Contents

1 Reti Logiche - Documentazione Prova Finale

Documentazione relativa al progetto finale del corso "**Reti Logiche**", professore Fornaciari William.

Studenti D'Alessio Edoardo (*codice persona 10714021*) e Deidier Simone (*codice persona 10742651*).

Anno Accademico 2022/2023

1.1 Introduzione

1.1.1 Specifica

Il tema della Prova Finale consiste nell'ideazione, realizzazione mediante linguaggio VHDL e relativo testing di un sistema di inoltro dati provenienti da una memoria esterna verso una delle possibili quattro uscite. I dati in uscita devono essere persistenti nel tempo e devono cambiare solamente in caso di riscrittura. Più precisamente, il componente riceve indicazioni circa una locazione di memoria ed il canale di output da utilizzare mediante un ingresso seriale da un bit, mentre le uscite del sistema forniscono tutti i bit della parola di memoria in parallelo.

Il componente da sintetizzare presenta le seguenti porte per interfacciarsi con il circuito ed i sistemi esterni:

- **CLOCK** (`i_clk`) è il segnale di clock in ingresso.
- **W** (`i_w`) è il segnale di input dati; esso comprende due bit relativi all'uscita verso la quale indirizzare il dato di memoria (2 bit $\implies 2^2 = 4$ valori $\implies Z_0, Z_1, Z_2, Z_3$), ed un numero $0 \leq x \leq 16$ di bit relativi all'indirizzo di memoria stesso. *La memoria accetta solamente indirizzi 16 bit, pertanto il valore letto in input viene esteso con un numero di bit a 0 tali per renderlo della dimensione desiderata.*
- **START** (`i_start`) è il segnale di in ingresso che, fintatntoché vale 1, segnala al componente di leggere il bit sulla linea *W*. Il segnale è sicuramente attivo almeno per 2 cicli di clock (*lettura dell'uscita sulla quale visualizzare il valore in output*) e per un massimo di 18 (*2 cicli + 16 per la lettura dell'indirizzo di memoria*).
- **RESET** (`i_rst`) è il segnale che inizializza la macchina per ricevere il primo segnale di *START*.
- **Z_0, Z_1, Z_2, Z_3** (`o_z0, o_z1, o_z2, o_z3`) sono i quattro canali di output, ognuno da 8 bit ciascuno (*i valori di output della memoria a cui il componente si interfaccia sono sempre di dimensione 8 bit*).
- **DONE** (`o_done`) è il segnale di output che comunica la fine dell'elaborazione, esso *assume il valore 1 nel ciclo di clock in cui il componente mostra i valori aggiornati su tutte e quattro le uscite*).
- **MEMORY ADDRESS** è il segnale di output da 16 bit che manda l'indirizzo alla memoria.
- **MEMORY DATA** è il segnale da 8 bit proveniente dalla memoria in seguito ad una richiesta di lettura.

- **MEMORY ENABLE** (`o_mem_en`) è il segnale di *ENABLE* da dover mandare alla memoria per poter comunicare sia in lettura che in scrittura.
- **MEMORY WRITE ENABLE** (`o_mem_we`) è il segnale di *WRITE ENABLE*; è necessario che questo segnale valga *1* per poter scrivere in memoria, mentre per leggere esso deve essere *0*.

```
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;
```

Figure 1: *Interfaccia del componente descritta in VHDL.*

1.1.2 Esempio di funzionamento

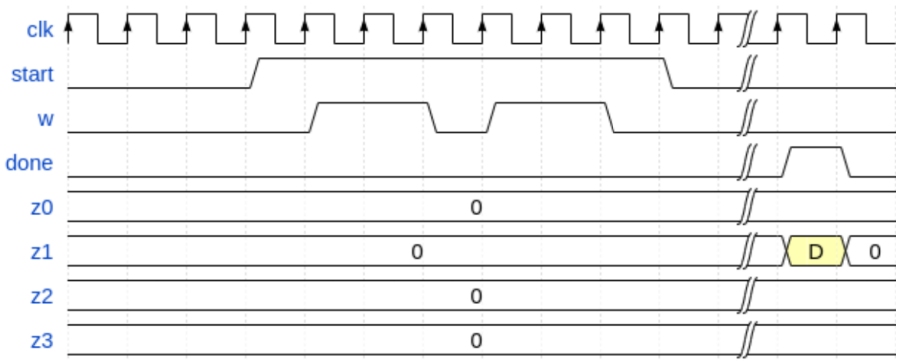

Figure 2: *Esempio di funzionamento del componente da progettare.*

Nell'esempio il segnale di *START* vale *1* per 7 cicli di clock: durante il primo ed il secondo ciclo di clock il componente legge il valore del segnale input *W* che andrà ad indicare su quale uscita mostrare l'output (INPUT = $(01)_2 \implies (1)_{10} \implies Z_1$); durante i successivi cinque cicli di clock in cui il segnale *START* rimane ad *1*, il componente legge il valore di *W*, il quale questa volta indica l'indirizzo di memoria da cui andare a leggere il dato da mostrare in output (*il componente legge solamente 5 bit di indirizzo, ovvero* $(10110)_2$*, ma alla memoria andrà a richiedere il valore all'indirizzo* $(0000000000010110)_2$*, estendendo con tutti 0 il valore letto in input per formare un indirizzo da 16 bit*). Il componente interrogherà la memoria esterna richiedendo il valore (*D*) presente all'indirizzo specificato dal'input e, una volta che la memoria ha correttamente comunicato l'indirizzo al componente, esso lo mostra sull'output indicato ad inizio sequenza. Il valore *D* viene mostrato in output per un solo ciclo di clock, dopodiché l'uscita torna a mostrare *0* come valore; inoltre, nello stesso ciclo di clock, il segnale *DONE* vale *1*, e tutte le altre uscite (Z_0, Z_2, Z_3) mostrano l'ultimo valore D_{old} che hanno mostrato in output precedentemente (*in questo esempio mostrano tutti 0, si suppone che la sequenza sia avvenuta dopo aver dato il segnale RESET al componente*).

1.2 Architettura

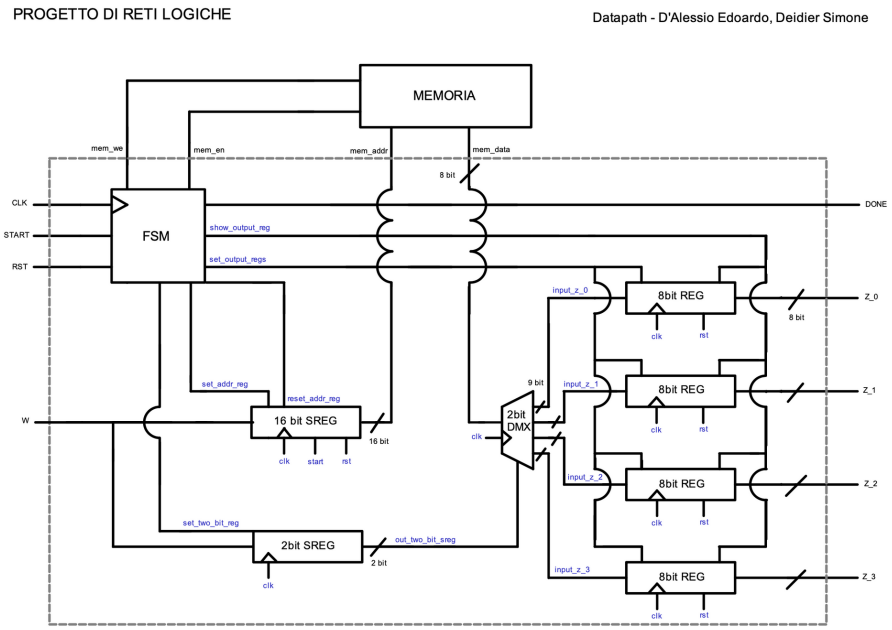

Figure 3: *Datapath del componente progettato.*

La figura mostra il *datapath* intero del componente da noi progettato (*memoria esclusa, il componente è tutto ciò che sta all'interno della linea tratteggiata grigia*) che rispetta le specifiche richieste. Di segutio i componenti verranno analizzati più nel dettaglio.

1.2.1 *FSM*

Questo componente implementa la *macchina a stati finiti* che abbiamo ideato per regolare la maggior parte dei segnali interni al componente; essa presenta i seguenti stati:

- **S_WAIT**, stato iniziale di *wait* in cui la macchina aspetta l'inizio delle operazioni (*ovvero il segnale* `start = 1`). Il comando *reset* (`reset = 1`) permette una corretta inizializzazione di tutti i registri ed i componenti. Una volta che arriva il segnale di *start*, la macchina effettua la transizione verso lo stato *SELECT_OUTPUT_LINE*.

- **SELECT_OUTPUT_LINE**, stato in cui la macchina acquisisce il dato che indica su quale tra le quattro uscite visualizzare il nuovo valore richiesto dalla memoria; lo stato dura *un solo ciclo di clock*, una volta terminato la macchina transisce verso lo stato *TAKE_MEM_ADDR* (*il perché della durata di un solo ciclo di clock per leggere due bit viene spiegato più approfonditamente nella sezione "**Registri a scorrimento**"*).
- **TAKE_MEM_ADDR**, stato in cui la macchina acquisisce i bit relativi all'indirizzo di memoria. Fintantoché il bit di *start* vale *1* la macchina rimane in questo stato, appena il bit *start* vale *0* avviene la transizione verso lo stato *MEM_REQ*.
- **MEM_REQ**, stato della durata di un singolo ciclo di clock nel quale la *FSM* invia i dati alla memoria (`mem_en = 1` *ed il valore dell'indirizzo di memoria*). Lo stato successivo è *MEM*.
- **MEM**, stato della durata di un ciclo di clock nel quale la memoria accede all'indirizzo indicato dallo stato precedente. Alla fine di questo stato il valore del segnale *mem_data* proveniente dalla memoria conterrà il valore *D* di memoria all'indirizzo comunicato. Lo stato successivo è *SET_OUT_REGS*.
- **SET_OUT_REGS**, stato della durata di un ciclo di clock nel quale la *FSM* invia ai registri di output il comando *set_output_regs*, comando **master set** (*descirtto più nel dettaglio nella sezione "**Registri di output**"*). Durante questo stato il registro selezionato (*stato SELECT_OUTPUT_LINE*) assume come valore il valore *D* proveniente dalla memoria (*segnale mem_data*). Lo stato successivo è *SHOW_OUTPUT*.
- **SHOW_OUTPUT**, stato della durata di un ciclo di clock nel quale la *FSM* imposta il segnale *show_output_regs* ad *1* (*descirtto più nel dettaglio nella sezione "**Registri di output**"*), segnale che permette a tutti i registri di comunicare in output il valore da loro salvato. Lo stato successivo è *S_DONE*.
- **S_DONE**, stato della durata di un ciclo di clock nel quale la *FSM* imposta il segnale *done* ad *1*. In questo ciclo di clock ogni uscita output, Z_0, Z_1, Z_2, Z_3, mostra il valore salvato nei registri ed il segnale *o_done* vale *1*. A fine del ciclo la *FSM* manda il segnale di *reset* al registro da 16 bit per l'indirizzo di memoria, e ritorna allo stato *S_WAIT*.

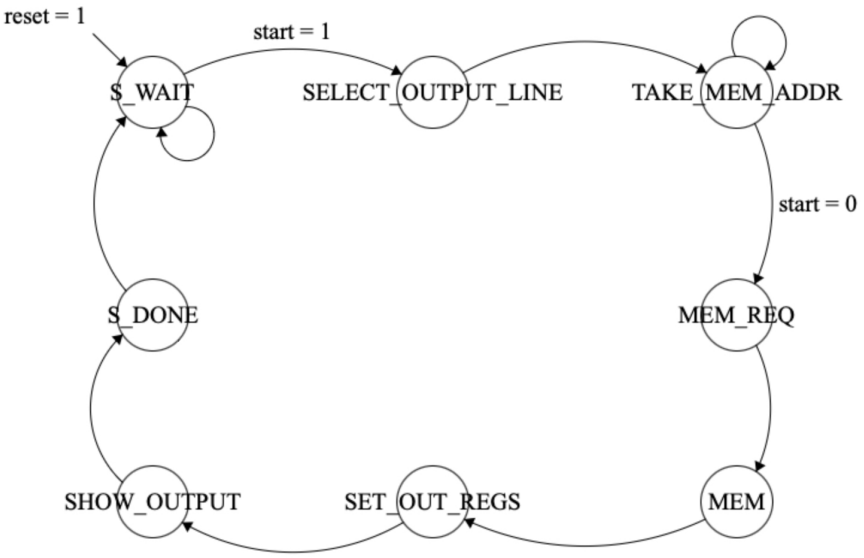

Figure 4: *Diagramma degli stati della FSM.*

1.2.2 Registri a scorrimento

Abbiamo utilizzato dei registri a scorrimento a sinistra (*left shift register*) per salvare le informazioni sia della scelta dell'output, sia l'indirizzo di memoria; questa scelta si è rivelata estremamente efficace dato che l'input viene dato serialmente un bit alla volta, infatti utilizzare registri non a scorrimento avrebbe implicato l'utilizzo di contatori accessori o di ulteriori accorgimenti a nostro parere meno efficaci di un registro a scorrimento. Per quanto riguarda il registro a scorrimento da 2 bit, esso salva nella posizione dell'*LSB* il valore in input (*segnale W*) dopo aver effettuato uno shift verso sinistra; questo accade solamente nello stato *SELECT_OUTPUT_LINE* dive il segnale di *set*, ovvero *set_two_bit_reg* vale *1*. Dato che la *FSM* impiega un ciclo di clock per la transizione di stato, il segnale *set* vale *1* anche nello stato *S_WAIT*, in modo tale che, appena il segale *start* vale *1*, il primo bit del segnale *W* viene automaticamente letto e quindi è necessario una sola lettura successiva per aver letto correttamente i due bit di scelta dell'output. Per quanto riguarda il registro da 16 bit, il funzionamento è essenzialmente identico al registro da 2 bit, ma con un accorgimento fondamentale: all'inizio di ogni operazione, con il comando *reset* e *reset_addr_reg*, il registro viene impostato con valore "0000000000000000"; questo è estremamente comodo perché, essendo un registro a scorrimento verso sinistra, se i bit inseriti per l'indirizzo sono minori di 16, automaticamente il valore è esteso con tutti *0* come da specifica, senza dover effettuare ulteriori operazioni. Etrambi i registri sono temporizzati in base al *clock*, ed anche il registro da 16 bit possiede un segnale di *set* chiamato *set_addr* che viene governato dalla *FSM* (*impostato ad 1 nello stato TAKE_MEM_ADDR*); inoltre il registro da 16 bit salva i valori in input solo se l'espressione SET_ADDR $\wedge$ START risulta *true*.

1.2.3 *DeMux*

Il DeMultiplexer che abbiamo deciso di utilizzare nel nostro circuito ha il compito di prendere in input il valore *D*, che la memoria restituisce dopo la richiesta da parte del componente, e di indirizzarlo al *registro di output* corretto in base all'uscita precedentemente selezionata dall'input del sistema. Il segnale di input è quindi *mem_data*, mentre il segnale di select del *DeMux* è il valore in memoria nel registro a 2 bit, ovvero *out_two_bit_sreg*. La particolarità di questo componente è l'output, infatti il segnale *mem_data* in input al *DeMux* è un segnale da 8 bit (*D*), invece ognuno dei quattro segnali di output del *DeMux*, ovvero i segnali *input_z_x*, sono **9 bit**; abbiamo deciso infatti di aggiungere, oltre agli 8 bit del valore di input, un segnale di ***slave set*** che arrivi direttamente al registro di output intreressato assieme al dato di memoria da salvare. Questa scelta si è rivelata estremamente efficace risparmiando informazioni da mandare alla *FSM* (*descirtto più nel dettaglio nella sezione "**Registri di output**"*).

1.2.3.1 ***Esempio di funzionamento DeMux*** *Supponendo il segnale di memoria* $D = (10100011)_2$ *ed il segnale di select del DeMux out_two_bit_sreg* $= (01)_2$*, il componente manderà in output i seguenti valori:*

$$\begin{cases} \text{out\_0} = (0)_2 \\ \text{out\_1} = (10100011)_2 \\ \text{out\_2} = (0)_2 \\ \text{out\_3} = (0)_2 \\ \text{set\_0} = (0)_2 \\ \text{set\_1} = (1)_2 \\ \text{set\_2} = (0)_2 \\ \text{set\_3} = (0)_2 \end{cases}$$

1.2.4 Registri di output

Per permettere al nostro componente di mantenere in memoria per ogni uscita l'ultimo valore D_{old} di memoria, abbiamo deciso di implementare un *registro di output* per ognuna delle quattro uscite presenti. Questi sono registri da 8 bit non a scorrimento, che presentano degli accorgimenti interessanti:

- Ogni registro presenta *de segnali di set*, uno singolo per ogni registro che proviene dal *DeMux*, chiamato anche ***slave set***, precedentemente descritto, mentre l'altro è in comune tra tutti i registri e proviene direttamente dalla *FSM*, detto anche ***master set***. Un registro sovrascrive il dato in memoria con quello in ingresso $\iff \text{SET}_{\text{master}} \wedge \text{SET}_{\text{slave}} = 1$. Questa scelta di progettazione è risultata estremamente efficace perché ci ha permesso di poter scandire il momento del *set* dei registri tramite *FSM*, senza dover però far sapere alla *FSM* a quale registro precisamente inviare il segnale di set.
- Ogni registro presenta un segnale *show_output* regolato dalla *FSM* ed in comune tra tutti i registri; questo segnale ci permette di scandire tramite *FSM* il momento in cui i registri devono effettivamente mostrare in output il valore da loro salvato (*ovvero nel ciclo di clock quando il segnale DONE vale 1*), mentre in tutti gli altri stati i registri porteranno in output un vettore da 8 bit di zeri, come richiesto da specifica.

Anche i *registri di output* presentano una funzione di *reset* che azzera completamente il loro valore in memoria quando l'omonimo comando viene inviato dalla *FSM*.

1.3 Risultati sperimentali

1.3.1 Sintesi

Il componente da noi ideato e descritto tramite linguaggio *VHDL* è risultato completamete e correttamente sintetizzabile. La sintesi ci ha permesso inoltre di poter valutare alcuni aspetti cruciali del progetto e del nostro componente, infatti, grazie al *report sull'utilizzo*, abbiamo potuto appurare che il nostro componente utilizza solamente *registri flip-flop*, portando a zero il numero di *registri latch* utilizzati.

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs*	32	0	0	134600	0.02
LUT as Logic	32	0	0	134600	0.02
LUT as Memory	0	0	0	46200	0.00
Slice Registers	121	0	0	269200	0.04
Register as Flip Flop	121	0	0	269200	0.04
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

Figure 5: *Parte del report di utilizzo di Vivado, come si può notare i "Register as Latch" sono 0.*

Un altro elemento cruciale da verificare post sintesi è il *constraint* di tempo del clock, ovvero i ritardi che ogni porta e componente logico apportano al sistema. Dal *report sui tempi* abbiamo potuto controllare che il *constraint* di clock da *100 ns* è ampiamente soddisfatto, con uno ***Slack** pari a **97 ns***.

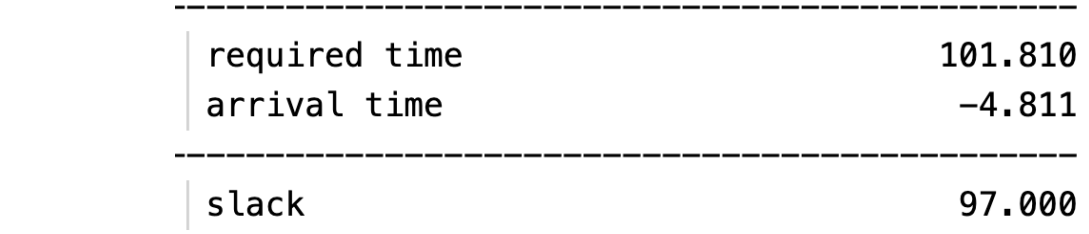

Figure 6: *Calcolo dello Slack effettuato da Vivado in fase post sintesi.*

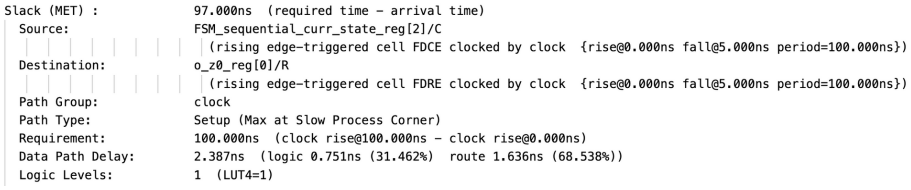

Figure 7: *Dal report di Vivado abbiamo potuto anche apprendere il delay imposto dal nostro datapath: 2.387 ns.*

1.3.2 Simulazioni

Oltre alla corretta sintetizzazione del circuito, un altro degli elementi fondamentali era la correttezza del componente. Per fare ciò ci siamo appoggiati all'utilizzo di *testbench*, ovvero codici *VHDL* che permettono di dare input ad un sistema o componente e verificarne il corretto funzionamento tramite *asserzioni* su valori di segnali, registri o stati della *FSM*. Dato che il componente è stato completamente ideato da noi, abbiamo potuto applicare alcune delle regole base del ***white-box testing***, andando ad individuare la maggior parte dei casi e stati critici in cui il nostro componente si può trovare; un rapido elenco dei casi testati:

- *test di indirizzo vuoto*, abbiamo voluto testare il caso in cui il segnale *start* rimane attivo per il minimo della sua durata (*2 cicli di clock per l'uscita*), controllando così che il registro di memoria fosse correttamente inizializzato al valore *0*.
- *test di indirizzo pieno*, contrariamente al test precedente, abbiamo voluto testare il caso in cui il segnale *start* rimane attivo per il massimo possibile (*18 cicli di clock, due per l'uscita, sedici per l'indirizzo*), dando un indirizzo casuale $i \geq 2^{15} = 32768$, controllando che il *MSB* dell'indirizzo fosse sempre *1* e che il resto dei bit salvati nel registro fossero corretti.
- *test di massimo indirizzo*, a seguito del test precedente abbiamo voluto testare il caso in cui l'indirizzo di memoria sia $(1111111111111111)_2 = 2^{16} - 1 = (65535)_{10}$, controllando il corretto funzionamento del registro a scorrimento da 16 bit inizializzato a *0*.

- *test di tutte le uscite*, abbiamo voluto testare una situazione in cui vengano dati quattro diversi indirizzi di memoria, ognuno assegnato ad una uscita diversa, controllando così il funzionamento generale del componente; abbiamo potuto controllare il corretto funzionamento dell'estensione a *0* degli indirizzi non 16 bit, il funzionamento dei *registri di output* che permettono di visualizzare sempre lo stesso valore D_{old} se non sovrascritti, ed il funzionamento del *DeMux* per selezionare le quattro diverse uscite.
- *test di overwrite delle uscite*, dopo il test precedente abbiamo voluto controllare il funzionamento della sovrascrittura dei dati nei *registri di output*, abbiamo così modificato la *testbench* affinché vengano dati otto diversi indirizzi di memoria che vengono divisi equamente tra le quattro uscite, controllando che i dati in output alla fine di ogni operazione siano sempre corretti, anche dopo un *overwrite* di dati di un *registro di output*.
- *test generali sul comando reset*, abbiamo voluto controllare che il comando *reset*, comando che in qualsiasi momento e stato il componente si trovi deve riportare tutto allo stato iniziale, funzionasse correttamente; per fare ciò abbiamo creato una *testbench* che azionava il segnale *reset* in momenti diversi: ad inizio test, dopo il segnale di *start*, dopo che la memoria abbia comunicato il valore D, dopo che il componente abbia salvato nel *registro di output* il valore D da memoria, dopo che il componente abbia visualizzato sull'uscita il valore D salvato nel *registro di output*. In ognuno di questi casi la *testbench* dopo aver dato il segnale di *reset* controllava che la *FSM* si trovasse sempre nello stato *S_WAIT*, che ogni registro fosse reinizializzato a *0*, e che ogni componente fosse pronto per ripartire con le operazioni nel modo corretto.

1.4 Conclusioni

Ci riteniamo soddisfatti riguardo il componente da noi ideato, sviluppato e testato. Siamo consapevoli che grazie alle conoscenze apprese nel corso di *Reti Logiche* siamo partiti da una buona idea di circuito iniziale che, durante lo sviluppo, abbiamo modellato e migliorato rendendolo sempre più efficace. La quantità di test e simulazioni che sono state fatte ci permettono di essere consapevoli di aver fatto un componente funzionante (*anche nei casi limite*) e che rispetti con grande margine tutti i *constraint* che sono stati imposti dalla specifica.

Questo progetto ci ha permesso di avere una visione molto dettagliata della programmazione hardware, partendo dall'ideazione di un circuito e dalla scelta dei componenti che si possono utilizzare, per poi arrivare alla descrizione tramite linguaggio *VHDL*, linguaggio completamente nuovo per noi che nel corso di questo progetto abbiamo saputo imparare ed apprezzare, così come il software *Vivado*, che ci ha permesso soprattutto di visualizzare graficamente ciò che accadeva al nostro componente, aiutandoci moltissimo nella ricerca ed individuazione dei bachi e delle criticità. La parte di ideazione e stesura dei test ci ha molto messo alla prova, ma allo stesso tempo ci ha consapevolizzato dell'importanza di questi ultimi per avere un componente affidabile e funzionante.