quindi, per progettare un componente conviene attendere un ciclo di clock prima di leggere il risultato, invece che campionarlo subito dopo aver abilitato la memoria. All'interno di una macchina a stati questo si traduce in uno stato di attesa interposto tra l'invio del comando e la lettura del dato.

2 Architettura

Il componente è organizzato tramite una macchina a stati che scandisce gli accessi alla memoria e le fasi di elaborazione. All'avvio e dopo ogni reset, la macchina rimane in attesa del segnale `i_start` e, dopo averlo ricevuto, recupera la dimensione W del flusso U e successivamente esegue in sequenza la lettura di un byte U_k, il calcolo dei due byte derivanti da U_k e la loro scrittura in memoria. Questo ciclo si conclude quando tutti i W byte sono stati elaborati.

2.1 Macchina a stati

La macchina a stati sintetizzata, mostrata in figura 5, è composta da i seguenti 12 stati:

- **IDLE**: La macchina è in attesa del segnale `i_start` e rimane in questo stato finché il segnale non viene portato a 1;
- **PREPARE_W** e **REQUEST_W**: Vengono impostati i segnali di controllo che si attivano durante la transizione tra questi due stati, la memoria viene quindi attivata in modalità lettura indicando l'indirizzo 0 per recuperare W;
- **WAIT_W** e **FETCH_W**: Si attende che la cella di memoria venga letta. Quindi l'output in `i_data` viene memorizzato. Sarà poi utilizzato per contare i cicli di computazione. Nel caso W fosse pari a 0, il modulo passa subito nello stato **DONE** portando a 1 il segnale `o_done`, altrimenti continua;
- **PREPARE_U**, **REQUEST_U** e **WAIT_U**: Viene utilizzata la memoria ancora in modalità lettura per recuperare un byte del flusso U, come indirizzo viene utilizzato il numero del ciclo corrente memorizzato nel segnale `U_count`;
- **COMPUTE_P**: In questa fase viene letto il byte U e vengono calcolati due byte del flusso in uscita, vedi sezione 2.2;
- **WRITE_P1** e **WRITE_P2**: Vengono scritti in memoria i due byte del flusso P calcolati precedentemente. Se sono stati elaborati tutti i W byte del flusso U, la macchina passa nello stato **DONE** portando a 1 il segnale `o_done`, altrimenti continua ripartendo da **PREPARE_U**;
- **DONE**: La computazione è terminata e viene atteso il reset del segnale `i_start` per riportare a zero `o_done` e tornare nello stato di **IDLE**.

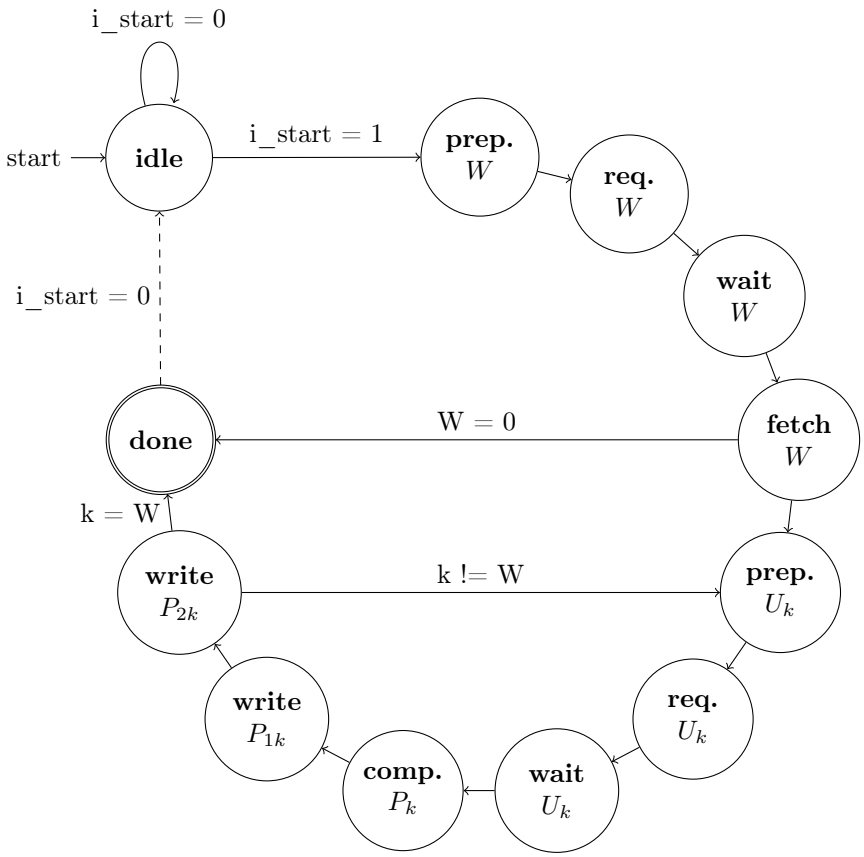

Figura 5: Macchina a stati del componente implementato

2.2 Calcolo del flusso Z

Il convolutore descritto in sezione 1.2 potrebbe essere implementato tramite la macchina a stati illustrata nelle specifiche del progetto[2], avendo quindi 4 stati ed elaborando ad ogni step i bit P_{1k} e P_{2k}. In questo modo ci vorrebbero 8 cicli di clock per concludere l'elaborazione di un byte letto da memoria. Considerando la dimensione di indirizzamento, è stato utilizzato un approccio differente che permette di elaborare un byte del flusso U in un singolo ciclo di clock.

All'interno del componente viene mantenuto un buffer composto da un vettore di 2 elementi in grado di memorizzare gli ultimi due bit del byte U letto durante il ciclo precedente. Nello stato **COMPUTE_P** il byte U corrente viene letto dalla memoria e associato ai 2 bit del buffer tramite una variabile (un vettore di 10 elementi). Grazie a questo possiamo eseguire le equazioni 1 e 2 su ciascuno degli 8 bit letti in un singolo ciclo. Come mostrato in figura 6, in VHDL questo viene ottenuto tramite un ciclo `for` che elabora il byte in ingesso e che produce i due byte del flusso in uscita P. Tale codice applica quindi il convolutore a tutti i bit letti come mostrato in figura 7.

Il risultato è un componente che riesce ad elaborare ciascun byte letto dalla memoria in un singolo ciclo di clock, riducendo così il tempo che il convolutore impiega per elaborare il flusso

[2] *PFRL_Specifica_21_22_V3.*

in ingresso U. Il vantaggio in termini di tempo viene ottenuto a discapito delle risorse hardware utilizzare, infatti vengono sintetizzate le eq. 1 e 2 per 8 volte.

```
-- Shift the current value and append U from memory
next_U_buffer <= i_data(1 downto 0);
U             := U_buffer(1 downto 0) & i_data;

-- Compute the 1/2 convolutional code
for k in 7 downto 0 loop
    -- Compute P1k and P2k
    P(k * 2 + 1) <= U(k + 2) xor U(k);
    P(k * 2)     <= U(k + 2) xor U(k + 1) xor U(k);
end loop;
```

Figura 6: Calcolo di due byte del flusso P

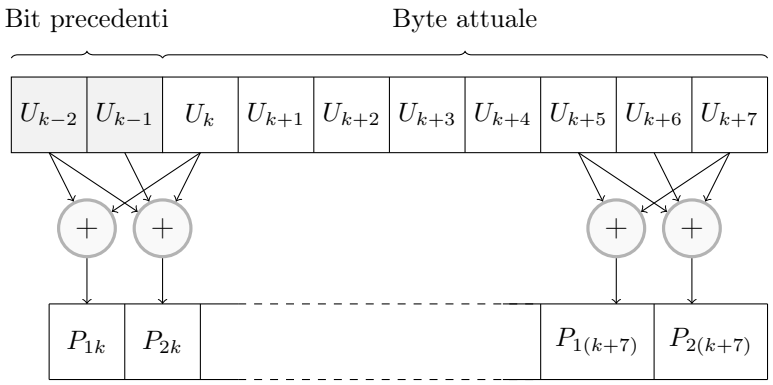

Figura 7: Schema del calcolo di due byte del flusso P

2.3 Architettura del componente

Il componente descritto in VHDL è organizzato in due processi separati.

Il primo rimane in ascolto dei segnali `i_rst` e `i_clk`. Nel caso il segnale di reset venga alzato, lo stato viene resettato immediatamente riportando lo stato ad `IDLE` e resettato il buffer utilizzato per il calcolo di P e tutti gli altri segnali. In caso contrario, al rising edge del clock si avanza allo stato successivo aggiornando i registri.

Il secondo processo si occupa invece di eseguire le operazioni rispetto allo stato corrente e si attiva ogni qual volta esso viene aggiornato, oppure quando il segnale `i_start` cambia valore. In quest'ultimo caso, se il segnale è alto e lo stato corrente è **IDLE**, la macchina si avvia passando allo stato successivo, altrimenti permane nello stato in cui si trova.

All'interno dell'architettura vengono inoltre usati diversi registri: `current_state`, `U_count`, `U_buffer` e `P_buffer` oltre che ai segnali collegati alla memoria. Per gestire i loro valori nel passaggio da uno stato all'altro, è stato definito un segnale ausiliario per ciascuno di essi. Questi segnali vengono modificati dalle operazioni di ciascuno stato e poi sostituiti alle loro controparti nel momento in cui lo stato viene aggiornato a causa del clock.

3 Risultati sperimentali

3.1 Sintesi

Il dispositivo progettato è perfettamente sintetizzabile ed occupa le risorse mostrate in figura 8.

Resource	Utilization
LUT	63
FF	60
IO	38
BUFG	1

Figura 8: Risorse hardware post implementazione

Analizzando il timing report mostrato in figura 9 è possibile vedere che il tempo di esecuzione del componente rispetta il limite di $100ns$ per il periodo di clock.

Setup	
Worst negative slack (WNS)	$95.491ns$
Total negative slack (TNS)	$0.00ns$
Number of failing endpoints	0
Total number of endpoints	98
Hold	
Worst hold slack (WHS)	0.239
Total hold slack (THS)	$0.000ns$
Number of failing endpoints	0
Total number of endpoints	98
Pulse width	
Worst pulse width slack (WPWS)	$49.500ns$
Total pulse width negative slack (TPWS)	0.000
Number of failing endpoints	0
Total number of endpoints	61

Figura 9: Timing report

3.2 Simulazioni

Lo sviluppo del dispositivo in codice VHDL è stato validato tramite i test bench forniti per il progetto. Tali test coprono diversi casi particolari come i casi limite di minima e massima lunghezza del flusso U, reset multipli ed elaborazioni successive di diverse sequenze. Oltre a questi, sono stati eseguiti anche diversi test bench generati casualmente tramite uno script il quale prepara diversi flussi U. Ciascun test bench provato in questa maniera conteneva 1000 flussi differenti che il dispositivo doveva elaborare.

3.3 Correzione delle simulazioni comportamentali

Successivamente alla consegna del progetto, il docente mi ha fatto notare come il componente sviluppato fallisse le simulazioni comportamentali mentre passasse tutte le altre. La causa del problema era un errato accesso alla memoria nelle le fasi di lettura, durante le quali non si

considerava il tempo necessario per recuperare il dato. In particolare, sia la memoria che lo stato del componente venivano aggiornati sul fronte di salita del clock. Dagli stati **REQUEST_W** e **REQUEST_U**, nei quali si impostano i segnali `o_address` e `o_en`, si passava immediatamente alle loro controparti **FETCH_W** e **FETCH_U**, nelle quali si legge il segnale `i_data`. La memoria percepisce però le modifiche sui segnali solo al successivo rising edge, proprio nello stesso momento in cui ci si aspettava di poter leggere il dato.

Durante le simulazioni comportamentali viene tenuto conto soltanto dei ritardi direttamente specificati, come il ritardo di $2ns$ utilizzato nella descrizione della memoria nei test bench, mentre non si considerano i ritardi delle porte logiche che andranno a costituire il componente. Per questo motivo, durante le simulazioni comportamentali, nel momento del rising edge, la memoria si attivava e veniva immediatamente letto il segnale `i_data` che però non poteva contenere ancora i dati siccome la memoria ha un ritardo.

Per correggere il problema è stato implementato un accesso alla memoria più robusto aggiungendo 2 stati e bufferizzando i segnali collegati alla memoria. Una operazione di lettura segue ora 4 step:

- **PREPARE**: Vengono impostati i segnali `o_address_next` e `o_en_next` che verranno applicati alla memoria al prossimo rising edge del clock;
- **REQUEST**: Questo è a tutti gli effetti uno stato di attesa, è necessario in quanto i segnali `o_address` e `o_en` vengono modificati in questo momento;
- **WAIT**: Si attende che la memoria legga il dato e lo riporti sul segnale `i_data`. Qui si assume che il ritardo di accesso alla memoria non sia più grande di un periodo di clock, altrimenti sarebbe necessario aspettare ulteriore tempo;
- **FETCH**: Il dato si considera disponibile e `i_data` viene letto;

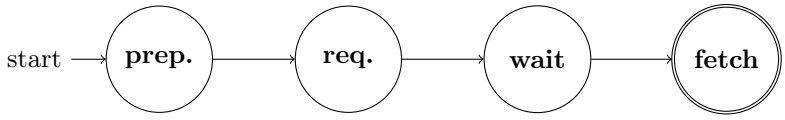

Figura 10: Sequenza di un'operazione di lettura da memoria

Sono quindi stati aggiunti 4 nuovi stati: **PREPARE_W**, **WAIT_W**, **PREPARE_W** e **WAIT_U**. Inoltre si è cercato di ridurre il numero degli stati inserendo la lettura del byte U direttamente in **COMPUTE_U**, evitando così **FETCH_U**. Non è stato possibile fare altrettanto con **FETCH_W** perché esso deve essere eseguito una volta sola mentre **PREPARE_U** viene ripetuto.

Per verificare il corretto funzionamento del nuovo design si sono ripetute le simulazioni e si è provato a variare anche il ritardo caratteristico della memoria. Il componente ora funziona con qualsiasi ritardo al di sotto del periodo di clock. Sono stati controllati anche i casi limite di ritardo nullo e ritardo maggiore del periodo di clock. Rispetto a quest'ultimo caso la macchina riesce a gestire ritardi anche leggermente maggiori rispetto all'ipotesi fatta, provandosi così molto più resiliente a quanto fosse prima.

Questo problema non era stato individuato per due motivi. Durante lo sviluppo il componente era stato configurato rispetto al falling edge del clock, separando così i due eventi e non riscontrando il problema nelle simulazioni comportamentali. Il risultato era una sequenza simile a quella indicata in figura 10. Successivamente la sensibilità era stata spostata sul rising edge e, siccome le simulazioni funzionali e temporali non rilevavano errori, non erano più state svolte simulazioni comportamentali.

4 Conclusioni

Il progetto è stato svolto in diversi step. Inizialmente ho studiato il linguaggio VHDL seguendo varie lezioni tenute dai docenti, integrandole con alcuni video e articoli trovati online. Successivamente sono passato a lavorare su Vivado, sintetizzando diversi esempi svolti a lezione. Questo mi ha permesso di capire il funzionamento del software, come funzionano le simulazioni tramite i test bench e come analizzare le forme d'onda.

Successivamente ho studiato la specifica del progetto e in particolare il funzionamento del convolutore. La comprensione del suo funzionamento è stata veloce principalmente grazie ai grafici autoesplicativi riportati nel documento. Sono poi passato a studiare l'accesso alla memoria e in questo caso i miei dubbi sono stati chiariti dalla descrizione in VHDL della stessa. Con queste informazioni ho potuto progettare la macchina a stati che inizialmente comprendeva più step perché implementavo direttamente la macchina a stati del convolutore. Una volta resomi conto che potevo ridurre i cicli di clock necessari per computare il flusso in uscita, sono riuscito a ridurre notevolmente il tempo di esecuzione globale.