Relazione Prova Finale di Reti logiche A.A. 2023/24

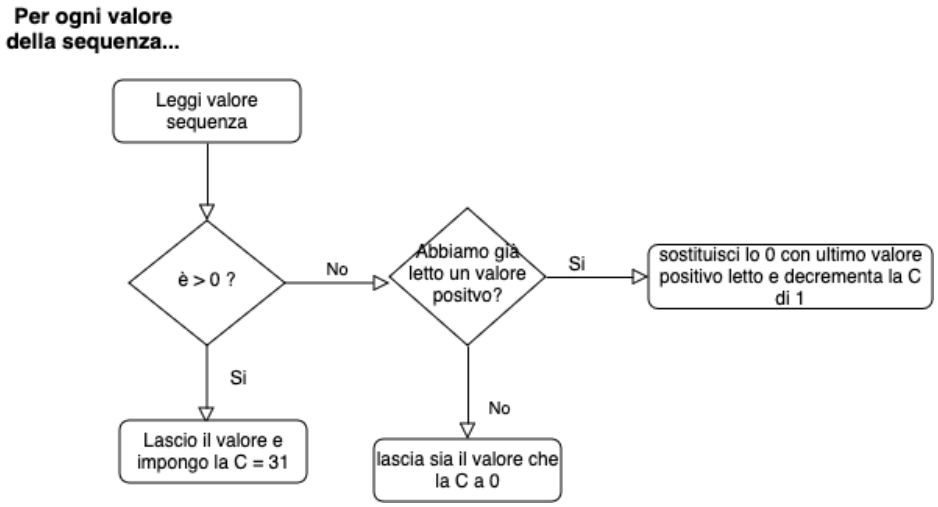
Emanuele Caruso (10788146) Ilaria Brunati (10766918)

0. Introduzione

Lo scopo del progetto è di implementare un componente hardware descritto in VHDL, in grado di elaborare una sequenza numerica lunga K letta da memoria RAM.

Ciascun valore è un byte di memoria, ed è associato ad una credibilità C (valore intero tra 0 e 31) che nella sequenza finale deve essere indicata nel byte immediatamente successivo. Il componente deve sostituire eventuali 0 presenti nella sequenza iniziale e associare a ciascun valore la giusta credibilità, secondo il seguente criterio.

Algoritmo:



Per chiarezza, riportiamo di seguito un esempio di sequenza iniziale e sequenza finale.

Esempio
In rosso i valori della credibilità

Sequenza iniziale

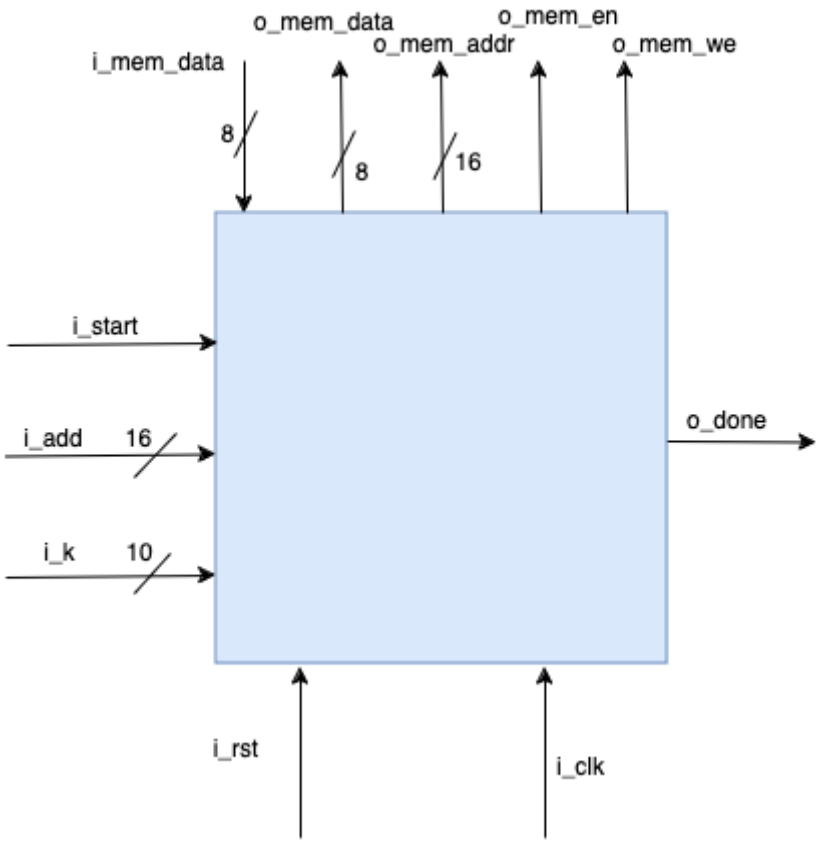
0	0	80	0	45	0	29	0	0	0

Sequenza finale

0	0	80	31	45	31	29	31	29	30

1. Architettura

Ad alto livello, il componente è rappresentato mediante il seguente schema.



Partendo dai segnali a sinistra, quelli di input, troviamo in ordine: i_start, i_add e i_k.

- i_start è il segnale necessario all'avvio del componente.
- i_add è il segnale che rappresenta l'indirizzo di memoria di partenza per la lettura dei valori nella RAM.
- i_k è il segnale che descrive la lunghezza della sequenza da leggere.

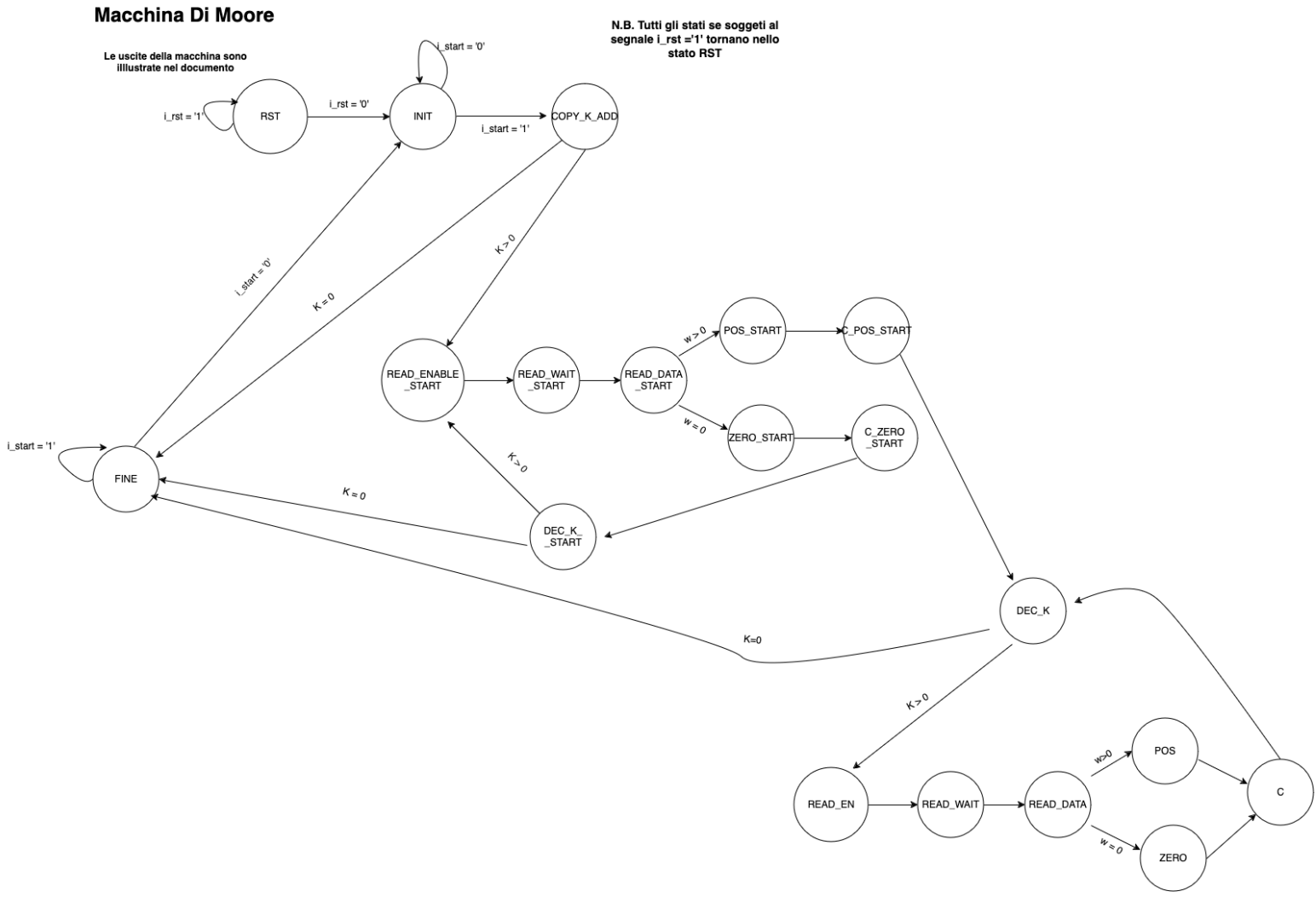
Il segnale di reset (i_rst) è asincrono, e serve per ripristinare la macchina mettendola in attesa del primo segnale di start. Inoltre, è necessario anche un segnale di clock (i_clk).

Sul lato superiore della figura sono presenti i segnali della memoria RAM: i_mem_data, o_mem_data, o_mem_addr, o_mem_en e o_mem_we.

- i_mem_data è il segnale che contiene il dato di input letto dalla memoria.
- o_mem_data è il segnale che contiene il valore da scrivere in memoria.
- o_mem_addr è il segnale che serve per indicare l'indirizzo da cui leggere/scrivere in memoria.
- I segnali o_mem_en e o_mem_we servono per attivare la scrittura e lettura della memoria. In particolare, quando si deve leggere, è necessario alzare ad 1 il segnale di o_mem_en, mentre per la scrittura è necessario che entrambi siano ad 1.

Infine, il segnale di o_done, mantenuto a 0 durante tutta l'esecuzione del programma, si alza ad 1 quando l'elaborazione della sequenza è completata.

Per quanto riguarda il codice vero e proprio, abbiamo scelto di strutturare il componente in un unico modulo: una macchina a stati finiti (FSM).
In particolare si tratta di una Macchina di Moore, in quanto l'uscita dipende esclusivamente dallo stato in cui si trova.



Abbiamo scelto di suddividere l'architettura della FSM in due process. Il primo è sequenziale ed ha come obiettivo quello di calcolare la funzione di stato prossimo. Il secondo, invece, è combinatorio e calcola la funzione di uscita.

Il process sequenziale, definito come *process(i_clk,i_rst)*, contiene nella *sensitivity list* i segnali di clock e di reset. Questo perché ad ogni fronte di salita del clock (i_*clk'event and i_clk = '1'*) è necessario calcolare il nuovo stato in cui si deve portare la FSM. È presente nella *sensitivity list* anche il segnale di reset asincrono, perché quando quest'ultimo viene portato a 1 la macchina a stati deve andare nello stato di RST.

Il secondo processo, *process(curr_state)*, presenta nella *sensitivity list* il segnale relativo allo stato corrente. Questo è essenziale poiché, al verificarsi di un cambiamento di stato, diventa necessario calcolare le uscite nel nuovo stato in cui la macchina a stati finiti si trova.

2. Descrizione stati FSM

In questa sezione, presentiamo le funzionalità degli stati della macchina.

- **RST**: è lo stato in cui mi devo trovare ogni qualvolta il segnale *i_rst* è ad "1".

- **INIT**: è lo stato preposto all'attesa dell'attivazione del segnale *i_start*, per avviare l'elaborazione della sequenza.

- **COPY_K_ADD**: è lo stato in cui vengono salvati i segnali di *i_k* e *i_addr*.

Per quanto riguarda gli stati successivi, quelli il cui nome termina con "START" servono per distinguere il caso in cui si debbano gestire degli zeri all'inizio della sequenza. Terminati gli zeri inziali, si passa al secondo gruppo di stati (a partire da DEC_K).

- **READ_ENABLE_START / READ_ENABLE**: è lo stato in cui il segnale *o_mem_en* è alzato a "1" per garantire la lettura dalla memoria. In aggiunta, associamo a *o_mem_addr* l'indirizzo di lettura corrente.

- **READ_WAIT_START / READ_WAIT**: è lo stato per attendere un ciclo di clock per completare la lettura.

- **READ_DATA_START / READ_DATA**: è lo stato in cui si salva il valore letto dal segnale *i_mem_data*.

- **POS_START / POS / ZERO_START**: questi stati servono per separare i casi in cui venga letto un valore 0 o un valore positivo, in quanto devono essere gestiti in maniera differente. In particolare viene calcolato, in base alle specifiche (riportate nel diagramma di flusso di sopra), il valore della credibilità.

- **ZERO**: rispetto a ZERO_START, questo stato deve anche abilitare la scrittura in memoria (*o_mem_en = "1", o_mem_we = "1", o_mem_addr* = indirizzo di memoria a cui scrivere) e mettere su *o_mem_data* l'ultimo valore positivo salvato, da scrivere in memoria.

- **C_POS_START / C_ZERO_START / C**: questi stati devono abilitare i segnali per la scrittura in memoria, *(o_mem_en ="1" e o_mem_we = "1"*, *o_mem_addr* = indirizzo di memoria a cui scrivere) e mettere su *o_mem_data* il valore della credibilità precedentemente calcolato.

- **DEC_K_START / DEC_K**: sono gli stati preposti al decremento di k di un'unità al termine di ogni iterazione.

- **FINE**: Lo stato di fine è lo stato che segnala il completamento dell'elaborazione della sequenza e che alza il segnale di *o_done* ad "1". Qualora, venisse dato il segnale di start a 0, riporta la sequenza allo stato di INIT per ripartire con un'ulteriore sequenza.

3. Sintesi

Di seguito riportiamo i risultati della sintesi del nostro componente.

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs*	116	0	0	8000	1.45
LUT as Logic	116	0	0	8000	1.45
LUT as Memory	0	0	0	5000	0.00
Slice Registers	55	0	0	16000	0.34
Register as Flip Flop	55	0	0	16000	0.34
Register as Latch	0	0	0	16000	0.00
F7 Muxes	0	0	0	7300	0.00
F8 Muxes	0	0	0	3650	0.00

Come si evince dalla tabella, la sintesi produce 0 latch. Per evitare di generarli, all'interno del nostro codice abbiamo adottato la seguente strategia.

Sono stati definiti i segnali "curr" e "next" per ognuno dei dati da memorizzare:
- store_value (l'ultimo valore POSITIVO letto)
- C (credibilità)
- read_value (l'ultimo valore letto dalla memoria)
- addr (per memorizzare l'indirizzo da cui leggere progressivamente i dati in memoria)
- K (decrementato a ogni iterazione)

Per tutti questi segnali, si salva il valore corrente sul corrispettivo "curr", mentre sul "next" si calcolano i nuovi valori.

Ad ogni ciclo di clock, il valore di next viene assegnato al curr per aggiornare correttamente i dati.

4. Simulazioni

Per garantire una completa copertura dei diversi scenari e dei casi limite, abbiamo ampliato il nostro processo di testing oltre al test bench standard fornito. Abbiamo creato 6 test aggiuntivi, mirati a esaminare e affrontare i vari casi limite a cui abbiamo pensato.

In particolare, il test bench 1 e il test bench 2, sono stati progettati per stimolare gli stati specifici della FSM etichettati come "START".

(i) TEST BENCH 1: è una sequenza di tutti 0 in modo tale da testare che la sequenza di input rimanga invariata rispetto all'output.

```
constant SCENARIO_LENGTH : integer := 5;
type scenario_type is array (0 to SCENARIO_LENGTH*2-1) of integer;

signal scenario_input : scenario_type := (0, 0,  0, 0,  0,  0,  0,  0,  0,  0);
signal scenario_full  : scenario_type := (0, 0,  0, 0,  0,  0,  0,  0,  0,  0);
```

(ii) TEST BENCH 2: è una sequenza che serve per testare che l'algoritmo ignori la prima sequenza di zeri, come richiesto dalla specifica, e che poi sia in grado di tradurre correttamente il resto dell'input fornito.

```
constant SCENARIO_LENGTH : integer := 8;
type scenario_type is array (0 to SCENARIO_LENGTH*2-1) of integer;

signal scenario_input : scenario_type := (0, 0,  0, 0, 6,  0,  9,  0,  12,  0,  0, 0,  1, 0,  20,  0);
signal scenario_full  : scenario_type := (0, 0,  0, 0, 6,  31, 9,  31, 12,  31, 12, 30, 1, 31, 20, 31);
```

(iii) TEST BENCH 3: la sequenza in questione ha lunghezza nulla, in quanto l'obiettivo è quello di verificare che immediatamente dopo la lettura di K la FSM raggiunga lo stato di FINE.

```
constant SCENARIO_LENGTH : integer := 0;
```

(iv) TEST BENCH 4: Questa è una sequenza che inizia con numero positivo seguito da 36 zeri. L'obiettivo, in questo caso, è di verificare che il valore della Credibilità C, una volta arrivata a 0, rimanga costantemente nulla senza assumere valori negativi dopo la lettura di una sequenza di almeno 31 zeri.

```
signal scenario_input : scenario_type := (64, 0,  0,  0,  0,  0,  0,  0,  0,  0,     0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0);
signal scenario_full  : scenario_type := (64, 31, 64, 30, 64, 29, 64, 28, 64, 27, ... 64, 2,  64, 1,  64, 0,  64, 0,  64, 0,  64, 0,  64, 0,  64, 0);
```

(v) TEST BENCH 5: Il test è stato sviluppato per assicurare il corretto funzionamento dello stato di RESET del componente. L'obiettivo principale è garantire che, in qualsiasi punto dell'esecuzione del componente, quando il segnale di reset (i_*rst*) viene portato a"1", la macchina raggiunge immediatamente lo stato di RESET (RST) e interrompe la procedura in corso.

Per raggiungere questo obiettivo, abbiamo inserito il segnale di reset (*i_rst*) nel test bench. Questo segnale viene attivato dopo 300 nanosecondi dall'inizio dell'esecuzione del test.

Quando il segnale di reset viene portato nuovamente allo stato logico "0", ci aspettiamo che l'elaborazione riparta dall'inizio con una nuova sequenza e prosegua correttamente fino al termine.

```
tb_start <= '1';

--Reset dopo 300 ns
wait for 300 ns;
tb_rst <= '1';
wait for 50ns;
tb_add <= std_logic_vector(to_unsigned(SCENARIO_ADDRESS_2, 16));
tb_k   <= std_logic_vector(to_unsigned(SCENARIO_LENGTH_2, 10));
tb_rst <= '0';

--Start è già alto, l'elaborazione parte subito
while tb_done /= '1' loop
    wait until rising_edge(tb_clk);
end loop;
```

(vi) TEST BENCH 6: L'ultimo test è quello preposto alla multi-esecuzione. Infatti, vengono fornite due sequenze di input, a due indirizzi di memoria diversi.
Quando l'elaborazione della prima sequenza termina, il segnale di start viene rimesso a 0 e si ritorna nello stato di INIT, pronti per una nuova sequenza.

```
constant SCENARIO_LENGTH : integer := 12;
type scenario_type is array (0 to SCENARIO_LENGTH*2-1) of integer;

signal scenario_input: scenario_type := (90, 0, 91, 0,  120, 0,  33, 0,  1,  0, 2,  0, 3,  0, 4,  0, 5,  0, 6,  0, 7,  0, 8,  0);
signal scenario_full: scenario_type := (90, 31, 91, 31, 120, 31, 33, 31, 1, 31, 2, 31, 3, 31, 4, 31, 5, 31, 6, 31, 7, 31, 8, 31);

constant SCENARIO_ADDRESS : integer := 107;

constant SCENARIO_LENGTH_2 : integer := 4;
type scenario_type_2 is array (0 to SCENARIO_LENGTH_2*2-1) of integer;

signal scenario_input_2: scenario_type_2 := (0, 0, 0, 0, 70, 0, 0, 0);
signal scenario_full_2: scenario_type_2 := (0, 0, 0, 0, 70, 31, 70, 30);

constant SCENARIO_ADDRESS_2 : integer := 500;
```

Nel process *create_scenario*, una volta terminata la prima sequenza, ripartiamo con la seconda.

```
while tb_done /= '1' loop
    wait until rising_edge(tb_clk);
end loop;

wait for 50 ns;

tb_start <= '0';
--Torniamo in init

wait for 50 ns;

--Ripetiamo il test per lo scenario 2
memory_control <= '0';  -- Memory controlled by the testbench
```

Nel process *test_routine*, testiamo poi la correttezza di entrambe le sequenze.

5. Conclusione

Concludendo, il componente sviluppato è stato in grado di superare con successo tutti i test condotti e quello fornito dal Docente. È importante notare che i test sono stati eseguiti sia nella simulazione di Post-Sintesi (functional) sia in Behavorial, confermando la correttezza del componente. Pertanto, si può affermare di aver progettato un componente hardware che soddisfa pienamente le specifiche e che è in grado di gestire in modo accurato qualsiasi richiesta.