POLITECNICO
MILANO 1863

Progetto Prova Finale Reti Logiche

Politecnico di Milano - Docente: William Fornaciari

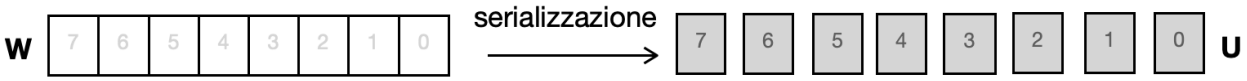
Laura Colazzo - Matricola: 933115

A.A. 2021/2022

1. Introduzione

Obiettivo del progetto di Prova Finale di Reti Logiche è la sintesi di un componente hardware, descritto in VHDL, che presenti le caratteristiche evidenziate nel documento di specifica.
Tali specifiche prevedono che il componente legga sequenze da W byte salvate in una memoria, le elabori e produca un risultato espresso in termini di bytes da scrivere nella stessa memoria.

Il processo di elaborazione prevede la serializzazione dei blocchi da 8 bit in entrata al componente:

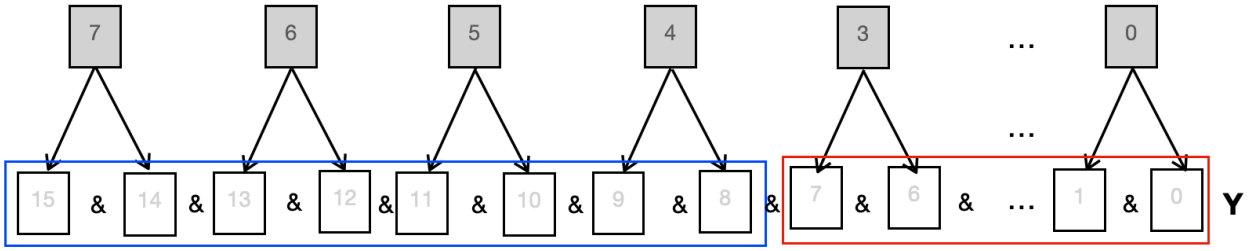


in questo modo viene prodotto un flusso continuo (U) di bit singoli. A ciascuno di essi viene, poi, applicato un codice convoluzionale 1/2. Questo passaggio consiste nel mappare un bit singolo su due bit, secondo una logica descritta nelle specifiche tramite una macchina di Mealy. Tale macchina riceve in ingresso un bit (U_k), frutto del processo di serializzazione e, tenendo conto anche dello stato corrente della FSM, produce due bit in uscita, indicati rispettivamente con i simboli P_{1k} e P_{2k}.
La tabella delle transizioni del modulo convolutore può essere, quindi, rappresentata come di seguito:

Stato corrente		Ingresso $U_k = 0$				Ingresso $U_k = 1$			
Q_1	Q_0	Q_1*	Q_0*	P_{1k}	P_{2k}	Q_1*	Q_0*	P_{1k}	P_{2k}
0	0	0	0	0	0	1	0	1	1
0	1	0	0	1	1	1	0	0	0
1	0	0	1	0	1	1	1	1	0
1	1	0	1	1	0	1	1	0	1

Stato di RESET

Le coppie di bit mandate in output dal convolutore vengono successivamente concatenate l'un l'altra, a formare un flusso (Y) di bit singoli:



A questo punto il flusso Y ottenuto viene parallelizzato su 8 bit a formare una catena di Z byte da scrivere in memoria:

La memoria con cui il componente deve interfacciarsi è una memoria RAM sincrona indirizzata al byte, strutturata in modo tale da contenere all'interno della parola di memoria di indirizzo 0 il numero W di byte da leggere, che da specifica è al più 255. Le parole da effettivamente leggere ed elaborare sono collocate a partire dall'indirizzo 1001 in poi. Quanto prodotto in output dal

componente, invece, deve essere scritto nella stessa memoria a partire dall'indirizzo 1000. Si noti che, secondo la precedente descrizione, per ogni byte letto (ad eccezione del primo) ne vengono scritti due. Quindi, la porzione di memoria preposta alla scrittura risulterà grande il doppio, rispetto a quella preposta alla lettura vera e propria.

Un ulteriore dettaglio riguardo al funzionamento atteso del componente è quello relativo ai segnali di inizio e fine della computazione. In particolare da specifica è richiesto che l'inizio e la fine delle elaborazioni siano sanciti dai due segnali (rispettivamente quello di `start` e quello di `done`). Il segnale di `start` rimane alto durante la computazione e, solo dopo che il segnale di `done` viene alzato, può tornare basso. Tornato basso il segnale di `start`, a questo punto anche quello di `done` può fare lo stesso. Per quanto riguarda il segnale di `reset`, invece, quest'ultimo verrà asserito all'inizio della prima elaborazione, considerando che più elaborazioni consecutive potrebbero essere operate.

I segnali appena citati saranno parte dell'interfaccia del modulo principale, di cui verrà discusso più avanti.

Di seguito viene riportato ,invece, un semplice esempio di funzionamento, completo di stato della memoria prima e dopo l'avvio della computazione.

1.1. Esempio

Supponiamo di avere una memoria inizializzata come di seguito (il contenuto dall'indirizzo 1000 in poi viene riempito soltanto una volta elaborato quanto presente nella porzione di lettura, pertanto prima di dare avvio alla computazione non è di interesse).

Indirizzo	Contenuto
0	0000 0001
1	0010 1001
2	…
…	…
1000	0000 1101
1001	0001 0100
1002	..
…	…

Una volta dato avvio alla computazione il modulo legge la parola di memoria di indirizzo 0. Il suo contenuto è il byte $[0000\ 0001]_2 = [1]_{10}$. Questo suggerisce che il modulo dovrà operare una ulteriore lettura dalla memoria.

La lettura successiva estrarrà il byte $[0010\ 1001]_2$, così che il modulo lo elabori secondo modalità descritte in precedenza:

U_K: 0 0 1 0 | 1 0 0 1

P_{1K}: 0 0 1 0 | 0 0 0 0

P_{2K}: 0 0 1 1 | 0 1 1 0

Y: 00001101 00010100

Dunque, in memoria verranno scritti i due byte $[0000\ 1101]_2$ e $[0001\ 0100]_2$.

2. Architettura

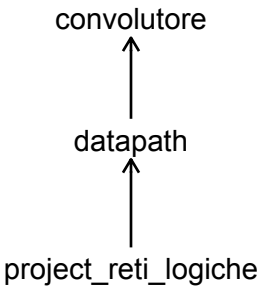
L'approccio utilizzato per la progettazione dell'architettura è stato di tipo ibrido: strutturale, in quanto si compone di un modulo principale che istanzia dei moduli ausiliari e allo stesso tempo comportamentale, in quanto la descrizioni dei componenti all'interno dei moduli è stata realizzata prevalentemente mediante una collezione di process.

Il processo di ideazione ha visto una prima fase di stesura del datapath, costituito da un insieme di componenti fisici (registri, mux, sommatori, …) interconnessi tra loro: una base su cui costruire una macchina a stati, in grado di governare il passaggio dei dati attraverso gli elementi fisici schematizzati.

La struttura del progetto prevede una divisione in tre moduli:

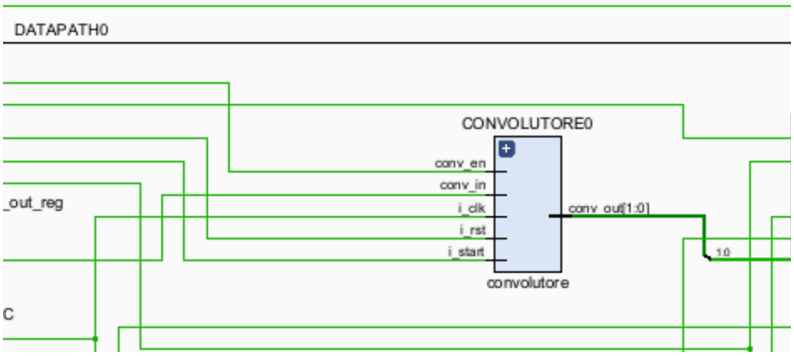
- project_reti_logiche: è il modulo principale. Modella la macchina a stati che fornisce l'intelligenza al componente richiesto da specifica. Istanzia il datapath;
- datapath: si occupa di descrivere i componenti fisici su cui i dati viaggiano e le relative interconnessioni. Istanzia il convolutore, che viene trattato come uno di tali componenti;
- convolutore: modella la macchina a stati che stabilisce la codifica convoluzionale.

La descrizione dell'architettura seguirà l'albero delle dipendenze dei moduli a partire dal modulo radice, per poi proseguire verso il modulo foglia, che di fatti è quello che dipende dai moduli sovrastanti. Si noti che a relazione di dipendenza deve essere letta nel seguente modo: A → B si legge come "A dipende da B".



2.1. Modulo 1: convolutore

Il modulo in questione si occupa dell' applicazione del codice convoluzionale sulle sequenze di bit lette da memoria e serializzate.



Il componente presenta la seguente interfaccia:

```
entity convolutore is
port (
        i_clk :         in std_logic;
        i_rst :         in std_logic;
        i_start :       in std_logic;
        conv_en :       in std_logic;
        conv_in :       in std_logic;
        conv_out :      out std_logic_vector(1 downto 0));
end convolutore;
```

Segue una breve descrizione dei segnali di input e output del componente:

- `i_clk`: è il segnale di clock globale dell'intera architettura;

- `i_rst`: è il segnale di reset globale dell'intera architettura. Una volta asserito riporta la macchina a stati che descrive il comportamento del componente allo stato di reset (codifica stato: "00");

- `i_start`: è il segnale di start globale dell'intera architettura che determina l'inizio di una nuova computazione. Una volta portato a '0' sulla macchina a stati ha lo stesso effetto del segnale di reset;

- `conv_en`: è il segnale di enable del componente, quando posto a '0' rende lo rende insensibile agli ingressi. Il segnale è stato introdotto al fine rendere il modulo inattivo nella fase di transizione tra la scrittura in memoria dei primi due byte e l'inizio dell'elaborazione dei successivi byte da scrivere. Questo permette alla FSM che descrive il comportamento del convolutore di rimanere nell'ultimo stato in cui si era fermata alla fine della precedente elaborazione delle sequenze di byte in ingresso provenienti dalla memoria;

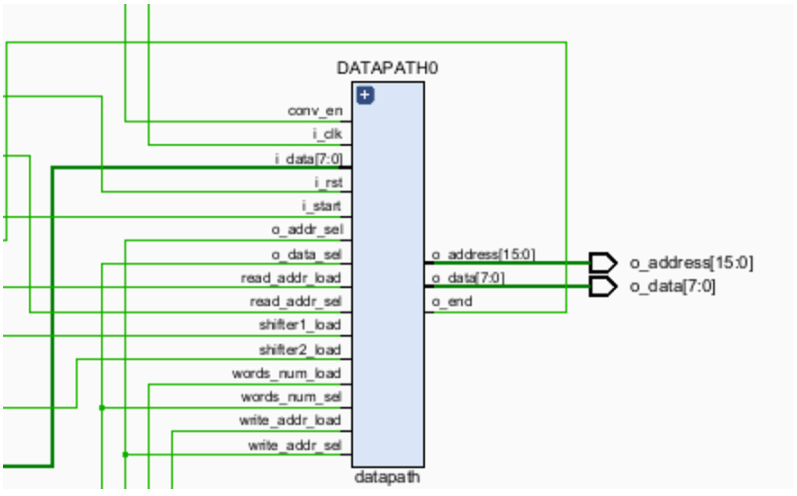
- `conv_in`: è il segnale che trasporta i bit a cui applicare il codice convoluzionale;

- `conv_out`: è il segnale che trasporta in uscita la coppia di bit ottenuta dall'applicazione del codice convoluzionale ai singoli bit di ingresso al componente.

Il comportamento interno del modulo è stato descritto a partire dal disegno della macchina a stati presente nella specifica, mediante l'impiego di due soli process. Trattandosi di una macchina di Mealy, infatti, è stato sufficiente racchiudere la funzione stato prossimo e la funzione d'uscita in un unico process, essendo l'uscita del componente dipendente sia dallo stato corrente, che dall'ingresso alla macchina. Il rimanente process, invece, è quello che si occupa di inferire il numero corretto di flip-flop per tenere traccia dello stato corrente della macchina.

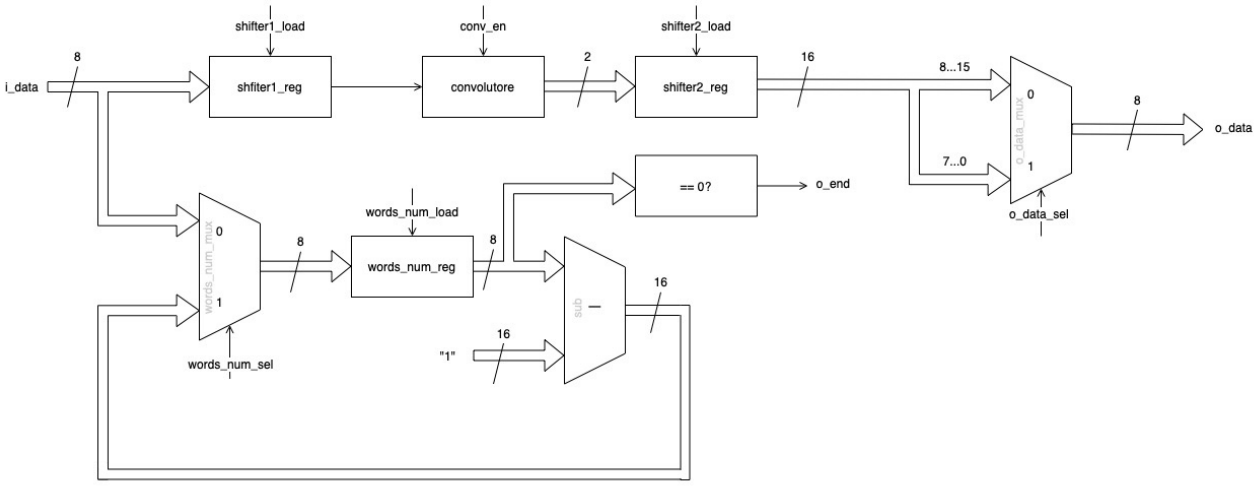
2.2 Modulo 2: datapath

Il modulo in questione descrive una struttura composta da interconnessioni di componenti fisici, all'interno della quale i dati provenienti da memoria viaggiano, subendo durante il loro passaggio alcune trasformazioni.



Lo schematico completo ideato viene riportato nelle pagine successive. Si noti che nel diagramma raffigurato compare anche il modulo convolutore precedentemente descritto, il datapath, infatti, lo istanzia come componente a cui collega dei fili in ingresso e da cui prende degli output.
Per comprendere nel dettaglio la struttura, occorre suddividerla in due porzioni: la prima riguardante la trasformazione dei dati di input vera e propria ed una seconda parte riguardante la gestione degli indirizzi di lettura e scrittura della memoria.

La prima porzione presenta la seguente struttura:



Il primo byte letto da memoria, ovvero quello contenente il numero di parole da leggere, viene inizialmente indirizzato dal “words_num_mux” verso il registro “words_num_reg”, il quale, se il suo segnale di `words_num_load` è alto salva il byte in questione. A partire dalle successive letture dalla memoria e solo dopo aver scritto i byte prodotti in quest’ultima, il contenuto del registro verrà decrementato, fino ad essere portato al valore $[0000\ 0000]_2$. A questo punto il comparatore di uguaglianza, preposto al controllo del fatto che ci siano ancora parole da leggere o meno, noterà la necessità di non dover operare ulteriori letture e asserirà il segnale di `o_end`, che verrà interpretato come segnale di fine della computazione.

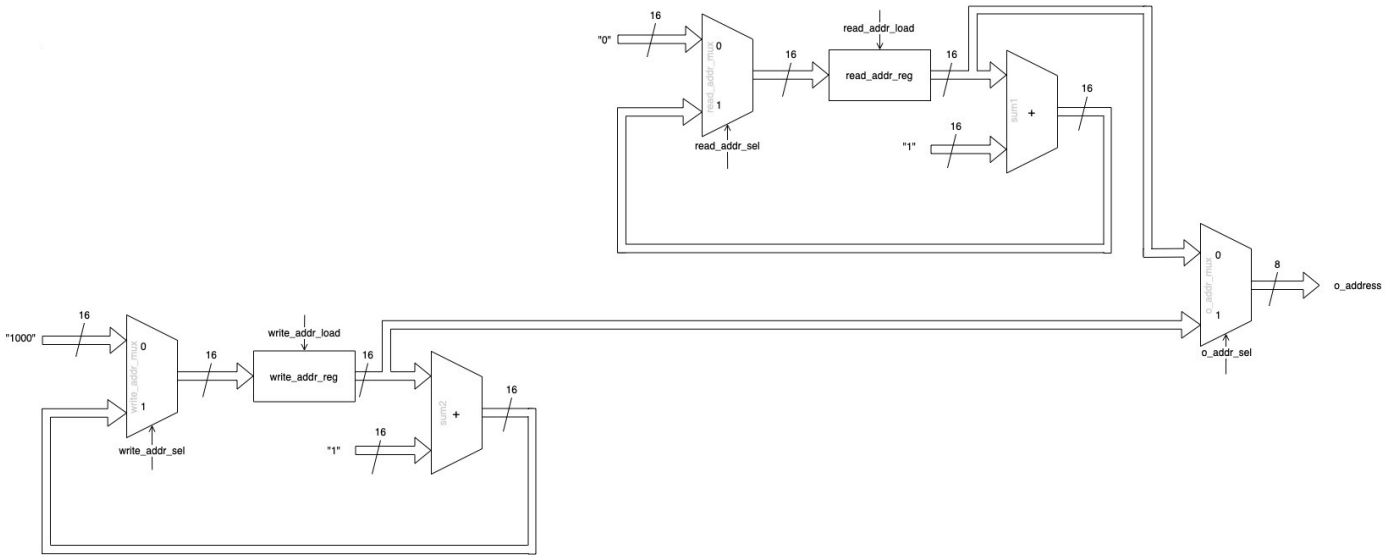
Sempre a partire dalle letture di parole di memoria successive, il flusso `i_data` prenderà un percorso differente: verrà indirizzato verso lo “shifter1_reg": il registro in questione modellizza uno shifter PISO (Parallel-In Serial-Out). Quindi, quando il segnale `shifter1_load` sarà alto, quanto trasportato da `i_data` verrà caricato nel registro, mentre, quando lo stesso segnale sarà basso, un bit (quello più significativo) verrà mandato in output al registro, shiftando quelli rimasti di una posizione a sinistra.

I bit singoli in output dallo shifter ad ogni ciclo di clock verranno mandati in input al modulo convolutore, che dopo avervi applicato il codice convoluzionale, manderà in output una coppia di bit (affinché questo avvenga il segnale di `conv_en` deve essere alto in questa fase).

Le coppie di bit in uscita dal convolutore vengono caricate in un ulteriore shifter (“shifter2_reg”), che, nel momento in cui il suo segnale di load è alto, carica in blocco due bit in posizione meno significativa e opera uno shift verso sinistra di due posizioni su ciascuno dei 16 bit del registro.

Finita la serializzazione di un byte letto da memoria da parte dello “shifter1_reg”, lo “shifter2_reg” conterrà i due byte da scrivere in memoria. L’operazione di scrittura consisterà nel prelevare prima la prima porzione (quella più significativa) da 8 bit del contenuto dello “shifter2_reg” e poi la rimanente (quella meno significativa), in modo da riportarle in memoria in due scritture distinte successive.

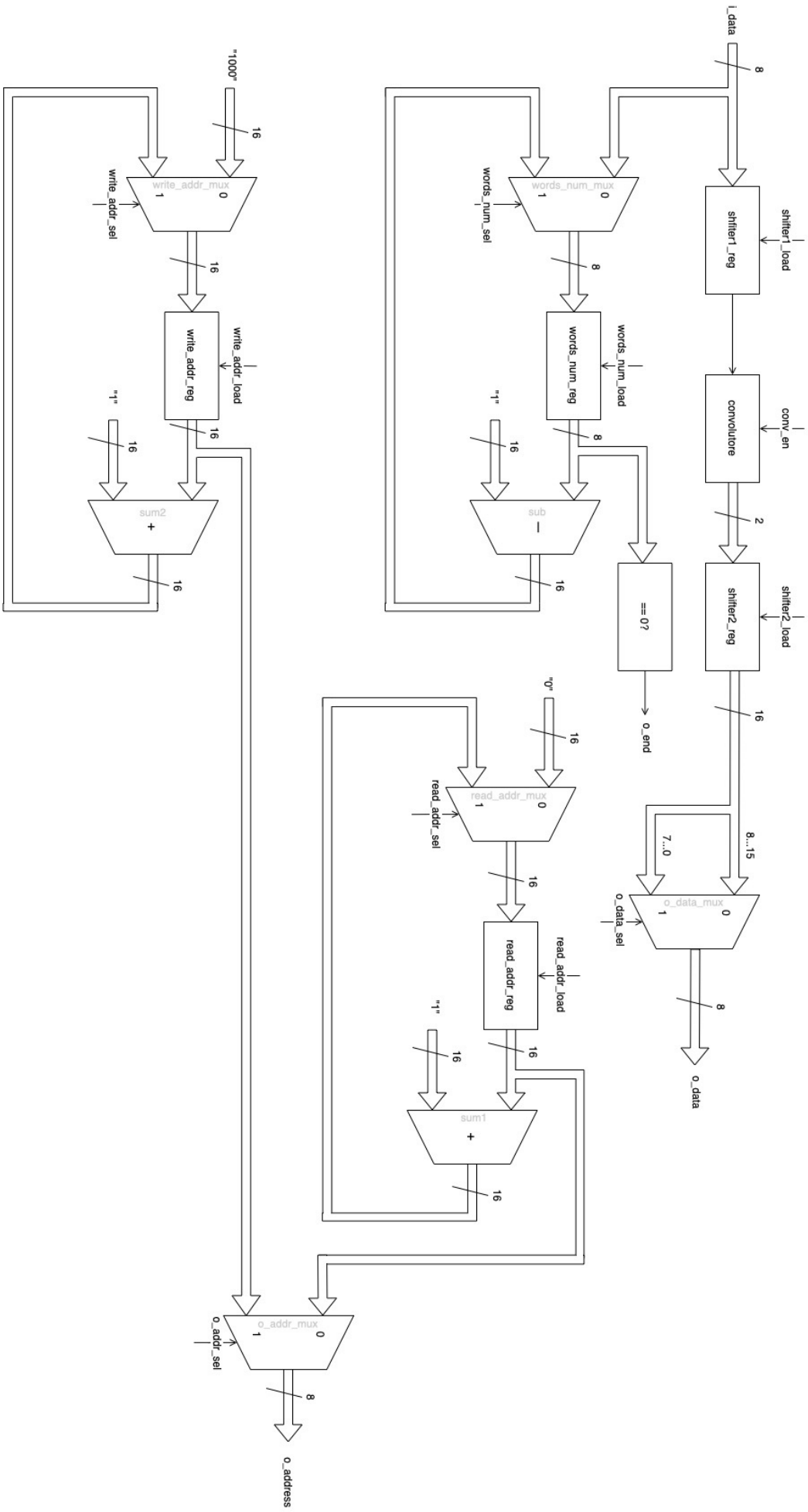
La seconda porzione del datapath è quella che determina l’indirizzo di memoria da cui leggere o scrivere e che gestisce l’incremento di tali indirizzi.



La struttura presenta due registri preposti al salvataggio degli indirizzi (uno per la lettura e uno per la scrittura), entrambi aventi come ingresso l'uscita di un proprio multiplexer. La presenza di ciascuno dei multiplexer è necessaria poiché, se il segnale di selezione è basso e il segnale di load dei registri alto, il contenuto dei registri viene ripristinato ai valori $[1000]_{10}$ e $[0]_{10}$, rispettivamente $[0000001111101000]_2$ e $[0000000000000000]_2$. In questo modo, nel caso di computazioni multiple consecutive, la memoria può tornare ad essere letta e scritta dai primi indirizzi a disposizione.

L'incremento dei registri avviene combinando l'effetto del sommatore a valle di ciascun registro, con l'effetto della selezione dell' ingresso '1' del multiplexer associato a ciascun registro ed, infine, del segnale di load del registro stesso. Così facendo il precedente valore memorizzato nei 16 bit messi a disposizione viene sovrascritto con il nuovo valore (uguale al vecchio, ma incrementato di 1 bit).

Infine, la selezione di quale indirizzo mandare in output al modulo (se quello di lettura o di scrittura), affinché questo venga utilizzato per interagire con la memoria, avviene attraverso il multiplexer "o_addr_mux" a valle di tutta lo schema: se il suo segnale di selezione è '0', allora in output si ha l'indirizzo di lettura, altrimenti se il segnale è '1', in uscita si ottiene l'indirizzo di scrittura.


i_data
8
shifter1_reg
shifter1_load
convolutore
conv_en
2
shifter2_reg
shifter2_load
16
8..15
7..0
o_data_mux
1
0
o_data_sel
8
o_data
words_num_mux
words_num_sel
words_num_reg
words_num_load
sub
"1"
== 0?
o_end
"0"
read_addr_mux
read_addr_sel
read_addr_reg
read_addr_load
sum1
"1"
"1000"
write_addr_mux
write_addr_sel
write_addr_reg
write_addr_load
sum2
"1"
o_addr_mux
o_addr_sel
o_address

Il datapath presenta la seguente interfaccia:

```
entity datapath is
port(
      i_clk:          in std_logic;
      i_start:        in std_logic;
      i_rst:          in std_logic;
      i_data:         in std_logic_vector(7 downto 0);
      conv_en:        in std_logic;
      words_num_load: in std_logic;
      words_num_sel:  in std_logic;
      shifter1_load:  in std_logic;
      shifter2_load:  in std_logic;
      read_addr_load: in std_logic;
      read_addr_sel:  in std_logic;
      write_addr_load:in std_logic;
      write_addr_sel: in std_logic;
      o_data_sel:     in std_logic;
      o_addr_sel:     in std_logic;
      o_end:          out std_logic;
      o_data:         out std_logic_vector(7 downto 0);
      o_address:      out std_logic_vector(15 downto 0)
);
end entity;
```

Descrizione dei segnali di interfaccia:

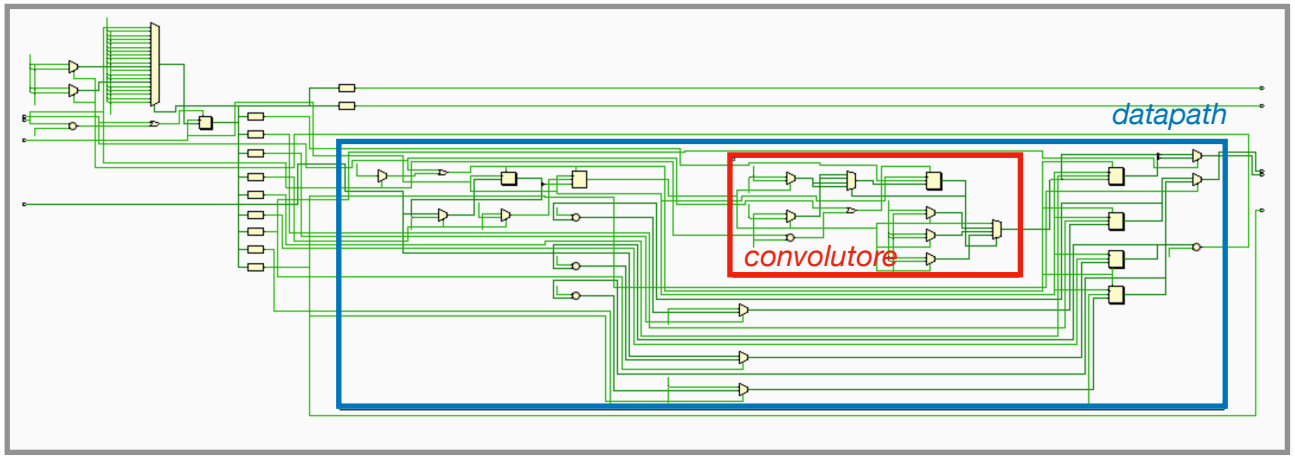
- `i_clk`: è il segnale di clock globale dell'intera architettura;
- `i_start`: è il segnale di start globale dell'intera architettura, determina l'inizio di una nuova computazione. Una volta portato a '0', attiva il circuito di reset dei vari process che compongono il datapath;
- `i_rst`: è il segnale di reset globale dell'intera architettura. Una volta asserito abilita i circuiti di reset dei vari process he compongono il datapath;
- `i_data`: è il segnale che trasporta i byte letti da memoria in entrata al componente;
- `conv_en`: è il segnale di enable da dare in ingresso al modulo convolutore;
- `words_num_load`: è il segnale di caricamento del registro "words_num_reg";
- `words_num_sel`: è il segnale di selezione del mux "words_num_mux";
- `shifter1_load`: è il segnale di caricamento del registro "shifter1_reg";
- `shifter2_load`: è il segnale di caricamento del registro "shifter2_reg";
- `read_addr_load`: è il segnale di caricamento del registro "read_addr_reg";
- `read_addr_sel`: è il segnale di selezione del mux "read_addr_mux";

- `write_addr_load`: è il segnale di caricamento del registro “write_addr_reg”;
- `write_addr_sel`: è il segnale di selezione del mux “write_addr_mux”;
- `o_data_sel`: è il segnale di selezione del mux “o_data_mux”;
- `o_addr_sel`: è il segnale di selezione del mux “o_addr_mux”;
- `o_end`: è il segnale che viene dato in output al modulo che stabilisce se la computazione sia finita o meno;
- `o_data`: trasporta i byte da mandare in lettura nella memoria;
- `o_addr`: trasporta gli indirizzi da utilizzare per eseguire le operazioni in memoria;

L’intero modulo è stati costruito combinando logica combinatoria a logica sequenziale, quest’ultima descritta in VHDL attraverso una collezione di process: uno per ogni componente dotato di memoria.

2.3 Modulo 3: project_reti_logiche

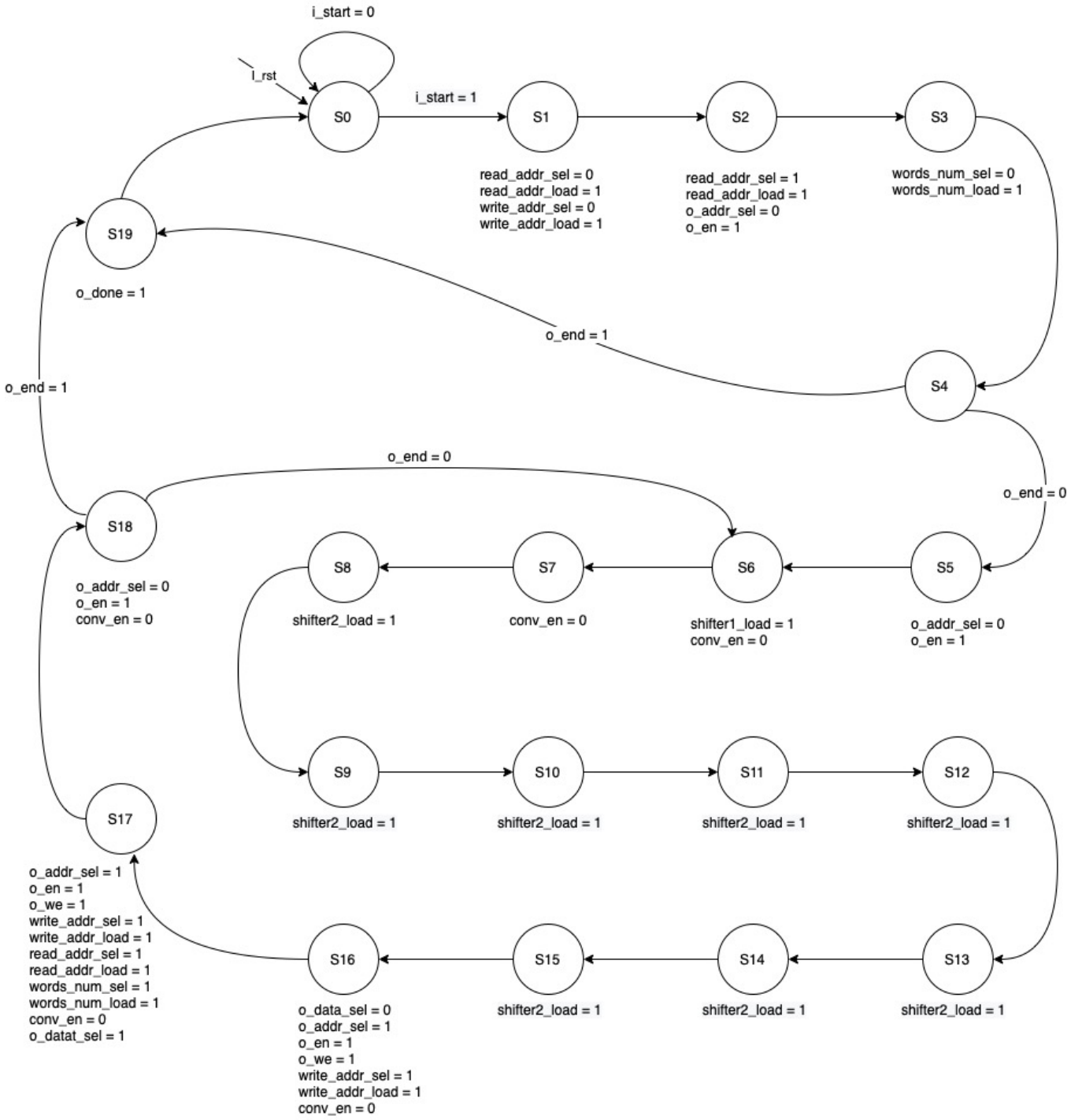
project_reti_logiche



Il modulo in questione è il modulo principale dell’intero design, la cui interfaccia è la stessa presente nel documento di specifica. Al suo interno racchiude i due i moduli precedentemente descritti, così come evidenzia la figura, in cui è presente un estratto del design elaborato dal tool di sintesi (approfondito più avanti). La logica sequenziale interna al componente è stata ricavata a partire dal design concettuale di una macchina a stati (macchina di Moore), sviluppata in una fase precedente a quella di implementazione.

La struttura della FSM in questione viene riportata nelle pagine successive. Vengono, invece, ora descritti i singoli stati che la compongono:

- `S0`:
  è lo stato in cui, in caso di computazioni consecutive, si riporta basso il segnale di `o_done` e si aspetta che il segnale relativo all’inizio di una nuova computazione (`i_start`) venga asserito, per poter avanzare verso lo stato `S1`. Inoltre, qualunque sia lo stato corrente della macchina, una volta asserito il segnale di `i_rst` avviene immediatamente la transizione verso questo stato (`S0`) (è lo stesso effetto ha il segnale di `i_start` quando è basso);



- `S1:`
  a questo punto si predispongono i due registri preposti al salvataggio degli indirizzi di lettura e scrittura da memoria, affinché il loro contenuto venga inizializzato rispettivamente ai valori $[0]_{10}$ e $[1000]_{10}$ (valori da convertire in binario);

- `S2:`
  si preleva l'indirizzo di lettura dal registro precedentemente inizializzato al valore $[0]_{10}$ e lo si utilizza per richiedere una lettura in memoria;

- `S3`:
  a seguito della precedente richiesta di lettura, in questo stato il dato richiesto diventa disponibile e viene caricato nel registro preposto al salvataggio del numero di parole da leggere da memoria;

- `S4`:
  in questo stato si verifica se il numero di parole da leggere sia nullo e in caso affermativo si passa allo stato `S19`; altrimenti si procede nello stato `S5`;

- `S5`:
  giunti in questo punto si procede ad una nuova richiesta di lettura da memoria;

- `S6`:
  il dato precedentemente richiesto diventa disponibile e viene caricato nel registro che, a partire dallo stato successivo, metterà in atto il processo di serializzazione. In questo stesso stato il convolutore viene momentaneamente disabilito, portando basso il suo segnale di `conv_en`, affinché i bit giunti al suo ingresso non vengano interpretati come input validi a cui applicare il codice convoluzionale. Questa accortezza è irrilevante durante l'elaborazione dei primi due byte da scrivere in memoria, ma diventa cruciale nel caso di elaborazioni successive ulteriori;

- `S7`:
  in questo stato al registro contenente il byte letto da memoria viene impartito il primo comando di shift, affinché il bit più significativo della sequenza salvata venga mandato in output al registro e il resto dei bit trasli a sua volta di una posizione verso sinistra. Si noti che il segnale di `conv_en` è ancora momentaneamente abbassato;

- `S8`:
  arrivati a questo punto si continua con un ulteriore comando di shift. Il bit prodotto dallo shift impartito nel precedente stato, invece, arriva all'ingresso del convolutore, che viene pertanto riabilitato, producendo due bit come risultato dell'applicazione del codice convoluzionale. Sempre in questo stato i due bit in questione vengono caricati in un secondo shifter, a cui viene dato un segnale di shift verso sx, così che la coppia appena caricata faccia posto alla prossima coppia in posizione meno significativa del registro;

- `S9 -> S15`
  questa sequenza di stati non è di particolare rilevanza, in quanto ciascuno degli stati che la compone, replica in maniera identica quanto svolto nello stato `S8`. Il motivo di tale forma di ripetizione risiede nel fatto che la serializzazione deve essere operata fino a terminare gli 8 bit salvati nel primo shifter e, allo stesso tempo, il secondo shifter (da 16 bit) deve continuare ad accogliere coppie di bit, fino a che il convolutore non termina di produrre risultati;

- `S16`:
  in questo stato si congela il contenuto del secondo shifter, si preleva la sua porzione da 8 bit più significativa e la si scrive in memoria, dopo aver selezionato come indirizzo quello proveniente dal registro degli indirizzi di scrittura. Si procede, inoltre, ad incrementare il contenuto di tale registro, affinché la scrittura della porzione rimanente da 8 bit avvenga nel successivo stato all'indirizzo contiguo. Infine, viene anche disabilitato il convolutore, affinché il suo stato corrente venga conservato fino alla prossima elaborazione;

- `S17`:
  a questo punto è possibile procedere con la scrittura in memoria della seconda porzione del registro citato in precedenza, selezionando come indirizzo di memoria quello appena incrementato. Si procede, inoltre, con l'incremento del contenuto dei registri relativi agli indirizzi di lettura e scrittura, per predisporre il componente a letture e scritture successive. Si decrementa, invece, il contenuto del registro con il numero di parole ancora da leggere;

- `S18`:

In questo stato si procede con la verifica che il numero di parole da leggere non sia diventato nullo e procedendo con una richiesta di lettura da memoria, per anticipare l'eventuale necessità di leggere ulteriori parole da memoria. Se non ci sono più parole da leggere, si procede verso lo stato `S19`, altrimenti si ricomincia dallo stato `S6`;

- `S19`:
  prima di riportare la macchina nello stato iniziale `S0`, si alza il segnale di `o_done` per sancire la fine della computazione della macchina.

Il modulo in VHDL presenta la seguente interfaccia:

```
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;
```

Segue una breve descrizione dei segnali, rielaborata a partire dal documento di specifica:

- `i_clk`: è il segnale di clock in input al componente;
- `i_rst`: è il segnale di reset in input al modulo. Una volta asserito abilita il circuito di reset della FSM, riportato la macchina allo stato `S0` di reset;
- `i_start`: è il segnale di start impartito al componente: determina l'avvio alla computazione. Una volta portato basso ha lo stesso effetto del segnale di reset;
- `i_data`: trasporta un byte proveniente da memoria;
- `o_address`: è il segnale fornito in output dal componente che trasporta l'indirizzo con cui leggere/scrivere in memoria;
- `o_done`: è il segnale mandato in uscita per comunicare la fine dell'elaborazione;
- `o_en`: è il segnale di enable da mandare in output verso alla memoria: Se è pari ad '1' una connessione con la memoria può essere stabilita;
- `o_we`: è il segnale che viene fornito alla memoria, al fine di specificare il tipo di operazione da effettuare ('1' per la scrittura, '0' per la lettura);
- `o_data`: è il byte mandato in uscita dal componente, da dare in input alla memoria affinché venga scritto.

Il funzionamento interno del modulo è stato descritto attraverso tre process: uno per la funzione stato prossimo, una per la funzione d'uscita e uno per la gestione dello stato corrente della macchina.

3. Risultati sperimentali

Il software utilizzato per la sintesi del componente è stato Vivado (Xilinx), utilizzando come FPGA target la Artix-7 FPGA xc7a200tfbg484-1.

3.1.1. Report di sintesi: utilizzo FPGA

Di seguito viene mostrato il report relativo al numero e al tipo di componenti inferiti dal tool a seguito della fase di sintesi:

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs*	85	0	0	134600	0.06
LUT as Logic	85	0	0	134600	0.06
LUT as Memory	0	0	0	46200	0.00
Slice Registers	87	0	0	269200	0.03
Register as Flip Flop	87	0	0	269200	0.03
Register as Latch	0	0	0	269200	0.00
F7 Muxes	0	0	0	67300	0.00
F8 Muxes	0	0	0	33650	0.00

Dai dati evidenziati si evince l'assenza di latch: questo ha garantito il corretto funzionamento del modulo anche in post-synthesis.

Un altro dato rilevante ricavato dal report in alto è quello relativo al numero di componenti inferiti: il numero di FF e LUT impiegato è risultato ben al di sotto il limite fisico imposto dall'FPGA (87 FF utilizzati a fronte dei 269200 disponibili e 85 LUT utilizzati a fronte dei 134600 disponibili).

3.1.2. Report di sintesi: tempistiche

Dopo la fase di sintesi è stato possibile consultare un ulteriore report, relativo alle tempistiche del componente in termini di periodo di clock.

```
Timing Report

Slack (MET) :             96.997ns  (required time - arrival time)
  Source:                 FSM_onehot_cur_state_reg[13]/C
                            (rising edge-triggered cell FDCE clocked by clock  {rise@0.000ns fall@50.000ns period=100.000ns})
  Destination:            DATAPATH0/shifter2_out_reg[0]/CE
                            (rising edge-triggered cell FDCE clocked by clock  {rise@0.000ns fall@50.000ns period=100.000ns})
  Path Group:             clock
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            100.000ns  (clock rise@100.000ns - clock rise@0.000ns)
  Data Path Delay:        2.621ns  (logic 0.875ns (33.384%)  route 1.746ns (66.616%))
```

Nel report viene messo in evidenza il fatto che il vincolo di periodo di clock pari a 100ns imposto dalle specifiche sia ampiamente rispettato, in quanto lo slack è di 96.997 ns. Questo significa che, se il vincolo in questione fosse abbassato di 96.997 ns, il componente sintetizzato sarebbe ancora in grado rispettare il vincolo, essendo il "Data Path Delay" di soli 2.621 ns.

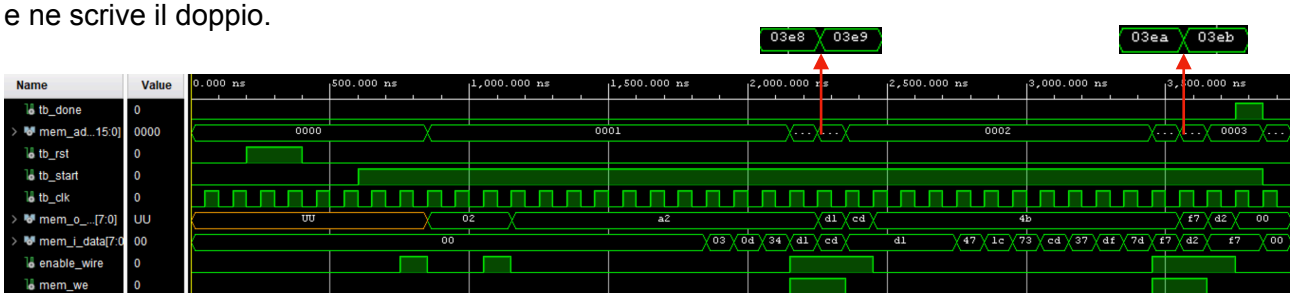
3.2. Simulazioni

Il componente è stato sottoposto ad un pool di simulazioni in Vivado, sia prima della sintesi (attraverso la funzione "Behavioural simulation"), sia a seguito della sintesi (tramite la funzione "Post-synthesis Functional Simulation").
In particolare, le simulazioni svolte sono state ideate al fine di sollecitare il componente sia in casi di funzionamento standard, sia in casi limite.

Nella prossima sezione verranno analizzati i dettagli di ciascun caso preso in considerazione. A tal proposito occorre sottolineare come i test utilizzati siano stati ripetuti in maniera identica sia prima che dopo la sintesi, con il fine di verificare che il comportamento del componente fosse rimasto invariato dopo questo processo.

3.2.1. Test 1: Computazione singola

La simulazione in questione è stata eseguita a partire dal testbench d'esempio fornito assieme alle specifiche. Il caso analizzato è il caso di una semplice computazione che legge da memoria 2 byte e ne scrive il doppio.



La waveform prodotta dalla simulazione mostra come al modulo sia dato prima un segnale di reset (`tb_rst`) e poi uno di start (`tb_start`). Il segnale di start avvia la computazione e seguono, poi, i seguenti eventi:

1. viene alzato il segnale di `enable_wire` per operare la prima lettura in memoria all'indirizzo 0x0000;

2. nel ciclo di clock successivo il segnale `mem_i_data` trasporta il byte 0x02, per cui si può notare come un segnale di `enable_wire` alto verrà combinato con un segnale di `mem_we` basso solo altre due volte (è questa la configurazione di segnali a rappresentare la richiesta di lettura da memoria);

3. la prima lettura all'indirizzo 0x0001 produce il byte 0xa2 (162_{10}), che, una volta elaborato, viene trasformato nei byte 0xd1 e 0xcd. Tali byte vengono scritti in memoria rispettivamente agli indirizzi 0x03e8 e 0x03e9 (1000_{10} e 1001_{10}). La scrittura avviene laddove nella waveform appaiono i due segnali di `enable_wire` e di `mem_we` contemporaneamente alti;

4. questa duplice scrittura viene seguita da una nuova richiesta di lettura, ottenuta abbassando il segnale di `mem_we` e mantenendo alto quello di `enable_wire`;

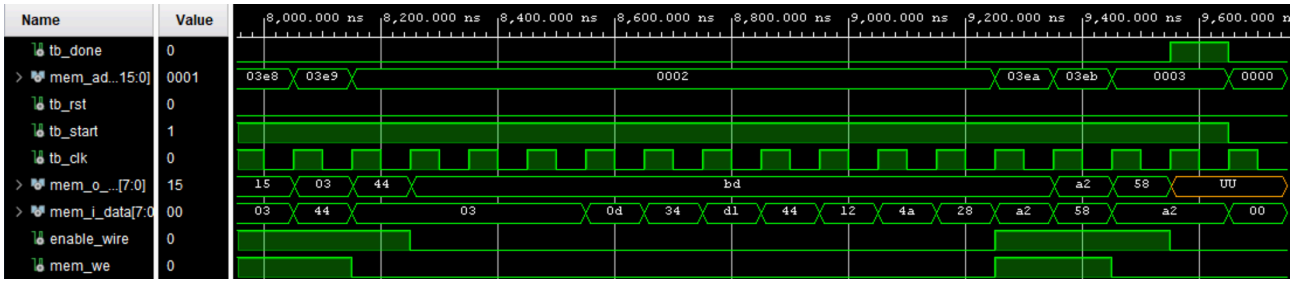
5. la lettura produce il byte 0x4b (72_{10}), che a seguito di successive elaborazioni si trasforma nei byte 0xf7 e 0xd2 da scrivere in memoria agli indirizzi 0x03ea (1002_{10}) e 0x03eb (1003_{10});

6. dopo un ciclo di clock, necessario per eseguire alcune elaborazioni interne, viene alzato il segnale di `o_done` a simboleggiare la fine della computazione.

3.2.2. Test 2: Computazioni multiple

Il caso di test analizzato è un'estensione del precedente: il modulo viene, quindi, sottoposto a computazioni multiple consecutive come la precedente, visto che le specifiche richiedono che il componente supporti anche questa casistica.

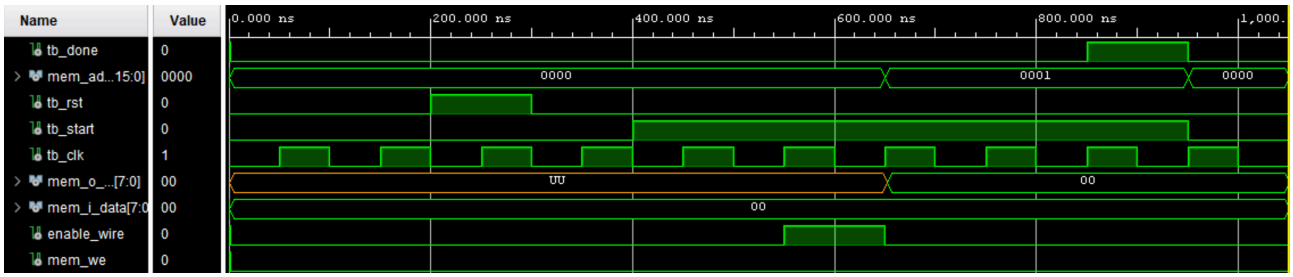
La prima computazione legge 1 byte da memoria e ne scrive 2, la seconda ne legge 2 e ne scrive 4 e la terza ne legge sempre 2 da memoria e ne scrive 4. L'intera computazione viene riportata nelle tre waveform successive:



Rispetto al caso precedentemente analizzato risulta interessante la transizione da una elaborazione alla successiva. Come è possibile constatare dalla waveform delle tre computazioni, finchè il segnale di done non viene alzato, il segnale di start non viene abbassato. Quando, poi, questo accade realmente, l'indirizzo con cui accedere alla memoria viene resettato al valore assunto all'inizio della computazione precedente e rimane tale fino all'avvio di quella nuova. A questo punto l'elaborazione successiva potrà essere svolta nelle stesse condizioni iniziali della precedente e con le stesse modalità.

3.2.3. Test 3: Sequenza di byte di lunghezza nulla

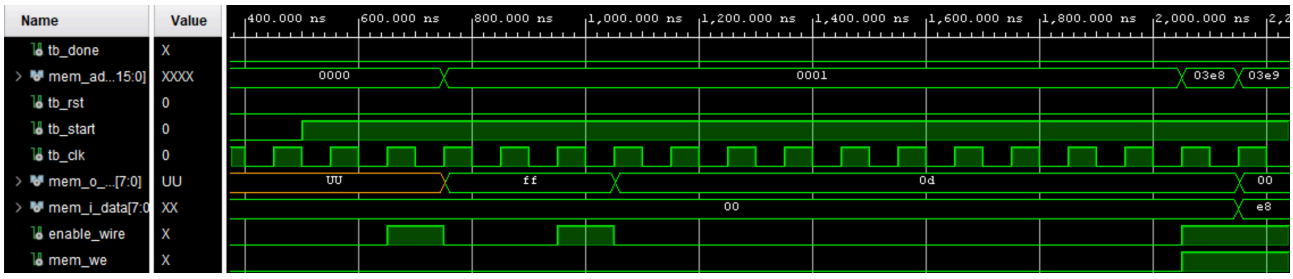
Il terzo caso oggetto di analisi prevede che la memoria sia stata inizializzata con il contenuto dell'indirizzo 0_{10} pari a 0_{10}. Questo significa che nessuna parola dovrà essere letta da memoria (ovviamente fatta eccezione per la prima parola di memoria contenente il numero di parole da leggere, che abbiamo detto essere nullo).



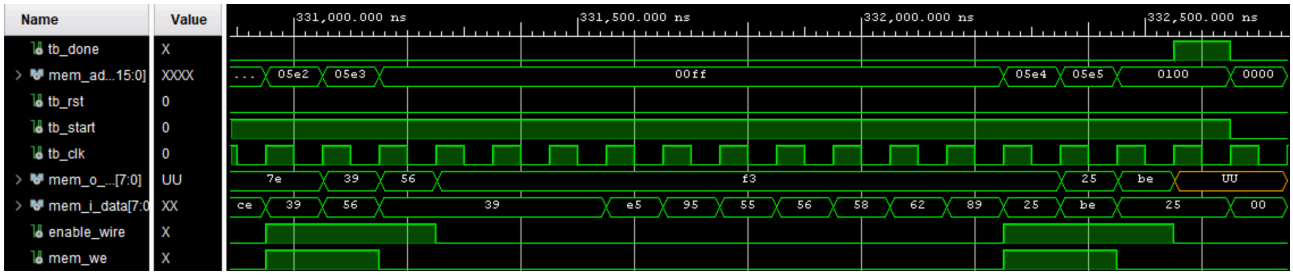
Dalla waveform della simulazione è possibile notare come, una volta alzato il segnale di start, venga effettuata un'unica lettura in memoria all'indirizzo 0x0000. Al ciclo di clock successivo tale lettura produce il byte 0x00: questo suggerisce l'assenza di ulteriori byte da dover leggere dalla memoria. Per questo motivo, trascorso il tempo necessario a delle verifiche interne al componente, il segnale di done viene portato alto, con lo scopo di porre fine all'elaborazione in corso.

3.2.3. Test 4: Sequenza di byte di lunghezza massima

La quarta simulazione messa in atto prevede che la sequenze di byte da leggere sia di lunghezza massima, cioè che sia composta da 255 byte.



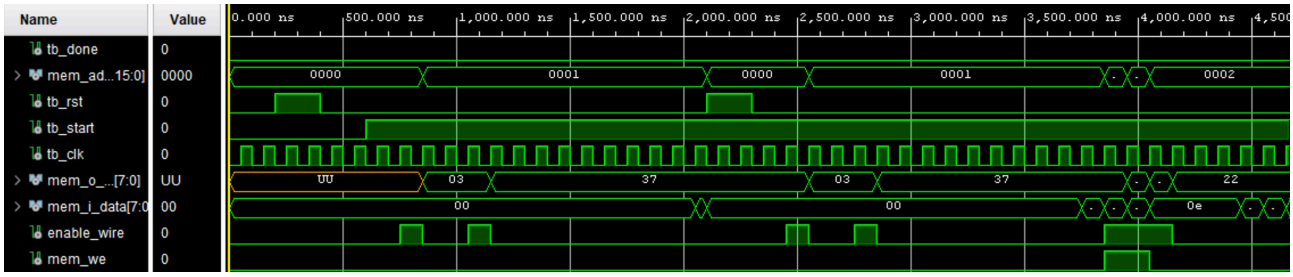
La waveform mostra come la prima lettura da memoria (che avviene quando `enable_wire` si alza per la prima volta) produca il byte 0xff (255_{10}). Seguono, dunque 255 letture da memoria, al termine delle quali la situazione è quella riportata nella waveform successiva:



L'ultima scrittura in memoria avviene in corrispondenza dell'indirizzo 0x05e5 (1509_{10}). Questo viene giustificato dal fatto che 255 letture vengono mappate su 255*2=510 scritture, che avvengono a partire a partire dall'indirizzo 0x03e8 (1000_{10}) incluso.

3.2.3. Test 5: Reset asincrono

L'ultimo caso di test evidenziato riguarda la situazione in cui durante una computazione (mentre il segnale di start è alto), venga asserito in maniera asincrona il segnale di reset. Il comportamento atteso in queste circostanze è che il modulo attivi i circuiti interni di reset e si riporti nelle condizioni iniziali, antecedenti all'inizio dell'elaborazione.



La waveform prodotta dalla simulazione mostra come, in corrispondenza del segnale di reset asincrono, l'indirizzo di memoria utilizzato per eseguire operazioni in memoria (`mem_address`), venga resettato al valore 0x0000 (nell'istante precedente al reset era posto a 0x0001). Allo stesso tempo l'evento di reset ha effetto anche sullo stato dell'elaborazione: quest'ultima riparte da capo, come se il segnale di start fosse stato appena portato alto.

4. Conclusioni

Alla luce delle precedenti simulazioni messe in atto, affiancate da simulazioni con sequenze casuali di parole da leggere lunghe fino a 10 000 byte, è stato, dunque, possibile affermare il corretto funzionamento del componente.

Il modulo, infatti, ha superato tutti i test che gli sono stati sottoposti, sia in fase pre-sintesi, che dopo la sintesi e non sono state evidenziate alcun tipo di differenze tra il comportamento in una fase e rispetto a quello nell'altra.

Inoltre, dai report prodotti dal tool di sintesi (analizzati in precedenza) è stato possibile confermare l'adeguatezza del design prodotto rispetto ai vincoli imposti.

Pensando a possibili spunti di ottimizzazione del componente, si sarebbe sicuramente potuto ridurre significativamente il numero di stati nella FSM del modulo principale (il numero di stati attuale è 20). Dall'analisi della macchina, infatti, era stata precedentemente evidenziata la presenza di 8 stati del tutto identici, seppur funzionali all' intero design. Introducendo nel modulo datapath un contatore opportunamente gestito si sarebbe potuto ovviare al problema della replicazione degli stati, anche se a costo di degradare (in maniera minimale) le prestazioni temporali.