

POLITECNICO DI MILANO 1863



POLITECNICO
MILANO 1863

Relazione Progetto Reti Logiche

Aleandro Pagani

Giugno 2026

Matricola: 236992
Codice persona: 10893566

Professore: Fabio Salice

Indice

1	Introduzione	1
1.1	Specifiche generali	1
1.1.1	Descrizione	1
1.1.2	Funzionamento	2
1.2	Descrizione modulo	3
1.3	Descrizione memoria	4
1.3.1	Implementazione fornita	4
1.3.2	Layout di memoria	5
2	Architettura	6
2.1	Segnali interni	6
2.2	Processo sincrono	7
2.3	Processo combinatorio	8
2.3.1	Stati di gestione del protocollo	8
2.3.2	Operazione 00	9
2.3.3	Operazione 01	10
2.3.4	Operazione 10	11
2.3.5	Operazione 11	14
3	Risultati sperimentali	15
3.1	Report di sintesi	15
3.1.1	Risorse utilizzate	15
3.1.2	FSM codificata	16
3.1.3	Statistiche RTL	17
3.1.4	Timing	18
3.2	Report di simulazioni	19
3.2.1	Test bench 1	19
3.2.2	Test bench 2 (casi limite)	21
3.2.3	Test bench 3 (latenze)	21
4	Conclusioni	22

1 Introduzione

1.1 Specifiche generali

1.1.1 Descrizione

Il progetto consiste nel descrivere un modulo hardware che si interfaccia con una memoria in cui la prima cella contiene il contatore dei task e le celle successive la lista ordinata. Il modulo deve supportare quattro tipi di operazioni il cui codice identificativo è fornito in ingresso:

- **00**: Decrementa la priorità di tutti i task
- **01**: Rimuove il primo task della lista
- **10**: Aggiunge un nuovo task in lista
- **11**: Svuota completamente la lista

0	00000011
1	00111100
2	01010000
3	01111111
4	



0	00000011
1	00111101
2	01010000
3	01111111
4	

OP 00.

0	00000011
1	00111100
2	01010000
3	01111111
4	



0	00000010
1	01010000
2	01111111
3	01111111
4	

OP 01.

0	00000011
1	00111100
2	01010000
3	01111111
4	



0	00000100
1	00111100
2	01010000
3	10000001
4	01111111

OP 10 (inserisco task 100000 01).

0	00000011
1	00111100
2	01010000
3	01111111
4	



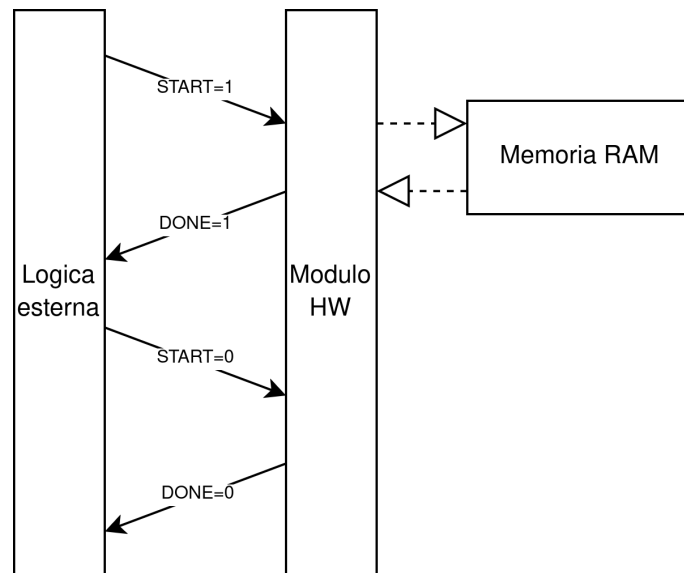
0	00000000
1	00111100
2	01010000
3	01111111
4	

OP 11.

1.1.2 Funzionamento

La sincronizzazione tra la logica esterna e il modulo hardware avviene tramite un protocollo di handshake **START-DONE**. Il funzionamento del modulo è scandito attraverso 5 fasi:

- **Inizializzazione:** Vengono forniti i dati e posto il segnale **START** a 1.
- **Elaborazione:** Il modulo avvia le operazioni e interagisce con la memoria.
- **Completamento:** Terminate le operazioni, il modulo imposta il segnale **DONE** a 1.
- **Riconoscimento:** La logica esterna pone il segnale **START** a 0.
- **Ripristino:** Il modulo riporta il segnale **DONE** a 0 e torna in attesa di nuove direttive.



1.2 Descrizione modulo

```
1  entity project_reti_logiche is
2    port (
3      i_clk: in std_logic;
4      i_rst: in std_logic;
5
6      i_start: in std_logic;
7      i_task_id: in std_logic_vector(5 downto 0);
8      i_task_priority : in std_logic_vector(1 downto 0);
9      i_op: in std_logic_vector(1 downto 0);
10
11     o_done: out std_logic;
12     o_task_id: out std_logic_vector(5 downto 0);
13
14     o_mem_addr : out std_logic_vector(15 downto 0);
15     i_mem_data : in std_logic_vector(7 downto 0);
16     o_mem_data : out std_logic_vector(7 downto 0);
17     o_mem_we: out std_logic;
18     o_mem_en: out std_logic
19   );
20 end project_reti_logiche;
```

- **i_clk**: Segnale di clock
- **i_rst**: Segnale di reset
- **i_start**: Segnale **START** di handshake
- **i_task_id**: Codice del task da inserire
- **i_task_priority**: Priorità del task da inserire
- **i_op**: ID dell'operazione da eseguire
- **o_done**: Segnale **DONE** di handshake
- **o_task_id**: Codice del task restituito dall'eliminazione
- **o_mem_addr**: Indirizzo di memoria con cui interagire
- **i_mem_data**: Informazioni lette dalla memoria
- **o_mem_data**: Informazioni da scrivere in memoria
- **o_mem_we**: Flag per abilitare la scrittura in memoria
- **o_mem_en**: Flag per abilitare la lettura in memoria

1.3 Descrizione memoria

1.3.1 Implementazione fornita

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
4
5 entity rams_sp_wf is
6 port(
7     clk    : in std_logic;
8     we     : in std_logic;
9     en     : in std_logic;
10    addr   : in std_logic_vector(15 downto 0);
11    di     : in std_logic_vector(7  downto 0);
12    do     : out std_logic_vector(7  downto 0)
13 );
14 end rams_sp_wf;
15
16 architecture syn of rams_sp_wf is
17 type ram_type is array (65535 downto 0) of std_logic_vector(7
18     downto 0);
19 signal RAM : ram_type;
20 begin
21     process(clk)
22     begin
23         if clk'event and clk = '1' then
24             if en = '1' then
25                 if we = '1' then
26                     RAM(conv_integer(addr)) <= di;
27                     do <= di; -- Write-First
28                 else
29                     do <= RAM(conv_integer(addr));
30                 end if;
31             end if;
32         end if;
33     end process;
34 end syn;
```

- **clk**: Segnale di clock
- **we**: Flag di scrittura
- **en**: Flag di lettura
- **addr**: Indirizzo con cui interagire
- **di**: Dati in ingresso
- **do**: Dati in uscita

1.3.2 Layout di memoria

La RAM indirizza 65536 celle da un byte (indirizzi a 16 bit); la lista ne utilizza al più 64: la prima (**0**) rappresenta il numero di task presenti, le successive contengono i task (al massimo 63, dato che gli ID sono a 6 bit e l'ID 0 è riservato).

Per ogni task, i primi 6 bit ne identificano il codice, mentre gli ultimi 2 la priorità.

0	00000011
1	00111100
2	01010000
3	01101111
...	...
63	11111111

In questo esempio abbiamo 3 task:

- **001111 00:** Task con codice 15 e priorità 0
- **010100 00:** Task con codice 20 e priorità 0
- **011011 11:** Task con codice 27 e priorità 3

Tutti gli altri elementi in memoria non vengono considerati validi.

2 Architettura

Data la dimensione ridotta del progetto, il componente è composto da un unico modulo con due processi: uno sincrono e uno combinatorio.

Nell'implementazione presentata è stata privilegiata la leggibilità rispetto all'efficienza e alla minimizzazione del numero di segnali.

2.1 Segnali interni

```
1 signal state: state_type;
2 signal next_state: state_type;
3
4 signal current_mem_addr: std_logic_vector(15 downto 0);
5 signal next_mem_addr: std_logic_vector(15 downto 0);
6
7 signal current_task_count: std_logic_vector(7 downto 0);
8 signal next_task_count: std_logic_vector(7 downto 0);
9
10 signal current_popped_id: std_logic_vector(5 downto 0);
11 signal next_popped_id: std_logic_vector(5 downto 0);
12
13 signal ctrl_done: std_logic;
14 signal ctrl_mem_en: std_logic;
15 signal ctrl_mem_we: std_logic;
16 signal ctrl_mem_data: std_logic_vector(7 downto 0);
17 signal ctrl_task_id: std_logic_vector(5 downto 0);
```

I registri interni sono organizzati in coppie **current_*/next_***: il processo combinatorio calcola il valore **next_***, il processo sincrono lo registra in **current_*** sul fronte di clock.

- **state**: Stato attuale dell'FSM
- **next_state**: Prossimo stato dell'FSM
- **current_mem_addr / next_mem_addr**: Indirizzo di memoria su cui la FSM sta lavorando (0 = contatore, 1..N = task)
- **current_task_count / next_task_count**: Copia locale del numero di task presenti in lista (cella 0 della memoria)
- **current_popped_id / next_popped_id**: ID estratto dall'ultima rimozione, presentato su **o_task_id** quando **DONE** viene alzato
- **ctrl_done**: Segnale di controllo di **DONE**
- **ctrl_mem_en**: Segnale per abilitare la lettura in memoria

- **ctrl_mem_we**: Segnale per abilitare la scrittura in memoria
- **ctrl_mem_data**: Dati da scrivere in memoria
- **ctrl_task_id**: Valore da presentare sull'uscita **o_task_id**

2.2 Processo sincrono

Questo processo registra sul fronte di salita del clock lo stato, i registri e tutte le uscite del modulo.

```

1 process(i_clk, i_rst)
2 begin
3     if i_rst = '1' then
4         state <= S_RESET;
5         o_done <= '1';
6         o_mem_en <= '0';
7         o_mem_we <= '0';
8     elsif rising_edge(i_clk) then
9         state <= next_state;
10        current_mem_addr <= next_mem_addr;
11        current_task_count <= next_task_count;
12        current_popped_id <= next_popped_id;
13
14        o_done <= ctrl_done;
15        o_mem_en <= ctrl_mem_en;
16        o_mem_we <= ctrl_mem_we;
17        o_mem_addr <= next_mem_addr;
18        o_mem_data <= ctrl_mem_data;
19        o_task_id <= ctrl_task_id;
20    end if;
21 end process;

```

Sul fronte di salita del clock i registri **current_*** vengono aggiornati con i corrispettivi valori **next_*** e, contemporaneamente, vengono scritti sui bus di uscita **o_*** i valori calcolati dal processo combinatorio.

Il reset, asincrono, porta la FSM in **S_RESET** con **DONE** a 1 (come richiesto dalla specifica) e disabilita l'accesso alla memoria: in questo modo il modulo non può produrre scritture involontarie mentre il reset è attivo, indipendentemente dall'istante in cui arriva.

Nota: **o_mem_addr** viene registrato da **next_mem_addr** (e non da **current_mem_addr**). In questo modo l'indirizzo arriva alla memoria con un ciclo di anticipo rispetto allo stato che ne consumerà il dato. Questa convenzione permette a ogni stato di richiedere una lettura e allo stato successivo (dopo un solo ciclo "cuscinetto") di usarne il risultato.

2.3 Processo combinatorio

Questo è il processo principale del modulo hardware: esso è responsabile della logica interna e della gestione degli stati FSM.

Vengono di seguito analizzati tutti gli stati presenti.

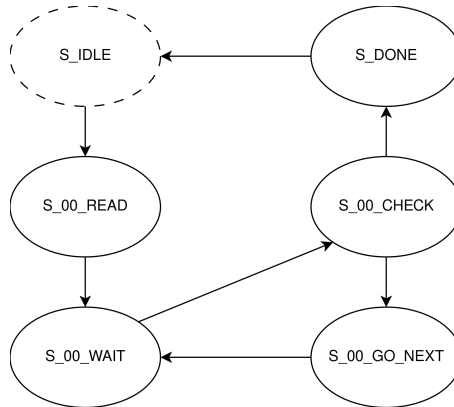
```
1  type state_type is (  
2      S_IDLE, S_RESET, S_DONE,  
3  
4      S_00_READ, S_00_WAIT, S_00_CHECK, S_00_GO_NEXT,  
5  
6      S_01_CHECK_NUMBER, S_01_WAIT, S_01_WRITE, S_01_CHECK_END,  
7          S_01_COPY, S_01_GO_NEXT, S_01_WAIT_FOR_COUNT,  
8  
9      S_10_PLACE_AT_START, S_10_WAIT_FOR_CHECK, S_10_CHECK_ID,  
10          S_10_WAIT, S_10_COMPARE, S_10_GO_NEXT, S_10_UPDATE_COUNT  
11          , S_10_WAIT_FOR_COUNT,  
12  
13      S_11_UPDATE_COUNT, S_11_WAIT_FOR_COUNT  
14  );
```

2.3.1 Stati di gestione del protocollo

- **S_IDLE**: Stato di attesa. Quando viene rilevato il segnale **START=1**, viene avviata l'operazione indicata dal codice **i_op**. Viene anche azzerato il registro **popped_id**, così **o_task_id** varrà 0 per ogni operazione diversa da una rimozione andata a buon fine.
- **S_RESET**: Stato di reset. Imposta il contatore nella prima cella di memoria a 0, invalidando tutte le altre, mentre **DONE** rimane a 1 come richiesto dalla specifica.
- **S_DONE**: Alza **DONE** e lo tiene a 1 finché **START** non torna a 0, come richiesto dal protocollo di handshake. Tutte le operazioni raggiungono questo stato un ciclo dopo il proprio stato terminale per permettere il completamento delle scritture in memoria e fornire i dati già aggiornati.

2.3.2 Operazione 00

Questa operazione decrementa la priorità di tutti i task presenti in lista. Se un task ha già la priorità minima **11**, questa non viene modificata.

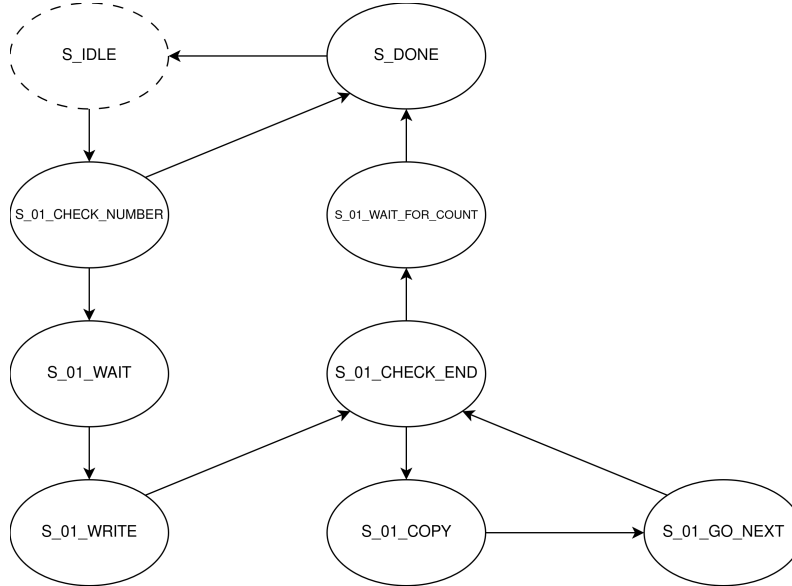


Gli stati presenti sono 4:

- **S_00_READ:** Seleziona il primo elemento in lista
- **S_00_WAIT:** Stato cuscinetto. Serve per dare tempo alla RAM di leggere il contenuto del task selezionato
- **S_00_CHECK:** Controlla se l'indirizzo selezionato è un task valido. In tal caso, se possibile, ne incrementa la priorità numerica di 1.
- **S_00_GO_NEXT:** Seleziona la cella di memoria successiva

2.3.3 Operazione 01

Questa operazione rimuove il primo task della lista e sposta in alto tutti i task sottostanti.



Gli stati presenti sono 7:

- **S_01_CHECK_NUMBER:** Se non ci sono task termina subito (**o_task_id** resterà **000000**). Altrimenti avvia la lettura del primo task.
- **S_01_WAIT:** Stato cuscinetto. Serve per dare tempo alla RAM di leggere il contenuto del task selezionato
- **S_01_WRITE:** Salva l'ID del primo task nel registro **popped_id** (verrà presentato su **o_task_id** in **S_DONE**) e avvia la lettura del task successivo.
- **S_01_CHECK_END:** Controlla se ha raggiunto la fine della lista. In tal caso scrive nella prima cella il valore del contatore decrementato di uno.
- **S_01_COPY:** Copia il contenuto della cella **n+1** dentro alla cella **n**.
- **S_01_GO_NEXT:** Passa al task successivo.
- **S_01_WAIT_FOR_COUNT:** Attende il commit della scrittura del contatore, poi transita in **S_DONE**.

2.3.4 Operazione 10

Questa operazione inserisce un nuovo task nella posizione corretta secondo la sua priorità, in coda ai task di pari priorità. L'operazione è divisa in due fasi:

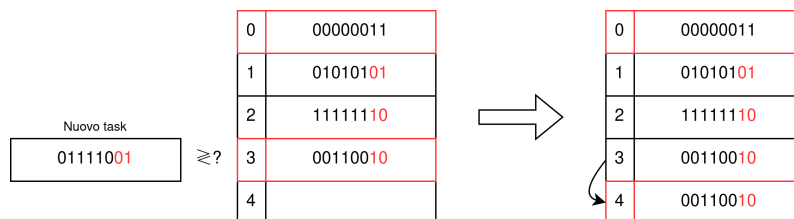
1. **Scansione duplicati:** la lista viene percorsa dall'ultimo task verso il primo; se viene trovato un task con lo stesso ID, l'inserimento viene ignorato (la specifica vieta la presenza di due task con lo stesso ID). Vengono inoltre scartate immediatamente le richieste con ID = 0.
2. **Inserimento dal fondo:** partendo dall'ultimo task e salendo, viene controllata la priorità del task corrente. Se questa risulta numericamente maggiore (gerarchicamente inferiore) rispetto a quella del task da inserire, la cella viene copiata in quella sottostante e il confronto prosegue verso l'alto. Contrariamente, se la priorità dovesse risultare minore o uguale, il nuovo task viene inserito nella cella sottostante: il confronto "minore o uguale" è ciò che colloca il nuovo task *dopo* quelli di pari priorità.

Si fa notare come il caso limite di lista piena non può verificarsi: con ID a 6 bit e ID = 0 escluso esistono al più 63 task distinti, e un 64-esimo inserimento sarebbe necessariamente un duplicato, quindi ignorato dalla fase di scansione.

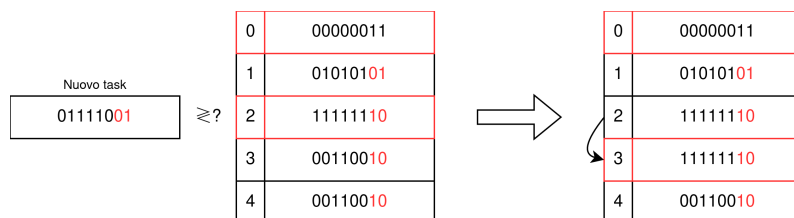
Di seguito viene illustrato un esempio di inserimento del task **011110 01**.

Nota: la scansione iniziale per identificare i duplicati viene ignorata nell'esempio.

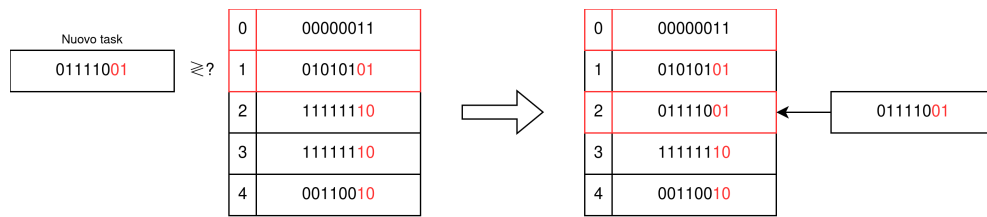
- L'ultimo task presenta una priorità gerarchicamente minore, di conseguenza viene copiato nella cella sottostante.



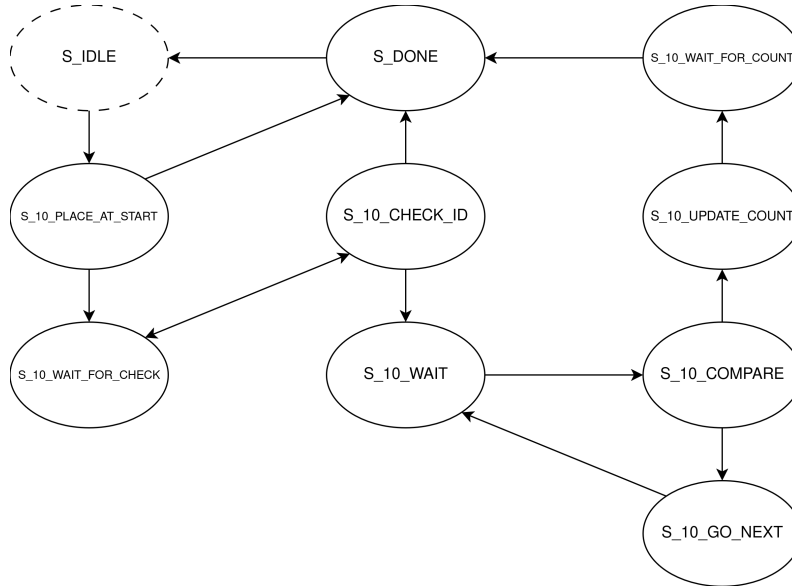
- Si passa quindi al task soprastante. Anche questo presenta una priorità inferiore e viene pertanto copiato nella cella sottostante.



- Il task nella cella 1 presenta invece una priorità maggiore o uguale: il nuovo task viene quindi inserito nella cella sottostante.



- Infine, viene aggiornato il contatore.

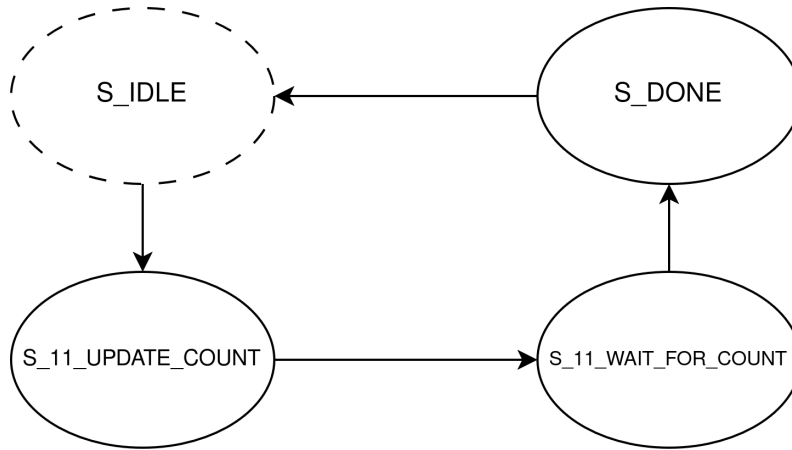


Gli stati presenti sono 8:

- **S_10_PLACE_AT_START:** Scarta le richieste con $ID = 0$ e avvia la scansione duplicati dall'ultimo task della lista.
- **S_10_WAIT_FOR_CHECK:** Stato cuscinetto della fase di scansione. Serve per dare tempo alla RAM di leggere il contenuto del task selezionato.
- **S_10_CHECK_ID:** Confronta l'ID del task letto con quello da inserire: se coincidono, l'operazione termina senza modifiche. Altrimenti se la scansione ha raggiunto la cima della lista, rilegge l'ultimo task e passa alla fase di inserimento. In caso contrario prosegue la scansione verso l'alto.
- **S_10_WAIT:** Stato cuscinetto della fase di inserimento. Serve per dare tempo alla RAM di leggere il contenuto del task selezionato.
- **S_10_COMPARE:** Verifica la condizione di inserimento del nuovo task. Se tutta la lista è stata sfogliata (o era vuota), il nuovo task viene scritto in testa. In caso contrario, se il task da inserire presenta una priorità numericamente maggiore o uguale rispetto al task corrente, viene copiato nella cella sottostante; se invece presenta una priorità numericamente minore, è il task corrente a venire scritto nella cella sottostante.
- **S_10_GO_NEXT:** Seleziona il task precedente nella lista, da confrontare al giro successivo.
- **S_10_UPDATE_COUNT:** Aggiorna il contatore con il numero totale di task presenti nella lista.
- **S_10_WAIT_FOR_COUNT:** Attende il commit della scrittura del contatore, poi transita in **S_DONE**.

2.3.5 Operazione 11

Questa operazione svuota completamente la lista impostando il contatore dei task a 0 (non è necessario resettare tutti i valori nella memoria).



Gli stati presenti sono 2:

- **S_11_UPDATE_COUNT**: Imposta il contatore a 0.
- **S_11_WAIT_FOR_COUNT**: Attende il commit della scrittura del contatore, poi transita in **S_DONE**.

3 Risultati sperimentali

3.1 Report di sintesi

3.1.1 Risorse utilizzate

La seguente tabella mostra le risorse hardware occupate dal design sul dispositivo target Artix-7 FPGA xc7a200tfbg484-1. L'utilizzo è estremamente contenuto: le Slice LUT occupano solo lo 0.13% delle risorse disponibili e i registri lo 0.03%. Inoltre, non è presente alcun latch: tutti gli elementi di memoria sono flip-flop.

Tabella 1: Risorse utilizzate

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs*	181	0	0	134600	0.13
LUT as Logic	181	0	0	134600	0.13
LUT as Memory	0	0	0	46200	0.00
Slice Registers	71	0	0	269200	0.03
Register as Flip Flop	71	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
Unique Control Sets	6	–	0	33650	0.02

3.1.2 FSM codificata

La tabella riporta la codifica degli stati FSM generata automaticamente dal tool di sintesi. Sono presenti in totale 24 stati. Vivado ha scelto una codifica **one-hot** (un flip-flop per stato, 24 bit complessivi). Per compattezza la tabella riporta, per ogni stato, l'indice del bit attivo della codifica one-hot e la codifica sequenziale a 5 bit di partenza.

Tabella 2: Codifica degli stati

Stato	One-hot (bit attivo)	Previous Encoding
s_reset	0	00001
s_done	1	00010
s_idle	2	00000
s_00_read	3	00011
s_00_wait	4	00100
s_00_check	5	00101
s_00_go_next	6	00110
s_01_check_number	7	00111
s_01_wait	8	01000
s_01_write	9	01001
s_01_check_end	10	01010
s_01_wait_for_count	11	01101
s_01_copy	12	01011
s_01_go_next	13	01100
s_10_place_at_start	14	01110
s_10_wait_for_check	15	01111
s_10_check_id	16	10000
s_10_wait	17	10001
s_10_compare	18	10010
s_10_update_count	19	10100
s_10_wait_for_count	20	10101
s_10_go_next	21	10011
s_11_update_count	22	10110
s_11_wait_for_count	23	10111

3.1.3 Statistiche RTL

La tabella riporta i componenti RTL usati dal tool di sintesi a partire dal codice VHDL. Sono presenti sommatore e registri da diversi bit per la gestione degli indirizzi RAM (16 bit) ed i segnali interni. Sono anche presenti multiplexer a 24 ingressi per la codifica one-hot dello stato.

Tabella 3: Statistiche RTL

Ingressi	Bit	Tipo	Quantità
<i>Adders</i>			
2	16 bit	Adder	4
2	9 bit	Adder	1
2	8 bit	Adder	3
<i>Registers</i>			
–	16 bit	Register	1
–	8 bit	Register	2
–	6 bit	Register	2
–	1 bit	Register	3
<i>Muxes</i>			
24	24 bit	Mux	1
4	24 bit	Mux	1
2	24 bit	Mux	7
2	16 bit	Mux	3
24	16 bit	Mux	1
2	8 bit	Mux	4
24	8 bit	Mux	2
2	6 bit	Mux	1
24	6 bit	Mux	2
2	1 bit	Mux	1
24	1 bit	Mux	6

3.1.4 Timing

La specifica richiede che il progetto funzioni con un periodo di clock di almeno 20 ns. Il requisito è stato verificato in sintesi applicando il vincolo `create_clock -period 20.000` sul segnale `i_clk`:

Tabella 4: Design Timing Summary

Metrica	Slack (ns)	Endpoint in violazione
Worst Negative Slack (setup, WNS)	+14.786	0 / 114
Worst Hold Slack (WHS)	+0.149	0 / 114
Worst Pulse Width Slack (WPWS)	+9.500	0 / 72

Il Worst Negative Slack indica che il modulo può operare fino a periodi di clock di 5.2 ns, rispettando il vincolo dei 20 ns con un margine di quasi 4 volte.

3.2 Report di simulazioni

3.2.1 Test bench 1

Il test bench di esempio fornito dal docente copre le operazioni base quali reset, inserimento in lista, rimozione dalla lista, decremento di priorità e svuotamento della lista.

Seppur non vengano coperti molti casi limite, questo test bench ci offre la possibilità di analizzare facilmente i tempi di esecuzione delle singole operazioni. Le 10 operazioni, infatti, vengono eseguite correttamente in 2540 ns.

In particolar modo, le latenze in cicli di clock (periodo 20 ns), con N task presenti in lista e k task spostati, sono le seguenti:

- **OP 00:** $(4 + 3N) \cdot 20$ ns
- **OP 01:** $(6 + 3k) \cdot 20$ ns, con $k = N - 1$; lista vuota: $2 \cdot 20$ ns
- **OP 10:** $(6 + 2N + 3k) \cdot 20$ ns
- **OP 11:** $3 \cdot 20$ ns

Nota: nell'operazione **10** il termine $2N$ è il costo della scansione duplicati, che percorre sempre l'intera lista, mentre $3k$ è il costo degli spostamenti, con k compreso tra 0 (inserimento in fondo) e N (inserimento in testa). Per caratterizzare il comportamento medio, si considera una distribuzione delle priorità equiprobabile: $E[T] = (6 + \frac{25}{8}N) \cdot 20$ ns.

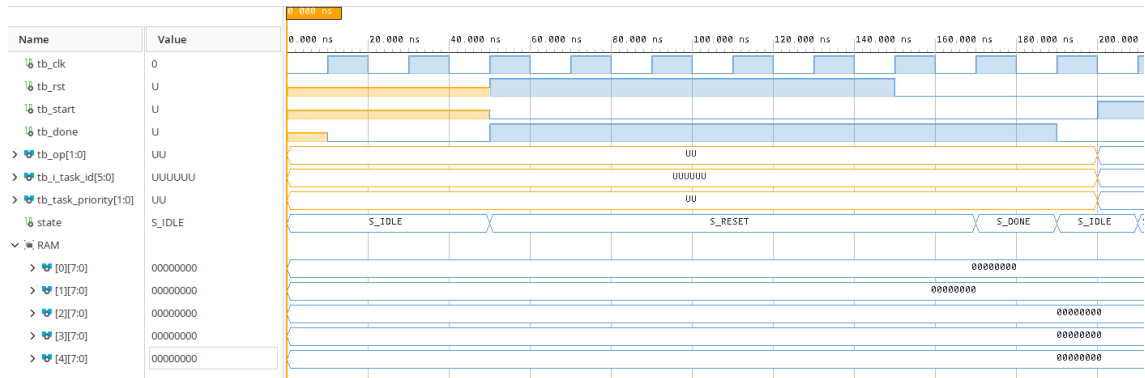


Figura 1: Operazione di reset.

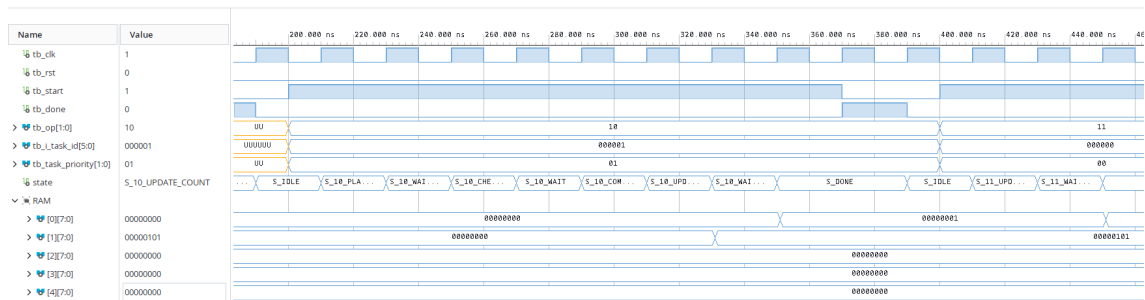
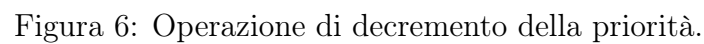
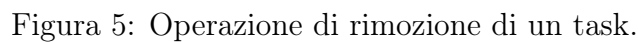
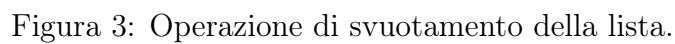


Figura 2: Operazione di inserimento in lista vuota.



3.2.2 Test bench 2 (casi limite)

Il secondo test bench è stato pensato per coprire i possibili casi limite che potrebbero presentarsi:

Caso testato	Descrizione
Reset asincrono	Reset durante operazioni in corso: il modulo torna allo stato iniziale senza corrompere la memoria.
Operazioni su lista vuota	Rimozione, decremento e svuotamento su lista vuota non producono scritture. La rimozione restituisce $ID = 000000$.
Saturazione della priorità	Task già a priorità 11 non vengono modificati da ulteriori decrementi.
Ordinamento	Inserimenti con priorità diverse e tra task di pari priorità mantengono il corretto ordinamento.
ID duplicati e $ID = 0$	L'inserimento viene ignorato se l'ID è già presente o vale 0. Il modulo conclude comunque correttamente il protocollo START-DONE.
Sequenza mista di operazioni	Inserimenti, invecchiamenti, rimozioni e svuotamento in sequenza mista.

3.2.3 Test bench 3 (latenze)

Un terzo test bench è stato usato per misurare, per ciascuna operazione, il numero esatto di cicli di clock che passano tra il ricevimento di START e l'invio di DONE. Usato per verificare le formule di latenza riportate nel paragrafo 3.2.1.

Le 15 misure (inserimenti con 0–3 spostamenti e inserimento in fondo, rimozioni con lista vuota e con 1–4 task, invecchiamenti con 0–4 task) coincidono tutte con il valore atteso, a conferma che le formule di latenza sono corrette.

Tutti e tre i test bench passano sia in simulazione comportamentale sia in simulazione post-sintesi functional.

4 Conclusioni

I test comportamentali e post-sintesi mostrano che il modulo funziona correttamente, rispettando la specifica. Il componente completa l'elaborazione del test bench di esempio in 2540 ns, con un utilizzo delle risorse hardware estremamente contenuto sul dispositivo target Artix-7 FPGA xc7a200tfbg484-1, senza alcun latch inferito. Il vincolo di clock di 20 ns è rispettato con ampio margine (percorso critico ≈ 5.2 ns).

In fase di progettazione ho scelto deliberatamente di adottare un numero di stati superiore allo stretto necessario, privilegiando la leggibilità rispetto all'efficienza. Questa scelta, pur garantendo la correttezza del comportamento, introduce una leggera inefficienza temporale: accorpendo o eliminando alcuni stati intermedi sarebbe possibile ridurre la latenza complessiva di qualche punto percentuale. Ad esempio, gli stati iniziali di alcune operazioni possono essere accorpati nello stato **S_IDLE**, all'interno del ramo di decodifica dell'operazione corrispondente.