Relazione di vhdl

Verdicchio Giacomo e Zarbo Nicola

25 marzo 2022

Indice

1 Introduzione

Progetto svolto da Nicola Zarbo(10677923) e Giacomo Verdicchio(10703196).

Lo scopo del progetto è stato creare un componente hardware che possa leggere dalla memoria un flusso di parole da 8 bit, convertirlo tramite un codificatore convoluzionale e scriverlo in memoria, rispettando il constraint di clock di almeno 100 ns, come da specifica di progetto.

Durante la fase di implementazione:
-abbiamo cercato di garantire la più alta leggibilità del codice e riadattabilità del modulo, in caso di possibili modifiche e ampliamenti futuri, mantenendo comunque una buona efficienza in termini di word per clock elaborate;
-abbiamo prestato poca attenzione alla quantità di componenti (look up table, flip flop), ma ponendone molta nell'evitare un qualsiasi utilizzo, erroneo o volontario, di latch

2 Architettura

Il nostro hardware è composto da un modulo principale (modulo 1), di cui sotto specificato, che utilizza al suo interno il codificatore convoluzionale (modulo 2) che legge e scrive dati da 2 registri gestiti dal modulo 1.

Durante la progettazione del modulo abbiamo preferito suddividere in

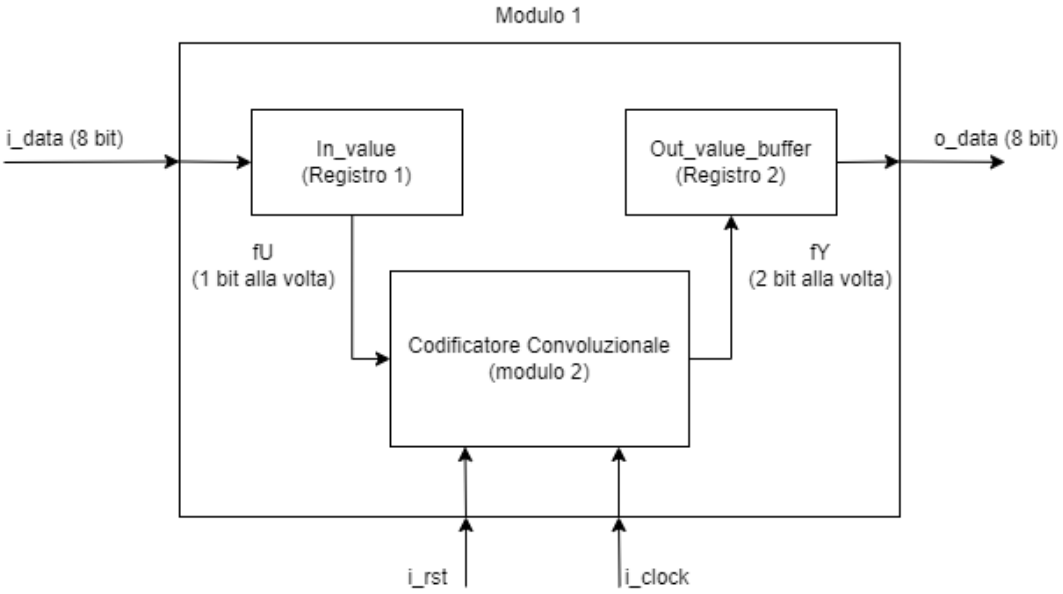


Figura 1: Schematico ad alto livello della composizione generale del modulo, senza entrare troppo nello specifico di tutti i segnali, ma solo mostrando l'utilizzo del componento interno (codificatore convoluzionale)

2.1 Modulo 1

Il modulo gestisce la lettura e scrittura da memoria e i segnali del modulo 2 'codificatore convoluzionale'.

2.1.1 In,Out Signals

- i_clk : segnale di clock
- i_rst : segnale di reset asincrono
- i_start : segnale di enable '1'=> operativo,'0' => fermo (quindi il modulo viene resettato)
- i_data : bus 8 bit, dati di lettura da ram
- o_address : bus 16 bit, comunica alla ram l'indirizzo su cui eseguire lettura/scrittura
- o_done : seganale di finita elaborazione '1' => flusso elaborato/scritto in memoria
- o_en : segnale di enable per ram
- o_we : segnale per comunicare alla ram quale operazione svolgere, '0'=> read, '1' => write
- o_data : bus 8 bit, dati in scritttura per ram

2.1.2 Registri

- stato_att, st_prox (8 bit): registri di stato per fsm
- in_value (4 bit): dove viene copiato la parola di 8 bit letta da i_data
- out_value_buffer (8 bit) : dove viene scritta la parola da scrivere, collegato a o_data
- in_addr (16 bit): per mantenere address per lettura e per controllo terminazione codifica
- in_a_prox (16 bit): per incrementare l'address per l'operazione di read
- out_addr, out_a_prox (16 bit):registri per mantenere e incrementare address per write
- nTerminazione (9 bit): usato per controllo terminazione, mantiene il valore della cella ram '0000' incrementato di 1

2.1.3 Segnali per componente interno

- fU : flusso di bit in lettura da codificare
- fY : flusso di 2 bit in uscita da codificatore
- stop_en : segnale per fermare la macchina a stati del codificatore al di fuori dei clock di codifica

2.1.4 FSM

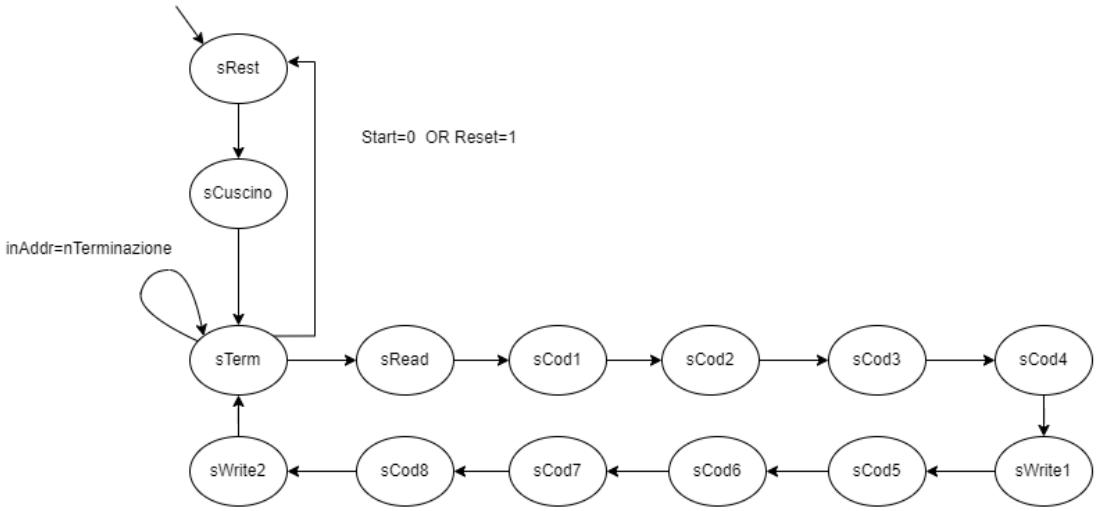


Figura 2: Disegno esplicativo della FSM

Descrizione stati:

- sReStart : stato di reset, viene letto la cella all'indirizzo '0000' e il suo valore incrementato di uno viene salvato in nTerminazione
- sCuscino : stato attraversato solo una volta in tutta l'operzione di codifica, serve per consentire funzionamento con n di parole nullo (per ulteriori info vedere test seq_min)
- sTerm : controllo terminazione codifica, confronta il numero totale di parole da leggere (+ 1) con il prossim indirizzo di lettura
- sRead : stato di lettura, fornisce alla ram i segnali per leggere la prossima parola da elaborare

- sCod1 : codificatore in funzionamento, viene salvata la parola appena letta da i_data nel registro in_value, viene inserito in fU il primo bit dalla parola letta
- sCod2 to sCod4 : inserito in fU il nuovo bit da leggere preso da in_value, bit codificati da fY salvati in out_value_buffer nell'apposita posizione
- sWrite1 : inseriti bit da fY negli ultimi due bit del registro
  out_value_buffer, forniti segnali alla ram per scrivere la parola appena codificata, codificatore bloccato
- sCod5 to sCod8 : codificatore in funzionamento, funzionamento equivalente a sCod1-sCod4, usando gli utlimi 4 bit di in_value
- sWrite2 : equivalente a sWrite1

2.2 Modulo 2 : Codificatore Convoluzionale

Il codificatore convoluzionale è il modulo che si occupa dell'effettiva codifica dei dati in ingresso

2.2.1 In,Out Signals

- i_U : flusso di bit in ingresso
- i_start : segnale di enable del componente, se off il componente viene resettato
- i_rst : segnale di reset asincrono
- stop_en : segnale di enable del componente, se off viene mantenuto in standby
- o_Y : flusso di 2 bit da codifica in uscita

2.2.2 Registri

st_att, s_pros : regsitri per stati della fsm di mealy

2.2.3 FSM

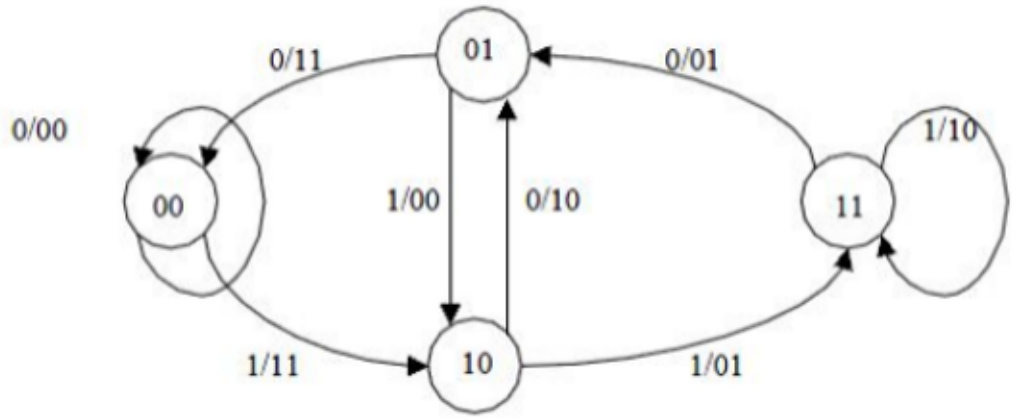


Figura 3: Disegno esplicativo del codificatore convoluzionale

3 Risultati sperimentali

3.1 Sintesi

(report di sintesi)

1. Slice Logic

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	84	0	134600	0.06
LUT as Logic	84	0	134600	0.06
LUT as Memory	0	0	46200	0.00
Slice Registers	97	0	269200	0.04
Register as Flip Flop	97	0	269200	0.04
Register as Latch	0	0	269200	0.00
F7 Muxes	0	0	67300	0.00
F8 Muxes	0	0	33650	0.00

1.1 Summary of Registers by Type

Total	Clock Enable	Synchronous	Asynchronous
0	_	-	-
0	_	-	Set
0	_	-	Reset
0	_	Set	-
0	_	Reset	-
0	Yes	-	-
0	Yes	-	Set
8	Yes	-	Reset
7	Yes	Set	-
82	Yes	Reset	-

```
7. Primitives
-------------

+----------+------+---------------------+
| Ref Name | Used | Functional Category |
+----------+------+---------------------+
| FDRE     |   82 |        Flop & Latch |
| LUT1     |   35 |                 LUT |
| LUT6     |   34 |                 LUT |
| OBUF     |   27 |                  IO |
| IBUF     |   11 |                  IO |
| LUT4     |   10 |                 LUT |
| CARRY4   |   10 |          CarryLogic |
| FDCE     |    8 |        Flop & Latch |
| FDSE     |    7 |        Flop & Latch |
| LUT3     |    4 |                 LUT |
| LUT2     |    4 |                 LUT |
| LUT5     |    1 |                 LUT |
| BUFG     |    1 |               Clock |
+----------+------+---------------------+
```

```
Hierarchical RTL Component report
Module project_reti_logiche
Detailed RTL Component Info :
+---Adders :
       2 Input     16 Bit       Adders := 2
       2 Input      9 Bit       Adders := 1
+---Registers :
                   16 Bit    Registers := 2
                    4 Bit    Registers := 1
+---Muxes :
       2 Input     16 Bit        Muxes := 2
       2 Input      8 Bit        Muxes := 3
       2 Input      4 Bit        Muxes := 1
      17 Input      4 Bit        Muxes := 1
       2 Input      1 Bit        Muxes := 5
Module codificatore_convoluzionale
Detailed RTL Component Info :
+---Registers :
                    2 Bit    Registers := 2
+---Muxes :
       6 Input      2 Bit        Muxes := 1
       4 Input      1 Bit        Muxes := 2
```

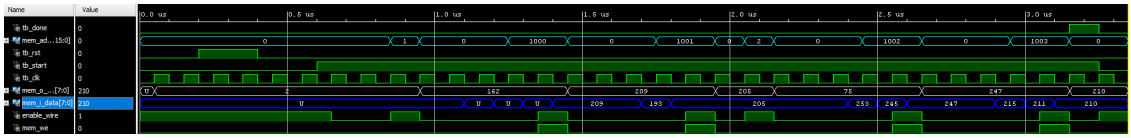
Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	45,744 ns	Worst Hold Slack (WHS):	0,142 ns	Worst Pulse Width Slack (WPWS):	49,500 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	169	Total Number of Endpoints:	169	Total Number of Endpoints:	98

All user specified timing constraints are met.

3.2 Simulazioni

Le simulazioni che abbiamo creato servono a testare i casi limite che potrebbero mandare in loop, blocco o crash la macchina a stati da noi svilupata



3.2.1 Esempio generico

Eseguo il programma con dei dati in input generici e verifico che tutti i segnali, soprattutto quelli in uscita, siano corretti ponendo particolare attenzione alla gestione dei registri di utilizzati per le operazioni di lettura e scrittura -tb_done, per verificare la corretta terminazione della macchina –mem_o_data, valori in uscita -mem_i_data, valori letti -mem_address, da cui vado a leggere l'indirizzo di memoria da cui leggo

3.2.2 Simulazione max

Riempio la RAM (inserendo 28 valori, cioè 255) e verifico che il programma gestisca correttamente tutti gli input, in particolare verificando con cura che non si realizzi un overflow nel registro di lettura. Segnali e registri particolari che abbiamo analizzato sono stati: -tb_done, per la terminazione del processo -mem_o_data, valori in uscita -mem_i_data, valori letti -enable_wire

3.2.3 Simulazione min

Inserisco solo alcuni zeri per verificare che la memoria li gestisca correttamente, cioè salvando solo zeri, senza elaborare numeri errati. I segnali più sfruttati nel caso particolare sono stati: -tb_start per poter provare i restart della macchina, -tb_rst per poter provare i reset, -t. . . . . .

3.2.4 Simulazione reset multipli

Resetto più volte la macchina per verificarne la corretta gestione del reset, sia quando la macchina è già in esecuzione e dei vari registri e sia quando è ferma, nel dettaglio i segnali più sfruttati nel caso particolare sono stati: -tb_rst, -tb_done

3.2.5 Simulazione start multipli

Pongo a 1 più volte il segnale di start per verificare la corretta gestione e specifico 3 macchine RAM separate gestite separatamente, verificando la corretta gestione dei segnali di terminazione e lo start dei vari blocchi RAM, andando a selezionarli tramite un contatore opportunamente dichiarato (quindi quando uguale a 1 seleziono la prima RAM, quando a 2 la seconda e così via). I segnali importanti in questo caso sono: -tb_start

4 Conclusione

leggibilità over velocità

-siamo riusciti a progettare un modulo che rispetti pienamente il constraint di clock (100 ns) e, per come l'abbiamo realizzato, è in grado di funzionare con clock molto più bassi (fino anche a 10 ns, con simulazione su Vivado in Post Synthesis);

tutto facilmente moidificabile e leggibile

4.1 Note particolari modulo 1

La fsm usa molti stati equivalenti, scelta adottata per favorire la comprensibilità usando un approccio simile agli automi a stati finiti, evitando quindi il più possibile dei controlli per la scelta del prossimo stato, tranne ovviamente in sTerm, dove viene verificata la terminazione della codifica.

4.2 Note particolari modulo 2

La codifica avviene tramite questo componente, pensato per rispecchiare il più possibile quello descritto dalla specifica e per essere facilmente utilizzabile in contesti diversi senza bisogno del Modulo 1. Questo tipo di codificatori sono usati nelle telecomunicazioni, quindi potrebbe essere utilizzato collegando l'uscita ad un componente per trasmettere direttamente il segnale codificato invece di salvarlo su una memoria.