Prova Finale
Progetto di Reti Logiche

Settembre 2023

C. Piccoli

POLITECNICO
MILANO 1863

Indice

1 Introduzione **2**
- 1.1 Scopo del progetto . . . 2
- 1.2 Specifiche generali . . . 2
- 1.3 Interfaccia del componente . . . 2
- 1.4 Dati e gestione della memoria . . . 3

2 Architettura **4**
- 2.1 Stati della FSM . . . 4
  - 2.1.1 IDLE . . . 4
  - 2.1.2 INPUT_READ . . . 4
  - 2.1.3 MEM_LOAD . . . 4
  - 2.1.4 MEM_READ . . . 5
  - 2.1.5 WRITE_OUT . . . 5
  - 2.1.6 RESET . . . 5
- 2.2 Datapath . . . 5

3 Risultati Sperimentali **7**
- 3.1 Simulazioni Behavioral . . . 7
  - 3.1.1 Testbench 1 . . . 7
  - 3.1.2 Testbench 2 . . . 7
  - 3.1.3 Testbench 3 . . . 8
  - 3.1.4 Testbench 4 . . . 8
  - 3.1.5 Testbench 5 . . . 9
  - 3.1.6 Testbench 6 . . . 9
  - 3.1.7 Testbench 7 . . . 10
- 3.2 Simulazioni Post-Synthesis Functional . . . 10
  - 3.2.1 Report di Sintesi . . . 11

4 Conclusioni **12**

1 Introduzione

1.1 Scopo del progetto

Lo scopo del progetto è di implementare un componente hardware descritto in VHDL che, data una stringa binaria *i_w* in ingresso, passata bit per bit, restituisca in output un valore ottenuto dalla memoria.

I bit della sequenza di ingresso vengono letti solo quando il segnale START è alto e termina quando il segnale START è basso.

Nello stesso ciclo di clock in cui il segnale *o_done* è alto, i quattro segnali di output (inizialmente a zero) mostreranno gli ultimi valori salvati nei rispettivi registri, mentre nei restanti cicli le uscite saranno tutte pari a zero.

Tutti i bit su W vengono letti sul fronte di salita del clock. Il tempo massimo per produrre il risultato deve essere inferiore ai 20 cicli di clock.

1.2 Specifiche generali

Il modulo da implementare ha due ingressi primari da 1 bit (W e START) e 5 uscite primarie.

Le uscite sono le seguenti: quattro da 8 bit (Z0, Z1, Z2, Z3) e una da 1 bit (DONE).

Inoltre, il modulo ha un segnale di clock CLK, unico per tutto il sistema e un segnale di reset RESET anch'esso unico.

1.3 Interfaccia del componente

Il componente da descrivere possiede 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_w     : in std_logic;
        o_z0    : out std_logic_vector(7 downto 0);
        o_z1    : out std_logic_vector(7 downto 0);
        o_z2    : out std_logic_vector(7 downto 0);
        o_z3    : out std_logic_vector(7 downto 0);
        o_done  : out std_logic;
        o_mem_addr : out std_logic_vector(15 downto 0);
        i_mem_data : in std_logic_vector(7 downto 0);
        o_mem_we   : out std_logic;
        o_mem_en   : out std_logic
    );
end project_reti_logiche;
```

In particolare:

- **i_clk** è il segnale di CLOCK in ingresso generato dal Test Bench;
- **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 Test Bench;
- **i_w** è il segnale W precedentemente descritto e generato dal Test Bench;
- **o_z0, o_z1, o_z2 e o_z3** sono i quattro canali di uscita:
- **o_done** è il segnale di uscita che comunica la fine dell'elaborazione;
- **o_mem_addr** è il segnale (vettore) di uscita che manda l'indirizzo alla memoria;
- **i_mem_data** è il segnale (vettore) che arriva dalla memoria in seguito ad una richiesta di lettura;
- **o_mem_en** è il segnale di ENABLE da dover mandare alla memoria per poter comunicare (sia in lettura che in scrittura);
- **o_mem_we** è il segnale di WRITE ENABLE da dover mandare alla memoria (=1) per poter scriverci. Per leggere da memoria esso deve essere 0.
  - NB: Poichè non è richiesto scrivere in memoria, il segnale di write enable viene sempre lasciato a zero durante il funzionamento di questo componente.

1.4 Dati e gestione della memoria

I dati vengono passati al componente bit per bit tramite il segnale *i_w*: quando il segnale START è alto possono passare dai 2 ai 18 bit in ingresso.

In particolare i primi due bit passati in input identificano il canale di uscita e i restanti identificano l'indirizzo di memoria da cui recuperare il valore. Nel caso in cui vengano passati meno di 18 bit, i bit successivi al secondo, che identificano l'indirizzo di memoria, vengono estesi con 0 sui bit più significativi.

Il modulo accede ad un indirizzo di memoria da 16 bit tramite il segnale *o_mem_addr*.

Dalla memoria viene restituito un valore da 8 bit tramite il segnale *i_mem_data*.

2 Architettura

Il componente progettato è composto da:

- **Macchina a Stati**: si occupa della parte algoritmica del progetto che comprende 4 fasi principali
  - Reset
  - Lettura dei dati in input
  - Accesso alla memoria ed elaborazione dei dati
  - Visualizzaione dei dati sugli output
- **Datapath**: è il componente che si occupa dell'inizializzazione e dell'aggiornamento dei registri.

2.1 Stati della FSM

Ad ogni ciclo di clock vengono resettati a 0 i seguenti segnali:

- *o_done*
- *dsel_load*
- *output_load*
- *o_mem_we*
- *o_mem_en*
- *load_to_mem*
- *load_from_mem*

2.1.1 IDLE

Stato iniziale in cui si attende che il segnale *i_start* venga portato a 1.

Nel ciclo successivo a quello in cui *o_done* è alto, si torna in questo stato.

Nel ciclo successivo a quello in cui *i_reset* è alto, si torna in questo stato.

2.1.2 INPUT_READ

Stato in cui viene letto l'input e viene salvato nei registri.

Il componente resta in questo stato finchè *i_start* è alto

2.1.3 MEM_LOAD

Stato in cui viene caricato l'indirizzo della memoria (ottenuto dall'input) sul segnale *o_mem_addr*.

In questo stato vengono settati a 1 i segnali *o_mem_en* e *load_to_mem* per permettere di accedere alla memoria.

2.1.4 MEM_READ

Stato in cui viene salvato il valore contenuto nell'indirizzo di memoria e passato tramite il segnale *i_mem_data* al componente (in particolare al registro selezionato).

In questo stato vengono settati a 1 i segnali *o_mem_en* e *load_to_mem* per permettere di accedere alla memoria.

2.1.5 WRITE_OUT

Stato in cui vengono impostati ad 1 i segnali *output_load* e *o_done* per abilitare i quattro segnali di output.

2.1.6 RESET

Stato di reset, a cui si accede quando il segnale di RESET è alto: in questo stato vengono inizializzati i vari registri e segnali.

Rappresentazione schematica

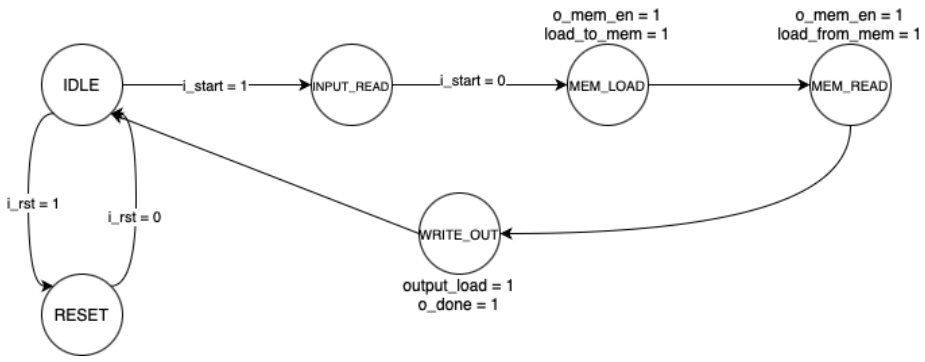

Figura 1: Diagramma della macchina a stati

2.2 Datapath

I primi due bit (contati tramite il contatore *bit_counter*, che si aggiorna ad ogni ciclo di clock) vengono salvati nel registro *d_sel*, i restanti vengono salvati sul registro *reg_input*, inizialmente impostato a tutti zeri.

I bit vengono salvati sul registro *d_sel* a partire dal bit più significativo, mentre sul registro *reg_input* vengono salvati "accodando" ogni bit nella posizione meno significativa e traslando quelli già presenti verso sinistra: in questo modo il registro è sempre composto da 16 bit, con gli zeri nei bit più significativi, come da specifica.

Quando i segnali *load_to_mem* e *o_mem_en* sono alti, il registro *reg_input* viene copiato sul segnale *o_mem_addr*.

Quando invece i segnali *load_from_mem* e *o_mem_en* sono alti, il registro *i_mem_data* viene copiato sul registro identificato da *d_sel*.

A sua volta, pilotati dal segnale *output_load* (che altro non è che una copia accessibile del segnale *o_done*), quattro multiplexer decidono se mostrare sui segnali di output soli zeri (DONE basso) o il valore dei relativi registri (DONE alto).

Rappresentazione schematica

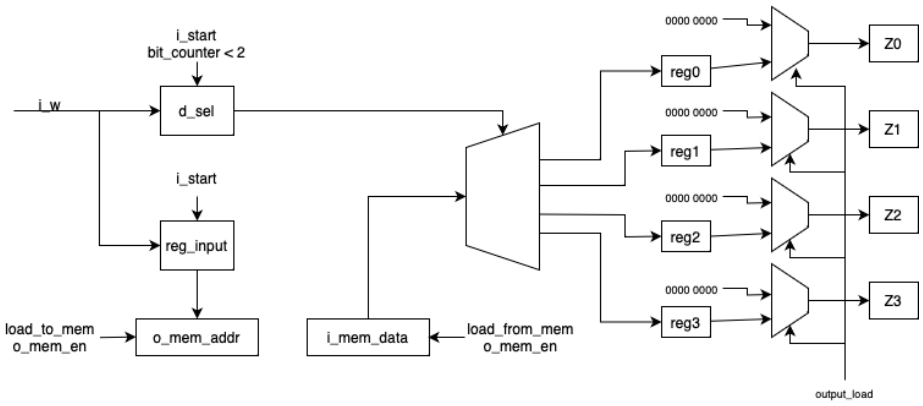

Figura 2: Diagramma del datapath

3 Risultati Sperimentali

3.1 Simulazioni Behavioral

Per verificare il corretto funzionamento del componente sintetizzato ho utilizzato i 7 testbench che ci sono stati forniti di esempio, che ho valutato esaustivi nel testare il comportamento del componente. In particolare viene controllato con input di varie lunghezze il comportamento del componente, verificando che a seguito del reset tutti i segnali ed i registri vengano correttamente inizializzati, che i segnali di uscita vengano abilitati (e restituiscano, quindi, il valore ottenuto dalla memoria) solo durante il ciclo di clock in cui il segnale DONE sia alto e che, al secondo segnale alto di DONE, vengano correttamente mostrati in uscita i valori precedenti, salvati nei relativi registri.

Di seguito vengono mostrati i grafici con i comportamenti del componente nei sette testbenches e gli output restituiti da Vivado.

3.1.1 Testbench 1

Il testbench viene superato, restituendo in output il seguente log:

```
Failure: Simulation Ended! TEST PASSATO (EXAMPLE)
Time: 34100 ns  Iteration: 0
```

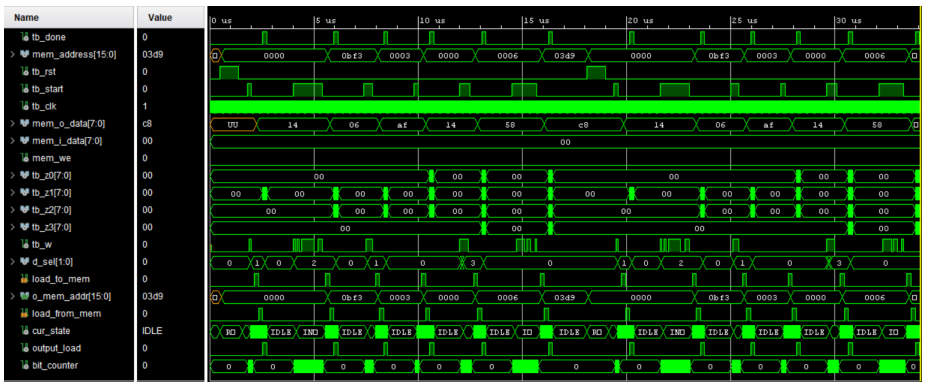

Figura 3: Grafico della simulazione *Behavioral*

3.1.2 Testbench 2

Il testbench viene superato, restituendo in output il seguente log:

```
Failure: Simulation Ended! TEST PASSATO (EXAMPLE)
Time: 32800 ns  Iteration: 0
$finish called at time : 32800 ns
```

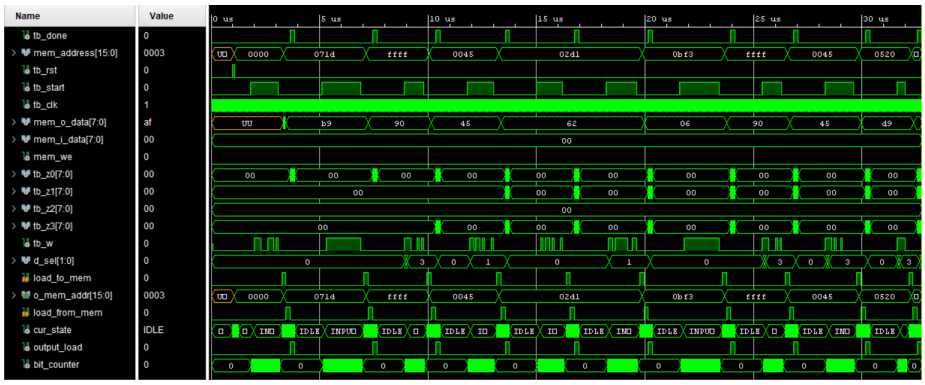

Figura 4: Grafico della simulazione *Behavioral*

3.1.3 Testbench 3

Il testbench viene superato, restituendo in output il seguente log:

```
Failure: Simulation Ended! TEST PASSATO (EXAMPLE)
Time: 58300 ns  Iteration: 0
$finish called at time : 58300 ns
```

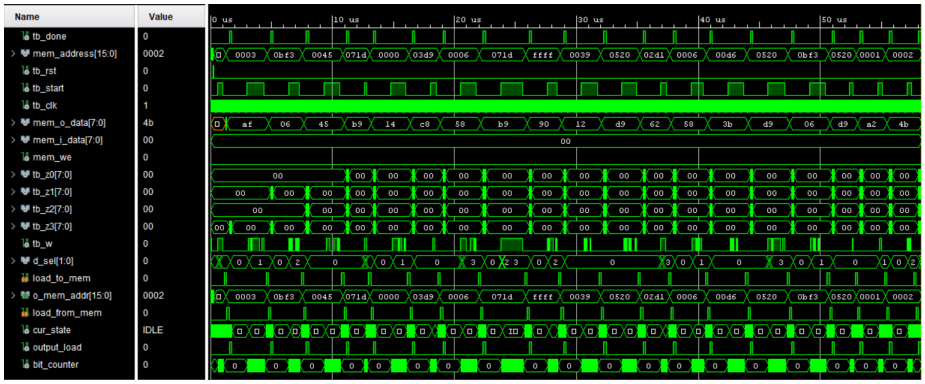

Figura 5: Grafico della simulazione *Behavioral*

3.1.4 Testbench 4

Il testbench viene superato, restituendo in output il seguente log:

```
Failure: Simulation Ended! TEST PASSATO (EXAMPLE)
Time: 12900 ns  Iteration: 0
$finish called at time : 12900 ns
```

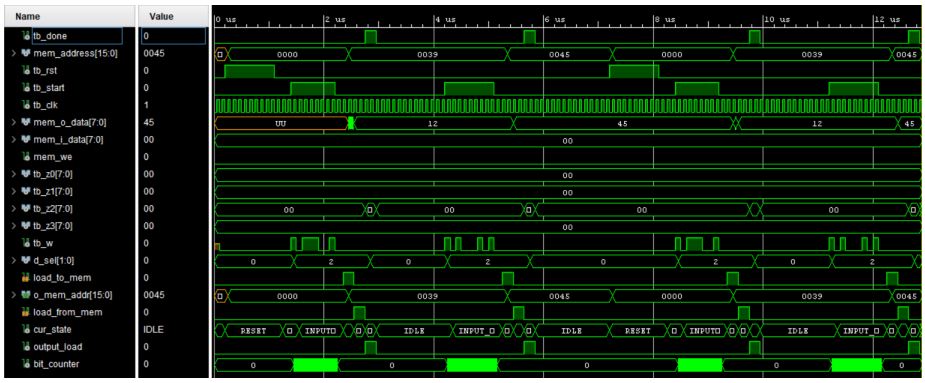

Figura 6: Grafico della simulazione *Behavioral*

3.1.5 Testbench 5

Il testbench viene superato, restituendo in output il seguente log:

```
Failure: Simulation Ended! TEST PASSATO (EXAMPLE)
Time: 62700 ns  Iteration: 0
$finish called at time : 62700 ns
```

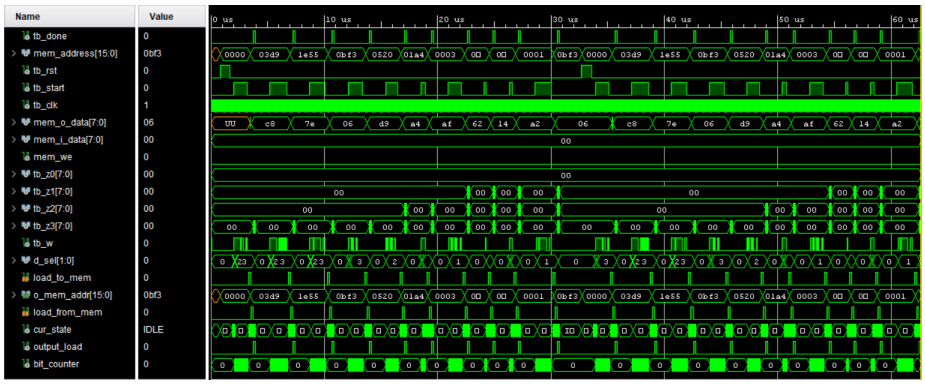

Figura 7: Grafico della simulazione *Behavioral*

3.1.6 Testbench 6

Il testbench viene superato, restituendo in output il seguente log:

```
Failure: Simulation Ended! TEST PASSATO (EXAMPLE)
Time: 3700 ns  Iteration: 0
$finish called at time : 3700 ns
```

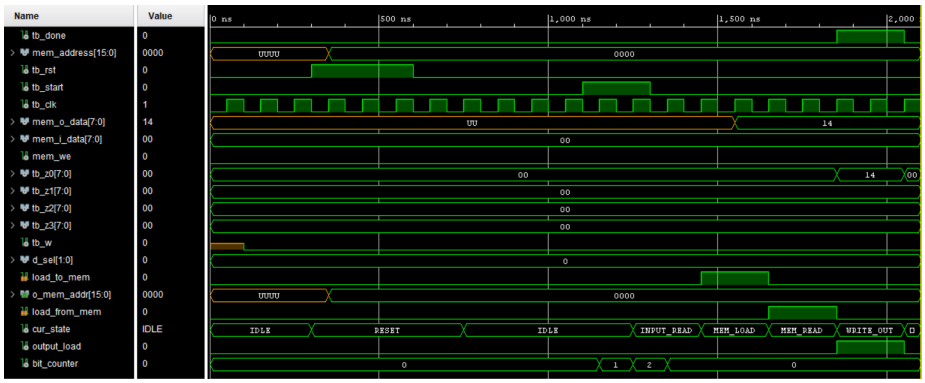

Figura 8: Grafico della simulazione *Behavioral*

3.1.7 Testbench 7

Il testbench viene superato, restituendo in output il seguente log:

```
Failure: Simulation Ended! TEST PASSATO (EXAMPLE)
Time: 2100 ns  Iteration: 0
$finish called at time : 2100 ns
```

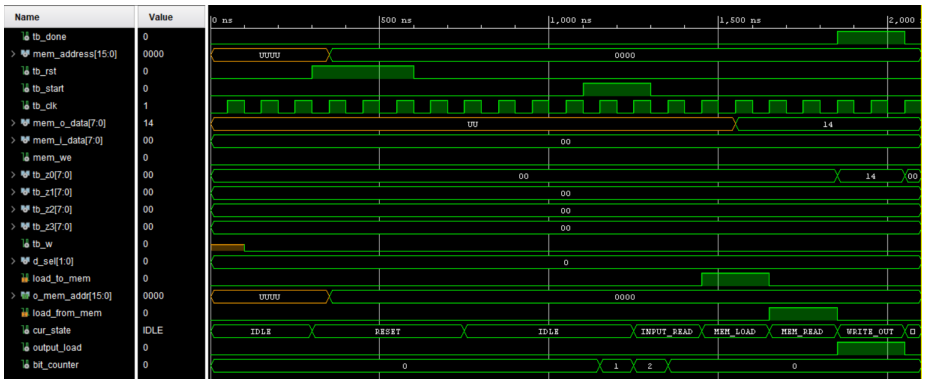

Figura 9: Grafico della simulazione *Behavioral*

3.2 Simulazioni Post-Synthesis Functional

Il comportamento nelle simulazioni *Post-Synthesis Functional* è uguale a quello nelle simulazioni *Behavioral* e supera tutti i 7 testbenches di prova.

Utilizzando, ad esempio, il testbench 5, il test viene superato e restituisce in output il seguente log:

```
Failure: Simulation Ended! TEST PASSATO (EXAMPLE)
Time: 62700100 ps  Iteration: 0
```

```
$finish called at time : 62700100 ps
```

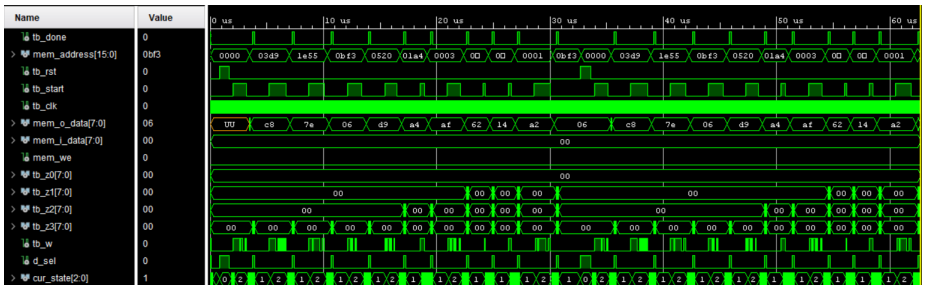

Figura 10: Grafico della simulazione *Post-Synthesis* durante il testbench 5

3.2.1 Report di Sintesi

Timing Il report di timing conferma che il progetto soddisfa i requisiti di timing per il clock indicati nelle specifiche (entro i 100ns) con ampio margine.

Utilizzando il comando *report_timing* con il testbench 5, ad esempio, si ottiene:

```
Timing Report
Slack (MET) :  96.983ns  (required time - arrival time)
```

Utilization Il report sull'*utilization* mostra che il progetto non ha alcun latch, confermando la bontà delle scelte progettuali e della logica di funzionamento.

Usando il comando *report_utilization* con, ad esempio, il testbench 5 si ottiene:

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	34	0	134600	0.03
LUT as Logic	34	0	134600	0.03
LUT as Memory	0	0	46200	0.00
Slice Registers	77	0	269200	0.03
Register as Flip Flop	77	0	269200	0.03
Register as Latch	0	0	269200	0.00
F7 Muxes	0	0	67300	0.00
F8 Muxes	0	0	33650	0.00

4 Conclusioni

Per questo progetto ho seguito i consigli del prof. Terraneo, utilizzando quindi due componenti: uno (*project_reti_logiche*) che si occupa della macchina a stati, collegato ad un secondo (*datapath*) che gestisce appunto il datapath della macchina e quindi il funzionamento dei registri interni in base ai segnali del componente.

Prima di iniziare a scrivere effettivamente codice ho progettato "teoricamente", su alcuni fogli di carta, il funzionamento della macchina a stati e la logica di funzionamento del datapath.

Una volta ottenuto un risultato che ritenevo adatto per le mie necessità ho proseguito ottimizzando il tutto: ho rimosso dalla macchina a stati quanti più stati possibili, lasciando solo quelli strettamente necessari e concentrando in quelli l'impostazione dei segnali in base alle necessità, rinominandoli poi in base alla funzione svolta.

Ho poi modificato anche il datapath, che inizialmente aveva una logica di funzionamento per cui tutti i bit in ingresso, in tutto il periodo in cui il segnale START era alto, venivano copiati su un registro. Il contenuto di questo registro veniva a sua volta separato: i primi due bit venivano copiati nel registro *d_sel*, mentre i restanti venivano immessi in un sommatore con, come secondo addendo, 16 zeri, il cui risultato veniva a sua volta copiato in un terzo registro, che veniva poi ulteriormente copiato sul segnale per accedere al contenuto nella memoria. A sua volta, quando veniva letto dalla memoria, il contenuto veniva copiato in un quarto registro che agiva come input del demultiplexer.

Ho rimosso tutti questi registri (con annessi segnali che ne abilitavano la scrittura o meno) fino ad arrivare al risultato attuale, decisamente più snello e ottimizzato, sfruttando il contatore *bit_counter* per capire se copiare o meno sul registro *d_sel*.

Il segnale *output_load* è invece una soluzione per potere accedere dal datapath al segnale DONE (che, essendo un segnale di output, non può essere letto): questo segnale agisce come copia del segnale DONE, venendo modificati entrambi negli stessi momenti dalla macchina a stati.

Solo dopo le ottimizzazioni ho iniziato a scrivere il vero e proprio codice, riuscendo così in brevissimo tempo a poter testare i testbench per verificare la correttezza funzionale del tutto.

Dopo alcuni aggiustamenti minori, che in molti casi erano semplicemente errori di sintassi di VHDL, sono arrivato nella situazione finale che ho poi consegnato, in cui tutti i testbench venivano passati sia testando il componente con le simulazioni *behavioral* che con quelle *post synthesis*.