Elaborato del Progetto di Reti Logiche

Oleg Nedina

POLITECNICO
MILANO 1863

Anno Accademico: 2024/2025
Matricola: 225897 Codice Persona: 10905938
Docente Referente: Fabio Salice

Indice

CAP.1: INTRODUZIONE E DEFINIZIONE DEL PROGETTO

1.0 INTRODUZIONE

L'elaborato si occupa di descrivere il progetto finale di *Reti Logiche* svolto relativamente all'anno accademico 2024/25. Si vanno ad affrontare i problemi posti dalla specifica, il modo in cui questa è stata analizzata e i vari tentativi che hanno portato alla sintesi del modulo finale, comprendendo considerazioni su ciò che si è testato e sui possibili margini di miglioramento, i quali però verrano solo accennati non essendo stati richiesti esplicitamente dalla stessa.

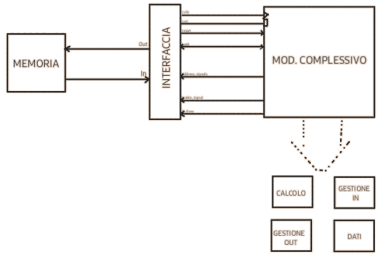
Il progetto è stato svolto individualmente da me: Oleg Nedina, Matricola 225897, Codice Persona 10905938, con professore di riferimento Fabio Salice.

L'elaborato comprenderà varie parti; doveroso però informare che la specifica non verrà completamente trascritta all'interno del lavoro, ma verranno riportate solo le informazioni utili per comprendere le scelte di design. Si rimanda al file ufficiale fornito dai docenti per maggiori dettagli.

1.0.1 DEFINIZIONE DELLA SPECIFICA

La specifica richiede di creare un modulo in VHDL che, dopo un segnale di *start*, richieda la lettura di dati da memoria per il setup di un filtro differenziale da applicare successivamente ad una sequenza di ordine prestabilito durante la fase iniziale.
I vari dati dopo essere stati modificati tramite il filtro verranno poi trascritti dal modulo stesso nella memoria con cui si interfaccia. Più nel dettaglio, ci si occupa, leggendo un byte alla volta, di:

1. leggere i primi 17 byte che contengono la lunghezza della sequenza di dati da rielaborare, l'ordine del filtro e i coefficienti del filtro stesso;
2. rielaborare e scrivere in memoria i dati della sequenza.



Il modulo quindi è composto principalmente da un'interfaccia che presenta tutti i segnali standard per comunicare con la memoria e vari "sotto moduli" interni per il corretto comportamento dello stesso.
Si riportano i segnali di comunicazione e una breve spiegazione:

- `i_clk`: consente la sincronizzazione sul medesimo segnale di clock fornito dalla memoria;
- `i_rst`: unico segnale asincrono; corrisponde al reset del sistema ed è usato per inizializzare la macchina prima dell'arrivo del segnale di *start*;
- `i_start`: segnale che avvia il funzionamento del modulo;
- `o_done`: segnale che permette al modulo di notificare alla memoria la fine della rielaborazione;
- `o_mem_addr`: segnale in uscita per inviare l'indirizzo alla memoria;
- `i_mem_data`: segnale in ingresso contenente il dato in seguito alla richiesta di lettura;
- `o_mem_data`: segnale in uscita contenente il dato da scrivere in memoria;
- `o_mem_en`: segnale di enable per comunicare con la memoria;
- `o_mem_we`: segnale di enable per la scrittura in memoria.

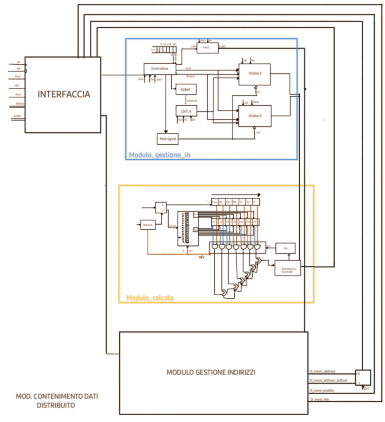
Si considerino gli *enable* attivi a `1`.
I segnali contenenti dati sono vettori, mentre i segnali di *clock*, *start*, *done*, *rst* ed *enable* sono a singolo bit.

1.1 DEFINIZIONE E LOGICA DELLE COMPONENTI

Il componente si comporta come una macchina che gestisce tutta l'interazione con la memoria al di fuori dello *start* e del *reset*.
Richiede i vari dati per configurare il propio setup e poi inizia con la elaborazione dei dati di sequenza.

È divisibile in cinque parti fondamentali: *interfaccia di comunicazione*, *modulo richiesta di lettura*, *modulo di contenimento dati*, *modulo di applicazione del filtro*, *modulo di richiesta scrittura*.

L'organo di controllo è una macchina a stati finiti (FSA); ogni blocco è stato suddiviso in sottofasi facenti parte della stessa. La descrizione dettagliata seguirà in seguito.



1.2 STORICO DELLE IMPLEMENTAZIONI E SCELTA DI DESIGN FINALE

Inizialmente il progetto è stato sviluppato come un insieme di vari moduli separati tra di loro, tutti volti a fare specifiche macro-azioni. Questo *approccio*, per quanto facilmente comprensibile a livello di scrittura, portava a codice molto lungo e a molti rischi di *incongruenze* tra la comunicazione dei vari moduli.

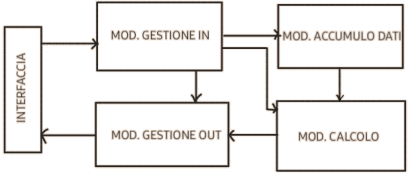
La logica implementata all'inizio non è stata però *abbandonata* in maniera completa. Prendendo sempre più padronanza con il programma di sviluppo (Vivado v. 2016.1) mi sono reso conto di come molte parti dei vari sotto-moduli non necessitassero di componenti *ad hoc*, ma il programma stesso fosse in grado, sfruttando le funzionalità standard, di creare automaticamente tutti i vari *mux*, *adders* etc... necessari alla costruzione di un modulo come quello richiesto da specifica.

Si passa quindi da una *versione 0* a una *versione 1* che comprendeva una vera e propria macchina a stati collegata ancora a vari sotto-moduli, i quali — versione dopo versione — sono stati sempre più implementati all'interno della FSA come stati veri e propri più che componenti a parte.

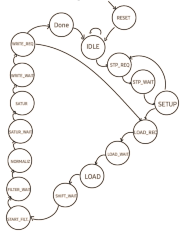
Si arriva così alla *versione 2*, una FSA che implementa un solo modulo esterno, ovvero uno *shift register* a 7 celle, ma non per una questione pratica bensì per una scelta personale.
Si cerca di implementare una logica classica, ovvero "uno stato per uno scopo preciso" e, in base a dei segnali interni (contatori), evolve di stato in stato in maniera differente nel tempo.

2 considerazioni sulle implementazioni:

- la prima è che la logica dei moduli iniziali si era dimostrata corretta ma incompatibile con uno sviluppo *scalabile* (questo dipende dal tipo di progetto richiesto); si è quindi preso ciò che funzionava indipendentemente dallo stato della macchina e lo si è trascritto sotto forma di stati;
- la seconda è che per garantire robustezza si sono utilizzati molti più stati del necessario; non avendo limiti a livello di performance ho preferito rimanere il più sicuro possibile.



Versione 0: modulare.

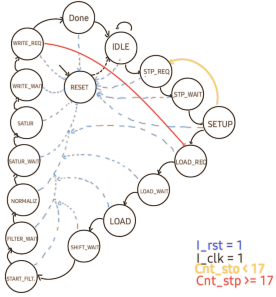


Versione 2: FSA finale.

2.0 LOGICA IMPLEMENTATA

È stata scelta una FSA ; questa scelta favorisce semplicità di scrittura e di comprensione, mantenendo al contempo una buona capacità di elaborazione. Ogni stato è da interpretare come un *sotto-modulo* che svolge determinate azioni al fine del corretto funzionamento complessivo.

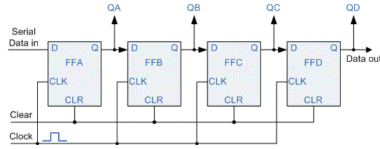
Il modulo è composto da due *process* sincroni principali: uno dedicato univocamente al calcolo dello *stato presente* e dello *stato prossimo*; l'altro dedicato alla implementazione dei calcoli svolti dai singoli stati della FSA.



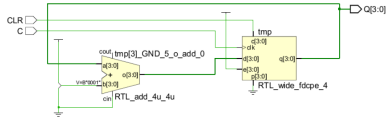
2.1 DEFINIZIONE DELLA FSA E FLUSSO COMPORTAMENTALE

La FSA comprende un totale di **17 stati** che verranno specificati nel paragrafo successivo.
Il flusso è quello tipico di un'architettura di questo tipo e in questo mini-paragrafo si descrivono i segnali e i componenti usati all'interno della implementazione:

- **Shift register a 7 celle** per mantenere i valori letti da memoria e usarli nel filtro; è collegato tramite 7 linee di uscita (una per dato), una linea di ingresso per il caricamento , un segnale di sincronizzazione con il resto della macchina e un segnale di rst.
- **Due segnali di tipo** `state` per tracciare l'evoluzione del modulo: `current_state` e `next_state`.
- **Due contatori interi**: `cnt_setup` (0..17) per i dati di setup, e `cnt_data` (0..2 147 483 647) per i dati da rielaborare.



Shift register 7 celle.



Adder+registro (logica di conteggio/accumulo).

- **Segnali `k1` e `k2`** per memorizzare la lunghezza della sequenza, combinati nel segnale `K`.
- **Segnale `s`** per il tipo di filtro (se S(0) $= 0 \rightarrow$ ordine 3, $1 \rightarrow$ ordine 5), mappato nel segnale di supporto `filter_type` a singolo bit.
- **Coefficienti** `C1..C14` memorizzati nell'ordine descritto dalla specifica; saranno usati nelle applicazioni del filtro.
- **Segnali di supporto** `acc` e `norm` (tipo `integer`) per facilitare i calcoli del filtro; il risultato, prima della scrittura in memoria, viene troncato in `result_byte` di tipo `std_logic_vector(7 downto 0)`.
- **Segnali di mappatura** per ciascun segnale dell'interfaccia (stesso tipo, non *in/out*) così da semplificare i calcoli e favorire la sintesi.

2.2 DEFINIZIONE DEGLI STATI

Si procede quindi a definire ogni stato della FSA e a specificare che cosa svolgono all'interno del modulo:

- **2.2.1 RESET**
  Azzera tutti i segnali del modulo e dei componenti collegati, in modo da reinizializzare la macchina.
- **2.2.2 IDLE**
  Raggiunto dopo il *reset*; è lo stato di attesa per *start* $= 1$. Lo *start* avvia il contatore di setup. Se dopo *reset* arriva subito *start*, questo stato è saltato e si passa al successivo.
- **2.2.3 SETUP_REQUEST**
  Porta alto il segnale di richiesta lettura da memoria e calcola l'indirizzo da cui leggere.
- **2.2.4 SETUP_WAIT_DATA**
  Stato di supporto per consentire l'assestamento dei segnali.
- **2.2.5 SETUP**
  Comportamento dipendente da `cnt_setup`. L'ordine dei dati (fisso per specifica) consente di associare, in base a `cnt_setup`, il dato letto al registro corretto. Se `cnt_setup` < 17 si torna a **SETUP_REQUEST**; altrimenti si passa a **LOAD_REQUEST** ed entra in funzione `cnt_data`.

- **2.2.6 LOAD_REQUEST**
  Calcola l'indirizzo di memoria di lettura del dato su cui verrà applicato il filtro.
- **2.2.7 LOAD_WAIT**
  Garantisce che l'indirizzo di memoria di lettura sia valido/stabile.
- **2.2.8 LOAD**
  Inserisce nello *shift_register* (inizialmente azzerato) il dato letto al ciclo precedente. Se `cnt_data < K+6` si prosegue; se `cnt_data = K+6` si va a **DONE**.
- **2.2.9 SHIFT_WAIT**
  Stato di supporto per l'assestamento dei segnali dopo lo shift.
- **2.2.10 START_FILTER**
  In base al *filter_type* (ordine 3 o 5) calcola `acc` come somma dei prodotti tra i 7 dati nello *shift* e i coefficienti: C_{i2} (i2 $= 2..6$) per ordine 3, oppure C_{i3} (i3 $= 8..14$) per ordine 5. Il calcolo è in `integer`; seguiranno conversione e saturazione prima del ritorno a `std_logic_vector`.
- **2.2.11 FILTER_WAIT**
  Stato di supporto per l'assestamento dei segnali dopo il calcolo di `acc`.
- **2.2.12 NORMALIZ**
  Prepara la normalizzazione: limita a 127 per eccesso e a -128 per difetto usando una variabile di supporto come accumulatore secondo la specifica. Per i valori negativi si applica la correzione $+1$ di approssimazione. Qui si calcola solo il numero da normalizzare; la saturazione vera e propria avverrà in **SATUR**.

- **2.2.13 SATUR_WAIT**
  Stato di supporto per la stabilizzazione dei segnali.
- **2.2.14 SATUR**
  Applica la saturazione a 127 / -128; se non necessaria, il dato resta invariato. Il risultato è scritto in `result_byte` convertito in `std_logic_vector(7 downto 0)`. Conclude i sotto–stati avviati da **START_FILTER**.
- **2.2.15 WRITE_WAIT**
  Stato di supporto; carica il dato calcolato in `result_byte` e attende l'assestamento.
- **2.2.16 WRITE REQUEST**
  Scrive il dato in memoria se `cnt_data` è > 3 e $< K + 3$. Calcola l'indirizzo di scrittura come base $+ 17 + K +$ `cnt_data` $- 3$; abilita i segnali di lettura/scrittura, incrementa `cnt_data` e ritorna a **LOAD_REQUEST**.
- **2.2.17 DONE**
  Come **RESET**, ma alza `done` per indicare la fine dell'elaborazione; segue reinizializzazione dell'intero modulo.

2.3 CONCLUSIONI SUL COMPORTAMENTO

La FSA ha un comportamento standard e coerente; i dati vengono scritti solo se `cnt_data` $\in (3, K+3)$ perché il filtro si applica al dato in **posizione centrale** dello *shift register* (quarta cella).

Per ottenere l'allineamento corretto bisogna dunque caricare almeno tre dati prima di iniziare ad elaborare.
Una volta fatto ciò si prosegue normalmente fino a quando non viene terminata la sequenza di K dati reali, dopo il quale è necessario caricare 3 zeri così che l'ultimo dato "scorra" al centro e il funzionamento resti corretto fino a completamento.

CAP.3: TB E RISULTATI SPERIMENTALI

3.0 STRUMENTI UTILIZZATI E OBIETTIVI DEI TEST

Lo strumento usato per effettuare i vari test di copertura è lo stesso impiegato per la progettazione e la sintesi del modulo, ovvero **Vivado** (versione **2016.1**). Per chiarezza, con `tb` si intende *testbench*.

Tutti i `tb` sono stati eseguiti tramite *Run Simulation* all'interno dell'applicativo, sia in *Behavioral Simulation* sia in *Post-Synthesis Functional Simulation*, e sono risultati **passati** in entrambe le modalità , ovvero il comportamente del modulo è stato quello previsto dai `tb` stessi.

I vari `test` cercano di coprire il maggior numero possibile di casi, mettendo in evidenza punti di forza e di debolezza del modulo. Non vengono riportati i primi `tb` usati per versioni precedenti del progetto; e si includono solo i più significativi per l'ultima versione (FSA integrata).

3.1 I TEST E LA LORO COPERTURA

- **3.1.1** `tb_2425`
  Test fornito dai docenti per verificare il funzionamento di base: *reset*, logica *start/done*, scrittura corretta in memoria. Nelle fasi iniziali ha motivato l'introduzione di più stati di *wait* e la specializzazione degli stati per ridurre le differenze dei risultati tra pre- e post-sintesi.
- **3.1.2** `tb_no_rewrite_input`
  Verifica che il modulo non sovrascriva i dati in ingresso durante l'elaborazione e controlla il corretto *handshake* memoria–modulo.
- **3.1.3** `tb_s_masking`
  Due run con lo stesso LSB di `S` per verificare che gli output coincidano e che la scelta del filtro dipenda solo da `S(0)`.
- **3.1.4** `tb_sign_symmetry`
  Doppia run con una sequenza `W` e una seconda `-W`; verifica coerenza rispetto al segno e le differenze dovute a normalizzazione/saturazione.
- **3.1.5** `tb_midrun_reset`
  Esegue una prima run `A`; dopo alcune scritture effettua un *reset* e avvia la run `B`; resetta anche `B`, la riavvia e verifica la coincidenza con i risultati attesi.

- **3.1.6** `tb_setup_reset`
  Esegue un *reset* durante la lettura del preambolo.
- **3.1.7** `tb_k_small_edges`
  Verifica proprietà di base con `K` molto piccoli ($K = 1, 2, 3$). Non richiesto da specifica, ma curioso da testare.
- **3.1.8** `tb_coeff_extremes_sat`
  Proprietà di base con dataset `W` che satura agli estremi a ogni scrittura; ha portato ad aumentare i bit usati nei calcoli prima della normalizzazione per gestire valori molto grandi.
- **3.1.9** `tb_ignore_start_while_busy`
  Ignora ogni *start* proveniente dalla memoria se il modulo è già in elaborazione, per evitare elaborazioni fantasma. Ha portato alla creazione dello stato IDLE.
- **3.1.10** `tb_addr_high_boundary`
  Scelta di `ADDR` e `K` in modo che l'ultima scrittura sia a `0xFFFF`, garantendo assenza di overflow/wrap.
- **3.1.11** `tb_back_to_back_runs`
  Verifica corretto funzionamento e proprietà di base in due run consecutive senza *re-assert*, anche se non richiesto dalla specifica. Ha portato lo stato DONE a essere rinizializzante.
- **3.1.12** `tb_k_high_stress`
  Sequenze con `K` molto grandi; i risultati hanno portato a definire `cnt_data` su 32 bit (invece di 8) per garantire robustezza anche in condizioni di forte stress.

- **3.1.13** `tb_k_zero`
  Comportamento con `K=0`; non richiesto da specifica ma testato.
- **3.1.14** `tb_final_stress`
  Esegue 6 elaborazioni, vari *reset* intermedi, testa saturazioni e troncamenti, la correttezza dei filtri e utilizza circa **48 KB** della RAM disponibile.

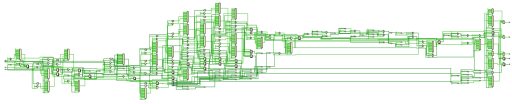
4.0 CONCLUSIONI

4.0.1 Riscontri sperimentali

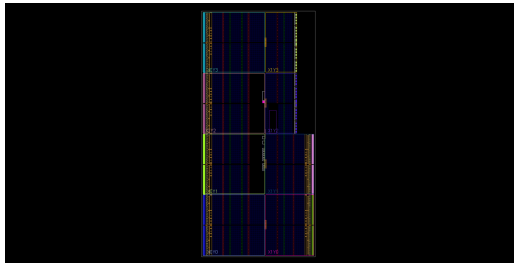
I vari `tb` creati e provati garantiscono una buona copertura degli *edge cases*. Il modulo passa tutti i `tb` proposti sia in pre che post sintesi e risulta **robusto**; tuttavia, come emerso nei test più pesanti, è relativamente **lento** a causa dell'elevato numero di stati. L'abbandono della struttura a componenti, pur preferibile su progetti molto grandi, in questo contesto ha reso la soluzione a FSA integrata più semplice da *debuggare* e migliorare in fase di sviluppo.

4.0.2 Possibilità di miglioramento

Si possono aumentare le **performance** riducendo il numero di stati totale e adottando un flusso meno spezzettato, ovvero unendo più micro-stati in macro-stati. A parte l'aspetto prestazionale, il modulo si dimostra **molto robusto**, capace di superare `tb` non richiesti e di reggere bene dati/situazioni estreme (nei limiti richiesti dalla specifica).



RTL progetto.



Device.

Oleg Nedina
Matr. **225897**
Cod.Persona **10905938**

Il codice e l'elaborato sono stati scritti interamente da me.
Il progetto è stato realizzato fino alla **sintesi comportamentale come richiesto**.
Gli schemi sono stati disegnati a mano da me, tranne le ultime due immagini che sono state ricavate direttamente da Vivado.