RELAZIONE

Prova finale (progetto di reti logiche), 2022 – 2023

Progetto svolto da: Andrea Ferrini 10693727 (codicepersona2), Figini Riccardo 10709191 (codicepersona1)

Sommario

A cura di:

- *Figini Riccardo, matricola: 957027, codice persona: 10709191*
- *Ferrini Andrea, matricola: 955880, codice persona: 10693727*

Introduzione

Lo scopo del progetto è descrivere in VHDL un modulo Hardware con l'obiettivo di lavorare accedendo a una memoria, costruendo l'indirizzo a cui accedere mediante la lettura di un ingresso seriale (*W*) e portando il contenuto di tale indirizzo in uscita su uno dei quattro canali di uscita da 8 bit (*Z0*, *Z1*, *Z2*, *Z3*).

Sono presenti segnali aggiuntivi (sia da specifica che da noi implementati), che verranno descritti nei paragrafi successivi, anche con l'aggiunta di disegni.

Ecco un **breve esempio**, senza scendere nei dettagli implementativi per adesso:

- *W* = 11100101, supponendo perciò che *START* resti alto (=1) per 8 cicli di clock
- I due bit più siificativi (11) individuano il canale di uscita *Z3*
- Quindi *N* = 100101 → indirizzo: 0000000000100101 (0025_{hex})
- Supponendo che all'indirizzo 0025 sia presente il dato A9, verrà portato in uscita sul canale *Z3* visibile quando *DONE* = 1, che rimarrà inalterato fino a nuova riscrittura o al *RESET*

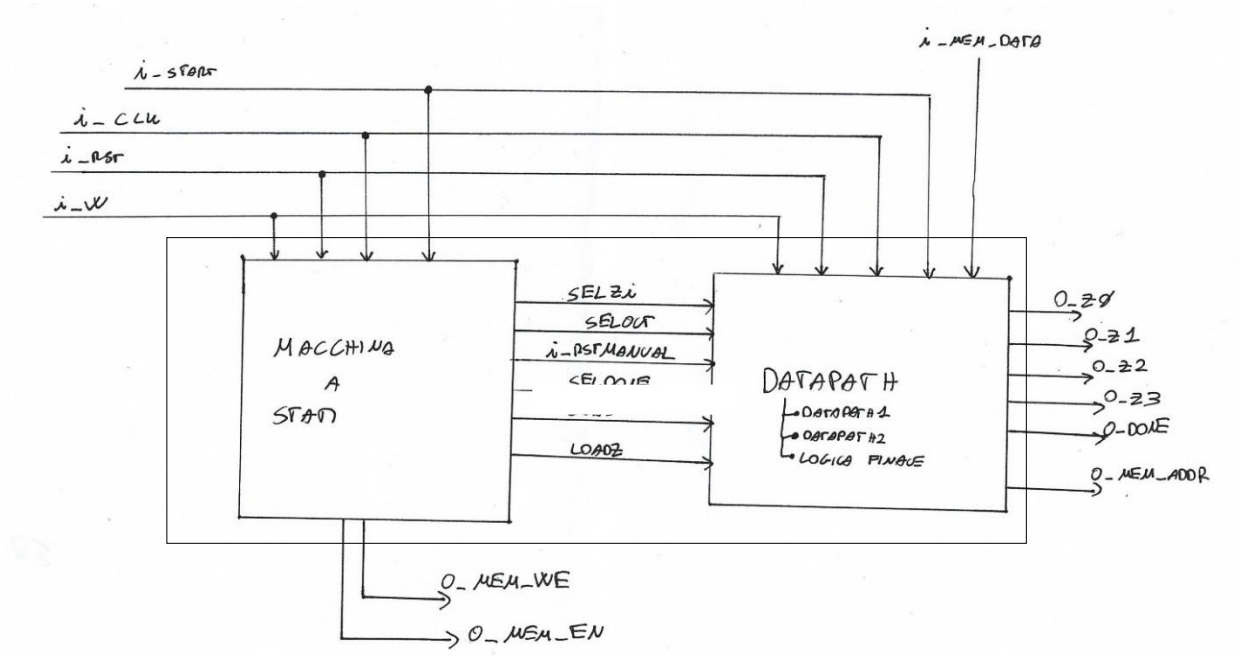
Si noti che la sequenza di bit *N* è stata estesa su 16 bit aggiungendo sufficienti 0 nella posizione più significativa e il canale di uscita è stato selezionato univocamente grazie alla codifica su due bit.

Una seconda lettura (*START* = 1) potrà essere possibile solo quando *DONE* sarà tornato al valore 0 (ricordando che il massimo tempo tra *START* = 0 e *DONE* = 1 è di 20 cicli di clock).

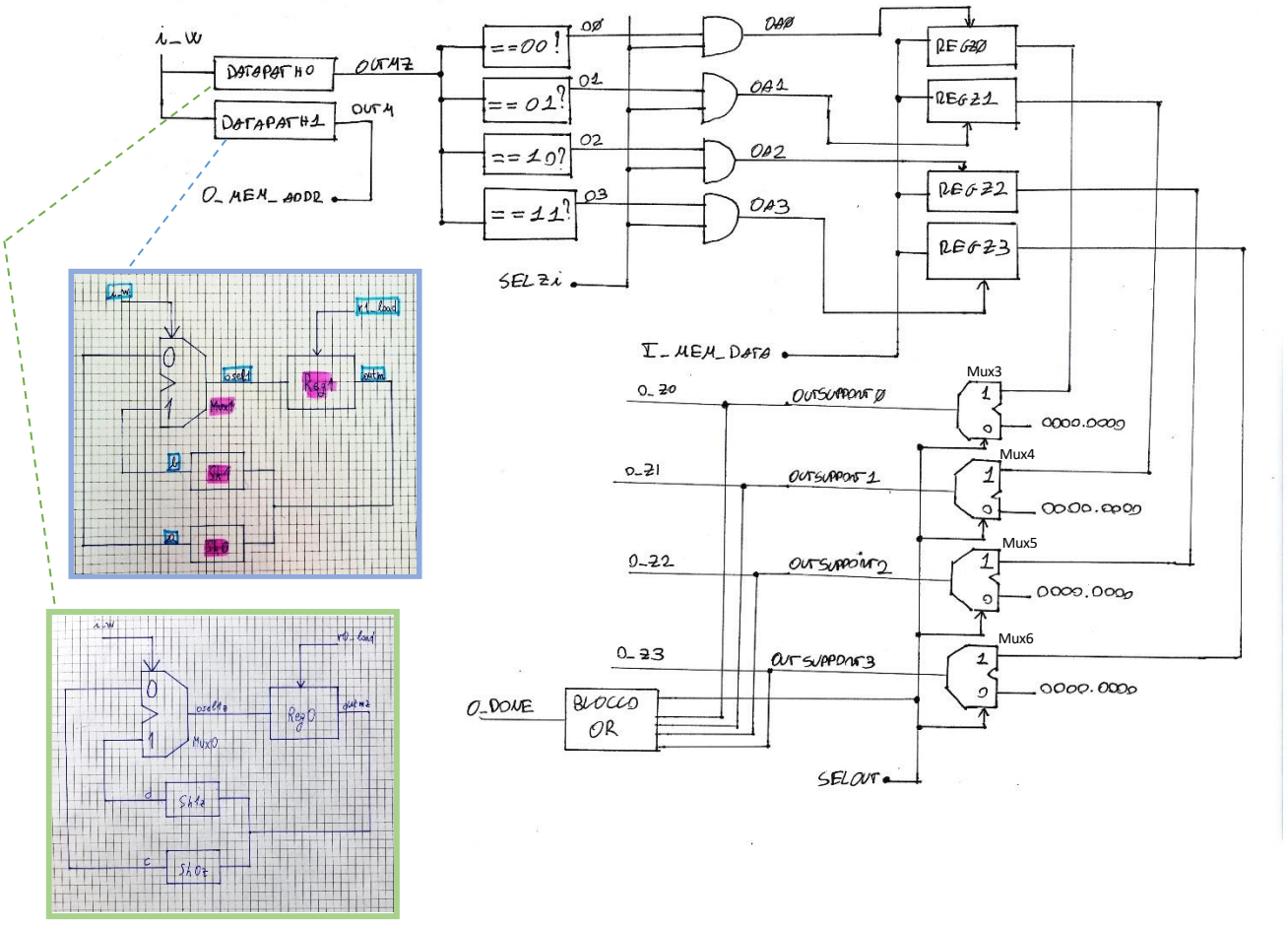
Architettura

Componenti

In questa sezione della relazione analizzeremo in modo esaustivo i **vari componenti** della rete logica considerandoli uno per uno.



(illustrazione generale del progetto, compresa di tutti i segnali aggiuntivi).



Segnali

Oltre ai segnali necessari da specifica, ne abbiamo introdotti di nuovi. Si possono notare qui sopra alcuni segnali che uscendo dalla macchina a stati entrano nel datapath; questi servono a pilotare il funzionamento del datapath. Nello specifico:

- *Load*: segnale di tipo std_logic, utilizzato per pilotare la scrittura sul registro *Reg1*
- *Loadz*: segnale di tipo std_logic, utilizzato per pilotare la scrittura sul registro *Reg0*
- *I_rstManual:* segnale di tipo std_logic. Questo segnale permette di resettare (solo) i registri utilizzati per salvare il numero del canale di uscita e dell'indirizzo di memoria. Segnale sincrono al clock.
- *Selout:* segnale di tipo std_logic, è il selettore dei 4 multiplexer *Mux3, Mux4, Mux5, Mux6* utilizzati per immettere sul canale di uscita o "0000 0000" o il valore contenuto nei rispettivi 4 registri. Verrà utilizzato anche per impostare *o_done* a 1.
- *Selzi*: segnale di tipo std_logic. Questo segnale viene impostato a 1 quando è possibile salvare sui registri (*Regz0*, *Regz1*, *Regz2*, *Regz3)* il dato proveniente da memoria RAM

I **segnali obbligatori** sono i seguenti (e funzionano come da specifica):

- i_clk: segnale di clock in ingresso. Tutte le operazioni saranno svolte sul fronte alto del clock. L'unica eccezione si presenta quando *i_rst* passa da 0 a 1.
- i_start: segnale in ingresso che avvia effettivamente la macchina a stati. Oltre a questo ruolo sarà in ingresso al datapath per gestire l'acquisizione di *i_w.*
- i_w: segnale in ingresso con il contenuto informativo.
- i_rst: segnale di reset. Come già anticipato, una sua transizione da 0 a 1 causerà delle operazioni asincrone rispetto al "normale" svolgimento
- i_mem_data: segnale a 8 bit proveniente da memoria
- o_z0: segnale output corrispondente all'uscita del multiplexer *Mux3 (*std_logic_vector(7 downto 0)
- o_z1: segnale output corrispondente all'uscita del multiplexer *Mux4 (*std_logic_vector(7 downto 0)
- o_z2: segnale output corrispondente all'uscita del multiplexer *Mux5 (*std_logic_vector(7 downto 0)
- o_z3: segnale output corrispondente all'uscita del multiplexer *Mux6 (*std_logic_vector(7 downto 0)
- o_done: segnale in output, di tipo std_logic
- o_mem_addr: segnale in output, di tipo std_logic_vector(15 downto 0), corrispondente all'indirizzo di memoria in cui andare a prelevare il dato. Questo segnale è uguale all'uscita del registro *Reg1*
- o_mem_en: segnale in output, di tipo std_logic, che attiva la lettura da memoria
- o_mem_we: segnale output, di tipo std_logic, che attiva la scrittura in memoria

Infine, sono presenti i segnali interni al solo datapath. Non tutti questi segnali però meritano una spiegazione dettagliata, essendo semplicemente dei "bus" tra due componenti. Alcuni di essi però la necessitano.

- oa0, oa1, oa2, oa3: segnali di tipo std_logic con tutti lo stesso compito. Rappresentano l'uscita dell'and tra l'if e *selzi.* L'if controlla se il canale di uscita in cui scriverò il dato proveniente da memoria è quello sul quale sto svolgendo il controllo (00, 01, 10, 11). *Selzi* invece è il segnale impostato a 1 nel momento in cui la macchina è pronta alla lettura dal segnale *i_mem_data.*

Scelte implementative

I vari componenti del datapath sono implementati seguendo le linee guida date a lezione e scritte nelle slide. Ci sono però alcune differenze implementative che necessitano di una spiegazione:

- *Reg0:* il registro è incaricato del salvataggio dei primi due bit provenienti da *i_w*.
  Ha due differenze rispetto a quelli visti in classe: possono essere resettati (impostati a 0) anche mediante l'uso del segnale *i_rstManual,* oltre a *i_rst.* Il motivo è il seguente: una volta che finisco una esecuzione (dopo *o_done*=1) è necessario immettere nei registri il valore "0" per il corretto funzionamento del blocco che si occupa del salvataggio dei bit. Tramite *i_rstManual* semplifico questa operazione.
  La seconda particolarità è il segnale che pilota la scrittura sul registro, ossia *input. Input* è l'and tra *loadz, i_rst* e *i_start*. Le motivazioni sono molteplici e verranno descritte successivamente nella macchina a stati.

```
begin
    input <= loadz and i_start and not i_rst;
    --primo blocco per salvataggio del canale di uscita
    process(i_clk, i_rst)
    begin
        if(i_rst = '1' or i_rstManual = '1')then
            outmz <= "00";
        elsif rising_edge(i_clk)then
            if(input='1')then
                outmz <= osellz;
            end if;
        end if;
    end process;
```

- *Reg1*: questo registro a 16 bit si occupa di memorizzare l'indirizzo di memoria (*o_mem_addr*). Anche in questo caso è presente *i_rstManual.* La motivazione è la stessa di *Reg0.* Il segnale pilota della scrittura sul registro è l'and tra *i_start* e load. La motivazione precisa sarà vista successivamente, nella descrizione della macchina a stati.

```
process(i_clk, i_rst)
begin
    if(i_rst = '1' or i_rstManual = '1')then
        outm <= "0000000000000000";
    elsif rising_edge(i_clk)then
        if(load = '1' and i_start='1')then
            outm <= osell;
        end if;
    end if;
end process;
```

Reg0 e *Reg1* sono asincroni. A causa della presenza della presenza del segnale *i_rst* nella sentivitity list è possibile attivare i registri anche non nei fronti bassi del clock.

- Blocchi per lo shift: è qui sotto riportato il codice che realizza lo shift del contenuto di *Reg0*. Abbiamo preferito non utilizzare istruzioni già esistenti come "shift_left(...)" o "sll".

```
oshift0z <=  outmz(0 downto 0) & '0';
oshift1z <=  outmz(0 downto 0) & '1';
```

  Stessa implementazione, ma su 16 bit, per lo shift del registro che si occupa del salvataggio dell'indirizzo di memoria.
- Gestione di *o_done* (blocco or nel disegno della pagina prima): l'uscita di *o_done* ci ha creato non pochi problemi. In un primo momento *o_done* era posto uguale a *selout*, in questo modo pilotavo contemporaneamente la scelta dell'uscita dei multiplexer *Mux3, Mux4, Mux5, Mux6* e di *o_done*. Banalmente quando *selout* era 0 *o_done* era ovviamente 0 e i multiplexer selezionavano l'uscita con "0000.0000". Quando *selout* era 1 *o_done* era 1 e i multiplexer selezionavano come uscita l'uscita dei registri *Regz0*, *Regz1*, *Regz2*, *Regz3.* Questa implementazione però non funzionava. Nello specifico era presente un istante di tempo in cui *o_done* era tornato ad essere 0, mentre le uscite dei multiplexer contenevano ancora i valori dei registri, come se ci fosse un ritardo nella commutazione.
  La prima ipotesi fu che la commutazione di o_z0, o_z1, o_z2 e o_z3 fosse più lenta perché erano presenti più bit da cambiare. Abbiamo quindi fatto variare anche *o_done* su 8 bit, ma senza ottenere miglioramenti.
  Soluzione e conclusione: dopo vari tentavi ci siamo resi conto che la scelta più semplice per risolvere il problema era quella di far tornare *o_done* a 0 solo dopo essere completamente certi che anche le uscite o_zi (i ∈ {0,1,2,3}) erano tornare 0. Ed ecco spiegati i 33 or presenti.
  Abbiamo pensato a soluzioni più efficienti, cercando di raggiungere lo stesso risultato senza dover aggiungere così tanti componenti, ma perdevamo coerenza tra quello che disegnavamo e la sintesi. Definendo controlli o process ad "alto livello" la macchina funzionava ma i componenti sintetizzati non erano più quelli previsti/disegnati da noi.

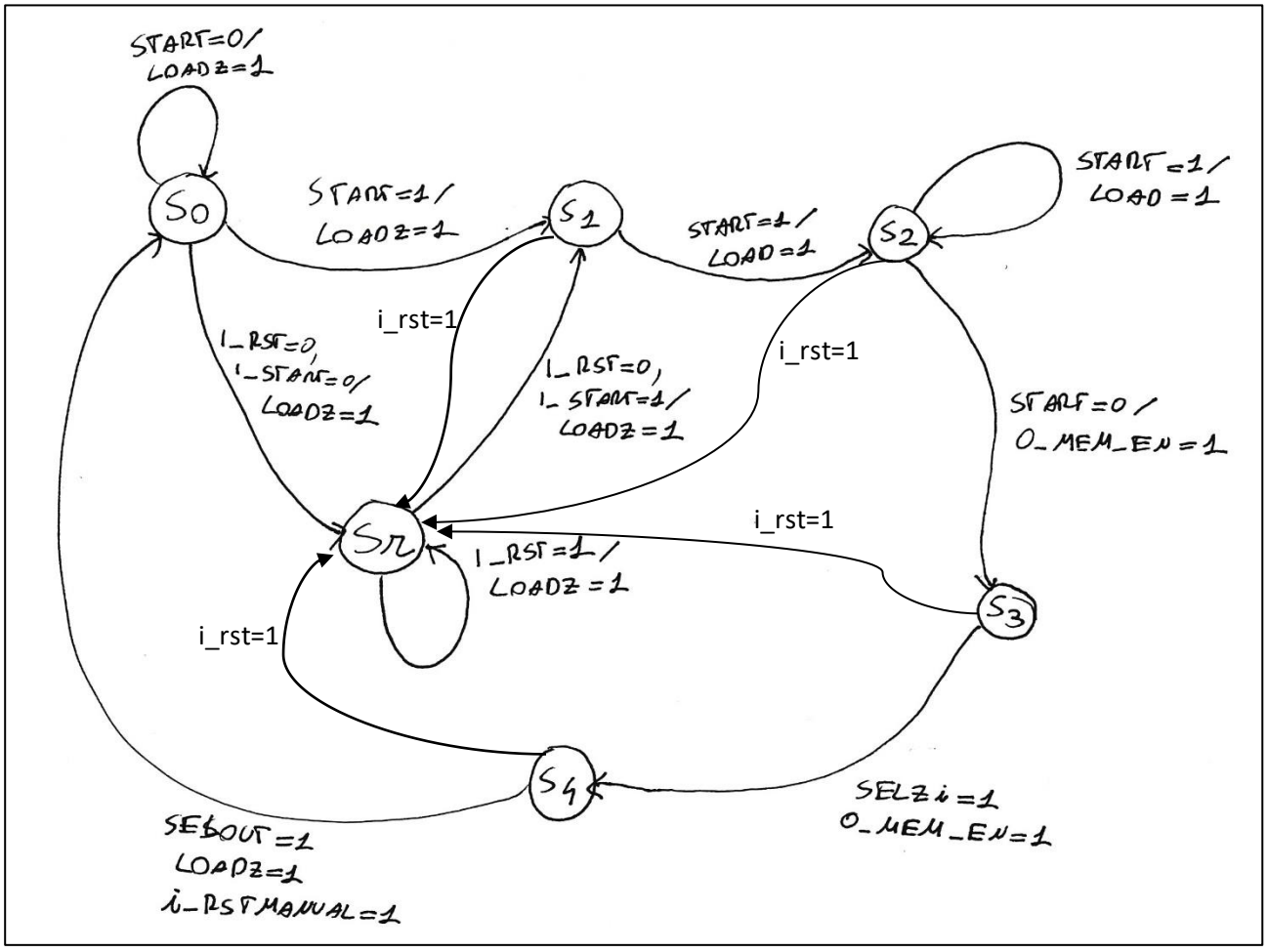
Macchina a stati

L'automa scelto per la gestione delle operazioni è una macchina di Mealy. Abbiamo preferito Mealy piuttosto che Moore per comodità di implementazione. Sono presenti 6 stati e 10 transizioni. Mi focalizzerò sulla descrizione delle possibili transizioni.

Il ruolo dell'automa è quello di pilotare il funzionamento del datapath e di generare segnali che saranno visibili direttamente in uscita. Si preoccuperà di gestire i segnali: load, loadz, selzi, i_rstManual, selout per il datapath; o_mem_en e o_mem_we in uscita.

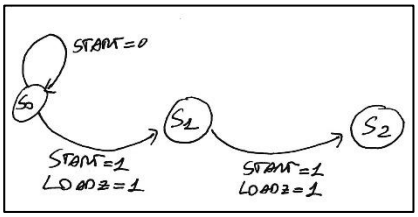
Nota 1: nel disegno della macchina a stati sono presenti solo i segnali che nelle transizioni assumono valore 1, quelli non presenti sono 0.

Nota 2: nelle transizioni ciò che viene prima di "/" è il segnale in input, i segnali di seguito (a capo) sono quelli impostati dalla macchina a stati (output). Nel caso in cui non sia presente "/" le transizioni sono obbligate.



Transizioni:

- S0 → S0: con i_*strat*=0 ho solo *loadz*=1, questa scelta implementativa è stata necessaria a causa del ritardo presente nelle transizioni da un valore ad un altro. Come promesso nel paragrafo "scelte implementative" è necessaria una spiegazione più approfondita del segnale pilota *input*, che svolgerò di seguito.



Durante la creazione del progetto, una volta definito il datapath e la macchina a stati, ci siamo resi conto che c'era qualcosa che non

andava. Il segnale *loadz*, che precedentemente veniva posto a 1 nelle transizioni *s0→s1* e *s1→s2*, e che non era messo a and con il segnale *i_start,* agiva con un ciclo di ritardo.

Nello specifico abbiamo notato che impostando *loadz*=1 nello stesso ciclo in cui la macchina a stati si rendeva conto che *i_start* era diventato 1 non permetteva di svolgere la scrittura nel registro *Reg0* istantaneamente.

La **conclusione** che abbiamo tratto è che siccome sul fronte di salita del clock tutti i process vengono aggiornati assieme, allora contemporaneamente *loadz* diventa 1 e viene eseguito l'if "**elsif loadz='1' then outmz <= osel1z**" nel process del *Reg0*. P1: Essendo le istruzioni eseguite simultaneamente, *loadz* non è ancora 1 quando viene eseguito l'if e quindi non viene svolta la scrittura nel registro *Reg0.* P2: Questo problema si ripresenta anche nello spegnimento del segnale *loadz*. Infatti, nel ciclo in cui imposto *loadz*=0 verrà effettuata un'altra scrittura sul registro

Nota: per riportare nelle spiegazioni delle transizioni dove si sono presentati dei problemi ho contraddistinto il problema dell'accensione (segnale = 1) come P1, il problema dello spegnimento (segnale = 0) come P2.

Soluzione: questi problemi si estendevano ovviamente al Reg1 e la soluzione è la stessa. I segnali *load* e *loadz* sono messi ad and con *i_start* e si attiveranno e/o si disattiveranno con almeno un ciclo di anticipo, a seconda dei casi. Saranno offerte spiegazioni più esaustive nell'analisi di ogni singola transizione.

Continuando quindi il discorso sulla transizione *S0→S0* si può capire finalmente perché *loadz=1*-P1. In questo modo, quando *i_start* diventerà 1 (sul fronte basso del clock), al fronte alto potrò già caricare sul *Reg0* il valore pilotato da *i_w*.

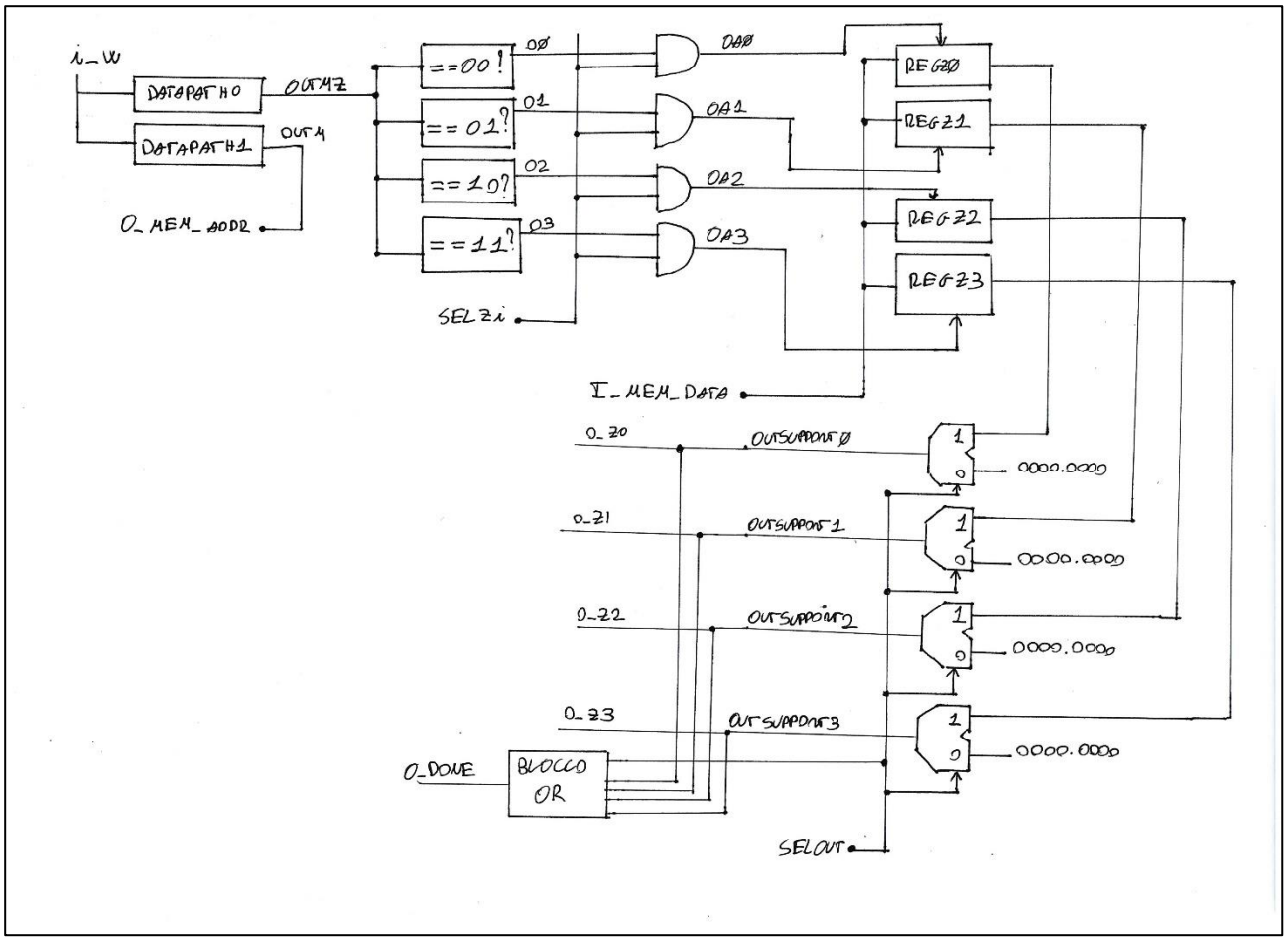
- S0 → S1: con i_start = 1 passo allo stato S1 e come detto al punto prima eseguo la prima lettura.
- S1 → S2: con i_start=1 mi sposto verso S2. Porto il segnale *loadz* a 0 ma, come descritto in P2, viene effettuata ancora una scrittura in registro (*Reg0* vede ancora 1), necessaria per salvare il secondo bit. Metto *load* uguale a 1, in anticipo di un clock – P1, così da essere pronto al ciclo successivo quando sarà necessario leggere i bit.
- S2 → S2: con *i_start*=1 continuo a rimanere in S2, a leggere l'ingresso *i_w* e a salvare i bit sul registro *Reg1*
- S2 → S3: quando *i_start* diventa 0 passo allo stato S3. In questo caso, se non ci fosse l'and tra *i_start* e *load* farei erroneamente ancora una scrittura sul registro *Reg1* – P2. Attivo anche il segnale *o_mem_en* per prelevare il valore da memoria.
- S3 → S4: in questo punto dell'evoluzione *i_start* non può che essere. *O_mem_en* è ancora 1, la motivazione è la presenza di un ritardo nel process della memoria RAM. Sulle slide è di 5ns, mentre nel test bench di 1 ns. Siccome 5ns erano sufficienti per far sì che la rete logica non funzionasse correttamente ho preferito predisporre due cicli per leggere dalla memoria RAM.
  Anche *selzi* viene impostato a 1, così che è possibile memorizzare sul corretto registro il dato proveniente da memoria.
- S4 → S0: in questa transizione attiverò il segnale *selout*, che permetterà di impostare *o_done*=1 e far vedere i valori sui 4 canali in uscita. *i_rstManual* sarà messo a 1 per poter resettare i registri *Reg0* e *Reg1.* Infine, *loadz* è impostato a 1 per essere pronto ad una istantanea ripartenza. Anche in questo caso, avendo sicuramente *i_start* ancora uguale a 0, non verrà caricato alcun valore sul *Reg0.*

Stato di reset Sr: le transizioni che coinvolgono lo stato Sr (stato di reset) possono essere descritte tutte assieme. In ogni fronte basso del ciclo di clock *i_rst* può diventare 1 e viene eseguita l'istruzione cur_state<= sr asincrona rispetto a tutte le altre. Mezzo ciclo di clock dopo, quando il clock sarà sul fronte di

salita, verrà ancora eseguita l'istruzione cur_state<=Sr. Questo stallo continua fin quando *i_rst* non torna 0 in cui si va effettivamente a valutare la prossima transizione, uscendo dallo stato di reset.

Funzionamento

Il componente hardware da noi realizzato è diviso in tre blocchi fondamentali.



I primi due hanno la stessa struttura, ma lavorano su sequenze di bit differenti. Nello specifico, il Datapath0 legge i primi due bit di intestazione, utili a identificare il canale Zi di uscita, mentre il Datapath1 effettua la lettura dei seguenti 0-16 bit, che rappresentano l'indirizzo di memoria a cui accedere per prendere il dato da portare in uscita sul registro selezionato.

Datapath1: La struttura prevede i seguenti componenti:

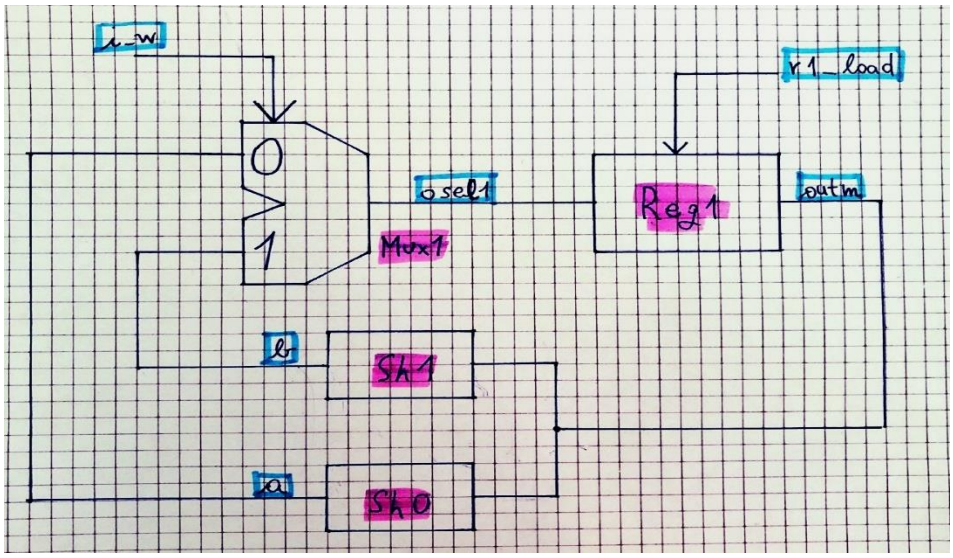
- 1 multiplexer (*Mux1*)
- 1 registro (*Reg1*)
- 2 blocchi che eseguono uno shift a sinistra (*Sh0*, *Sh1*)

che funzionano in questo modo:

Nella fase iniziale (o al reset) il contenuto di *Reg1* è 0000000000000000, tale valore viene shiftato a sinistra di 1 bit e viene aggiunto a destra sia 0 che 1 (parallelamente) generando rispettivamente le uscite *a* e *b*, grazie ai blocchi *Sh0* e *Sh1*.

Questi due segnali andranno in ingresso a *Mux1*, che ha come segnale di selezione *i_w*, che porterà in uscita da *Mux1* il segnale *a* quando leggo *0* da *i_w* e il segnale *b* quando leggo *1*.

In tal modo, ogni volta che leggeremo un bit da i_w, esso verrà aggiunto nella posizione meno significativa dell'indirizzo "parziale" che stiamo salvando.



(datapath1, per la lettura dei bit di indirizzo.)

Per una migliore comprensione del funzionamento, riportiamo un **esempio** di seguito:

Fase iniziale con *Reg1* = 0000000000000000;

i_w = 10;

Il contenuto di *Reg1* viene shiftato come descritto, ottenendo:

a = 0000000000000000, *b* = 0000000000000001;

Adesso leggeremo il bit *1* da *i_w*, che andrà a selezionare in uscita da *Mux1* il segnale *b*;

Questo sarà salvato in *Reg1*, verrà shiftato producendo:

a = 0000000000000010, *b* = 0000000000000011;

Successivamente, leggendo il bit *0* da *i_w*, verrà selezionato in uscita da *Mux1* il segnale *a*;

Il procedimento di lettura di *i_w* proseguirà ciclicamente come mostrato in figura fino a quando il segnale *START* è alto (=1), terminerà perciò con *START* = 0.

La **lettura dei primi due bit di intestazione** avviene in maniera analoga:

Inizialmente (o al reset) il contenuto di *Reg0* è 00.

Questo valore sarà shiftato a sinistra di un bit e verranno aggiunti a sinistra uno *0* e un *1*, sempre separatamente, portando in uscita (rispettivamente) i segnali *c* e *d*.

Questi andranno in ingresso al *Mux0*, che ha come segnale di selezione *i_w*, il segnale selezionato andrà in *Reg0*, sarà shiftato una seconda (e ultima) volta e portato nuovamente in ingresso su *Mux0*.

A questo punto l'ultimo bit di intestazione verrà letto da *i_w* e determinerà il segnale che salveremo su *Reg0*, identificando uno dei quattro canali d'uscita (Z0, Z1, Z2, Z3).

Il segnale outmz va in ingresso a **quattro blocchi differenti che selezionano il canale di uscita** (confrontando outmz con 00, 01, 10, 11). Le uscite di questi blocchi entrano in ingresso ognuna a una porta *AND*, insieme a un aggiuntivo segnale, chiamato *selzi*. Questo meccanismo ci assicura che venga eseguito il *load* su uno solo dei quattro registri *Regz0*, *Regz1*, *Regz2*, *Regz3*.

La scrittura sui canali di uscita è regolata da quattro multiplexer (*Mux3, Mux4, Mux5, Mux6)*, che hanno come segnale di selezione *selout*, il quale, se uguale a 0, porterà in uscita 00000000, mentre, se uguale a 1,

porterà in uscita il valore contenuto nel rispettivo registro. In questo modo le uscite saranno aggiornate correttamente.

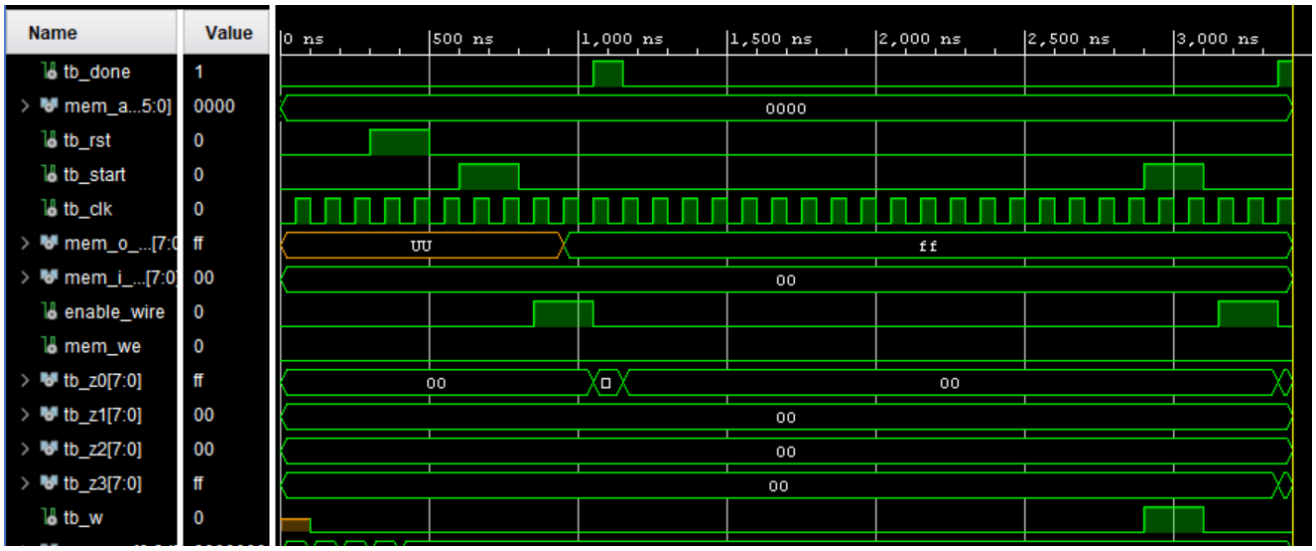
Risultati sperimentali

Test

Test1: test costruito sulla falsa riga di quello fornito. In questo test andiamo a verificare il caso limite in cui il valore da prelevare da memoria è contenuto nell'indirizzo di memoria 0000.0000.0000.0000 e quindi il segnale *i_strat* rimarrà a 1 solo per 2 cicli. Farò due esecuzioni:

1) L'ingresso *i_w* è sempre 0, quindi salvo il valore contenuto all'indirizzo 0 sul canale di uscita 0
2) L'ingresso i_w è uno per due cicli di clock, quindi salvo il valore contenuto all'indirizzo 0 sul canale di uscita

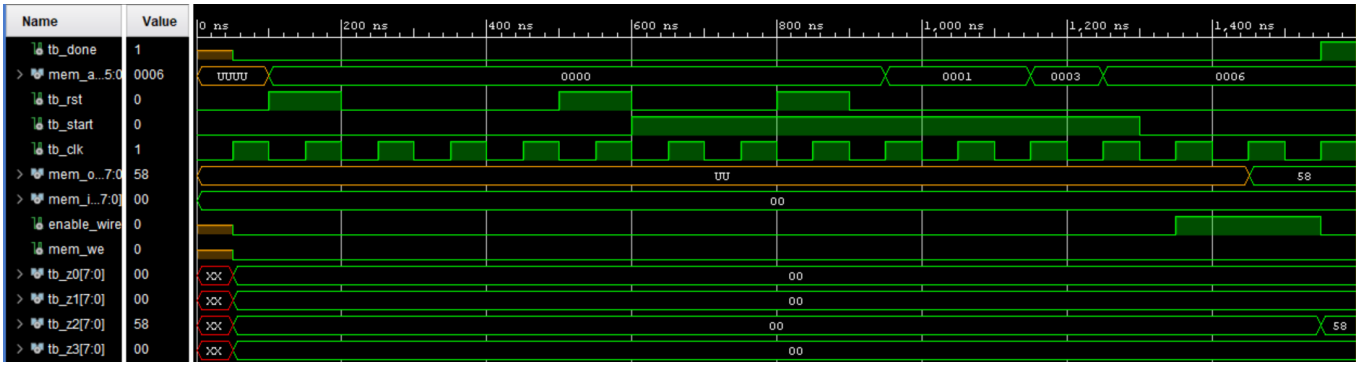
Simulation behavioral and simulation post-sintesi risultano uguali:



Test2: test utilizzato per studiare la prontezza della macchina a seguito del reset. Il segnale *i_rst* si attiva per un totale di due volte:

- Attivazione di *i_rst* con disattivazione dello stesso contemporaneamente all'attivazione di *i_start*
- Attivazione di *i_rst* con *i_start* che continua a rimanere attivo per tutto il tempo

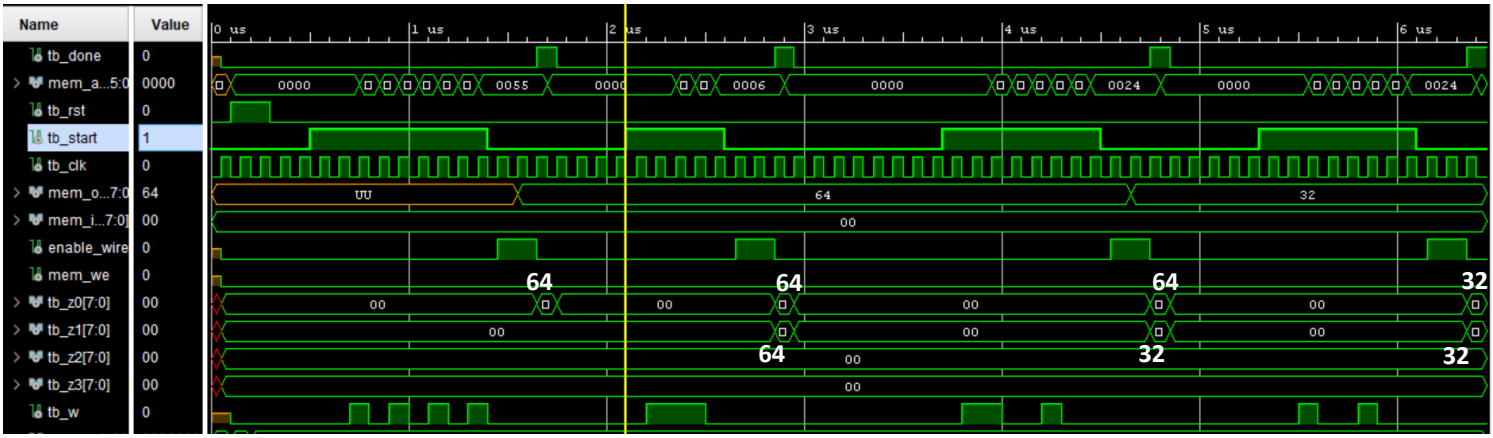
Infine, porto a termine la lettura e la stampa del valore.



Test3: test che prevede la riscrittura di un nuovo valore su un canale di uscita. Quindi si verifica che il valore salvato durante le esecuzioni precedenti venga effettivamente aggiornato. Procedimento:

- 1° esecuzione: salvo all'uscita 00 il valore 100
- 2° esecuzione: salvo all'uscita 01 il valore 100
- 3° esecuzione: salvo all'uscita 01 il valore 50
- 4° esecuzione: salvo all'uscita 00 il valore 50

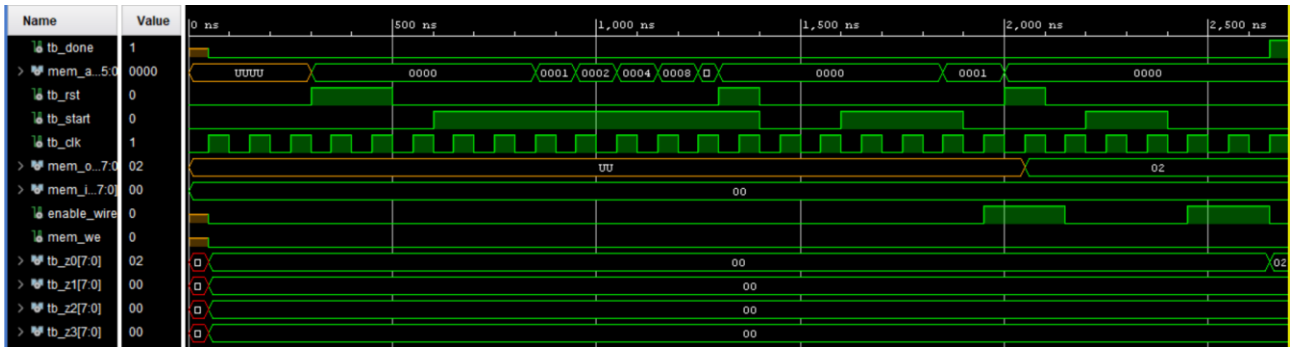
Si è quindi verificata la riscrittura sia immediatamente successiva, che in un qualunque ciclo successivo



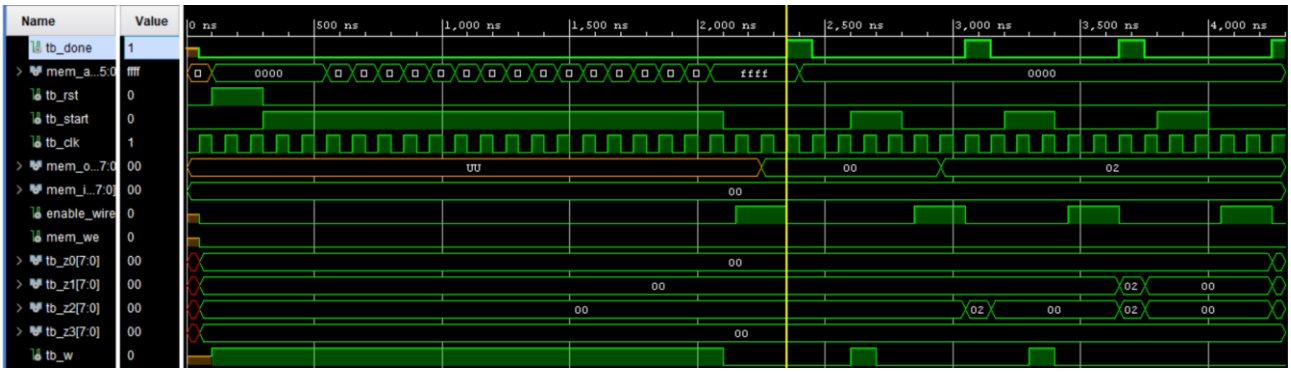
Purtroppo, non si vedono i valori, quindi li ho riportati a mano all'interno del file

Test4: in questo test testiamo l'efficacia del reset e non la sua prontezza. Attiveremo il reset due volte:

- Durante l'acquisizione delle informazioni, precisamente mentre si stanno leggendo i bit per identificare l'indirizzo di memoria
- Dopo l'acquisizione di tutti i dati, quando ormai *i_start=0* ma *o_done* non è ancora diventato 1



Test5: in questo caso di test vado a verificare che il riempimento di tutti i registri avvenga correttamente. Inoltre, verifico due casi limite: i_w è mantenuto attivo alto per tutto il primo periodo in cui *i_start=1*, scriverò quindi il valore contenuto nell'ultima cella di memoria (65535) sull'ultimo canele z3. Il secondo caso limite è che la cella di memoria 65535 contiene il valore 0, quindi non dovremmo notare nessun cambiamento nelle uscite anche se *o_done* è 1 (come si può notare sul puntatore giallo).

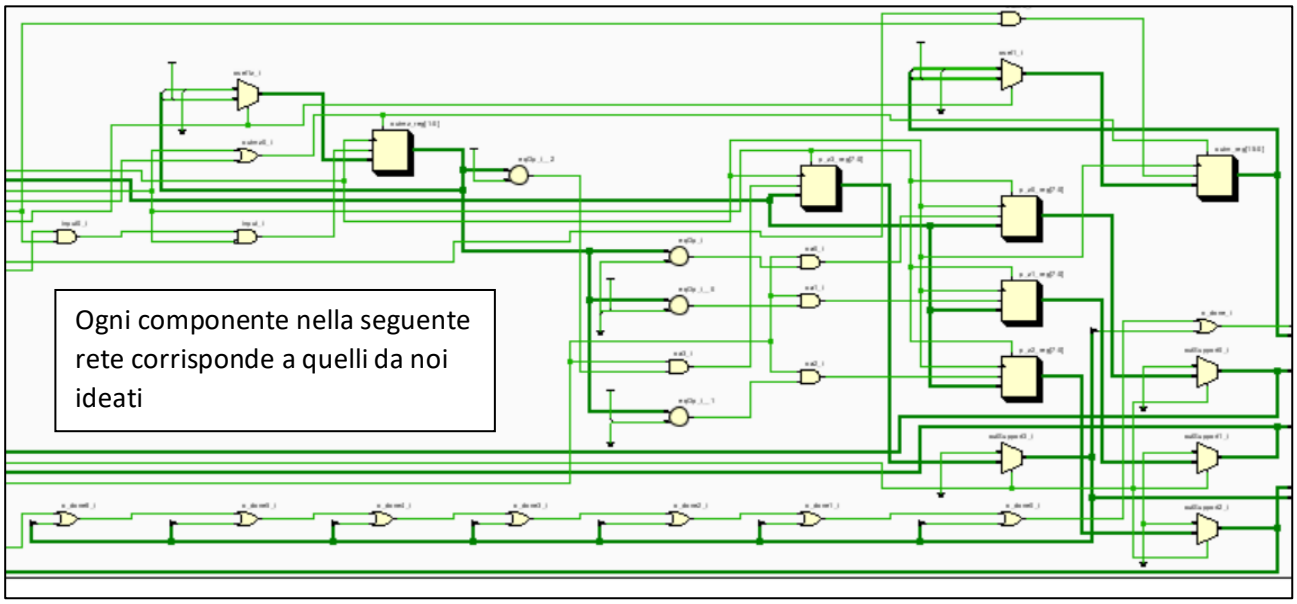


Report utilization

Tramite il comando report_utilization siamo in grado di verificare che non sono utilizzati latch come richiesto.

Successivamente abbiamo anche verificato che i componenti del datapath che abbiamo definito nei nostri disegni in fase di progettazione siano stati implementati correttamente. Questa operazione è stata svolta tramite l'elaborated design.

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	36	0	134600	0.03
LUT as Logic	36	0	134600	0.03
LUT as Memory	0	0	46200	0.00
Slice Registers	65	0	269200	0.02
Register as Flip Flop	65	0	269200	0.02
Register as Latch	0	0	269200	0.00
F7 Muxes	0	0	67300	0.00
F8 Muxes	0	0	33650	0.00



Report timing

Timing come richiesto

```
Timing Report

Slack (MET) :             7.443ns  (required time - arrival time)
```

Conclusioni:

Il componente da noi progettato è in grado di superare anche tutti i test forniti dai professori, sia quello inziale che gli ultimi rilasciati.

L'unica accortezza per il corretto funzionamento del componente è evitare tempi di risposta troppo lunghi della memoria RAM. Come abbiamo già accennato, l'istruzione "AFTER 1 ns" presente nei test bench ci ha portato a definire uno stato in più per la lettura. Nel caso questo tempo si dilatasse troppo ovviamente non

riuscirei ad ottenere il dato dalla memoria entro l’esecuzione dello stato che si occupa di salvarlo nei registri.