

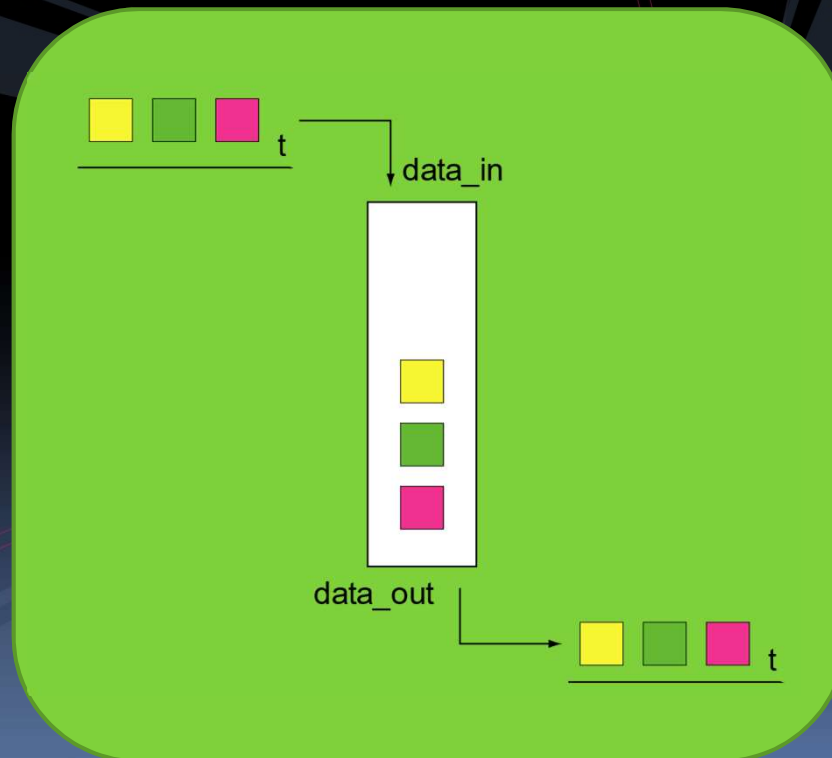


# VHDL

FIFO - EXE

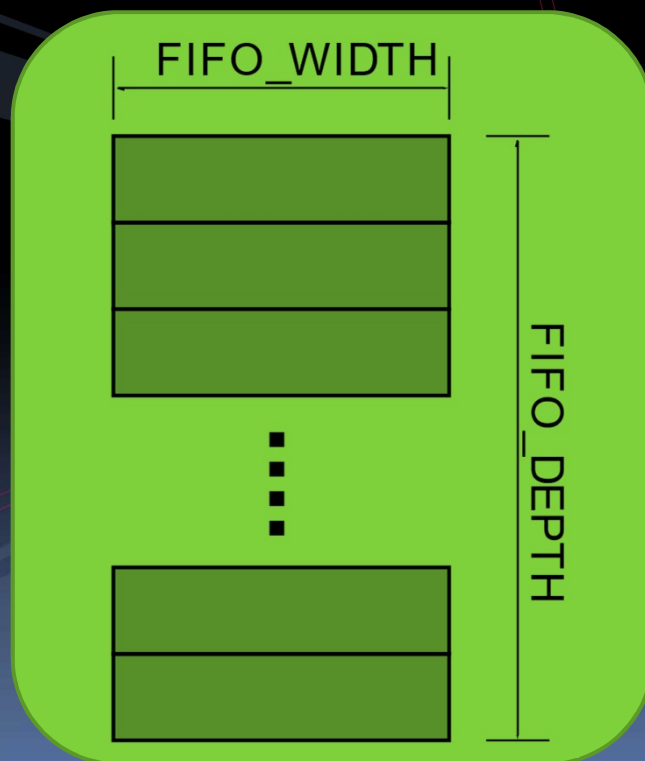
# FIFO

Una «**FIFO**», **First-In First-Out**, è una struttura di memoria che permette di memorizzare dati nella porta d'ingresso e restituirli, su richiesta, nella porta d'uscita. I dati salvati più vecchi saranno i primi ad essere espulsi.



# FIFO

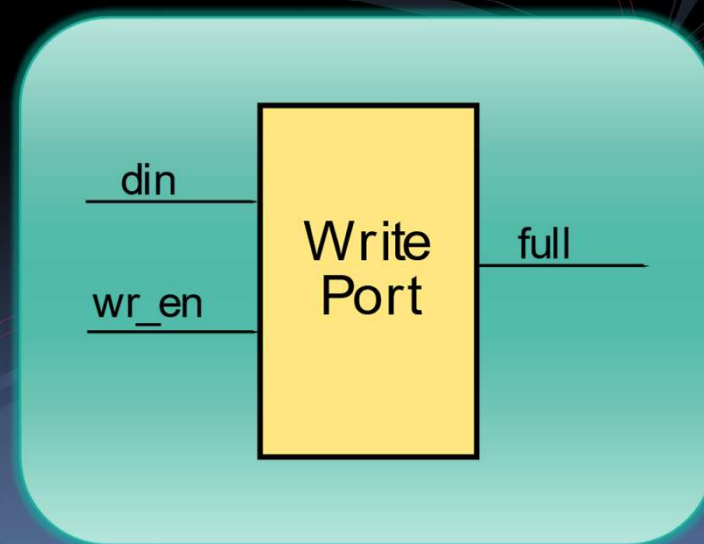
Le «**FIFO**» si differenziano in base al numero di parole memorizzabili prima di essere piene (**FIFO\_DEPTH**) ed alla larghezza della parola accettata in ingresso (**FIFO\_WIDTH**).



# WR Port

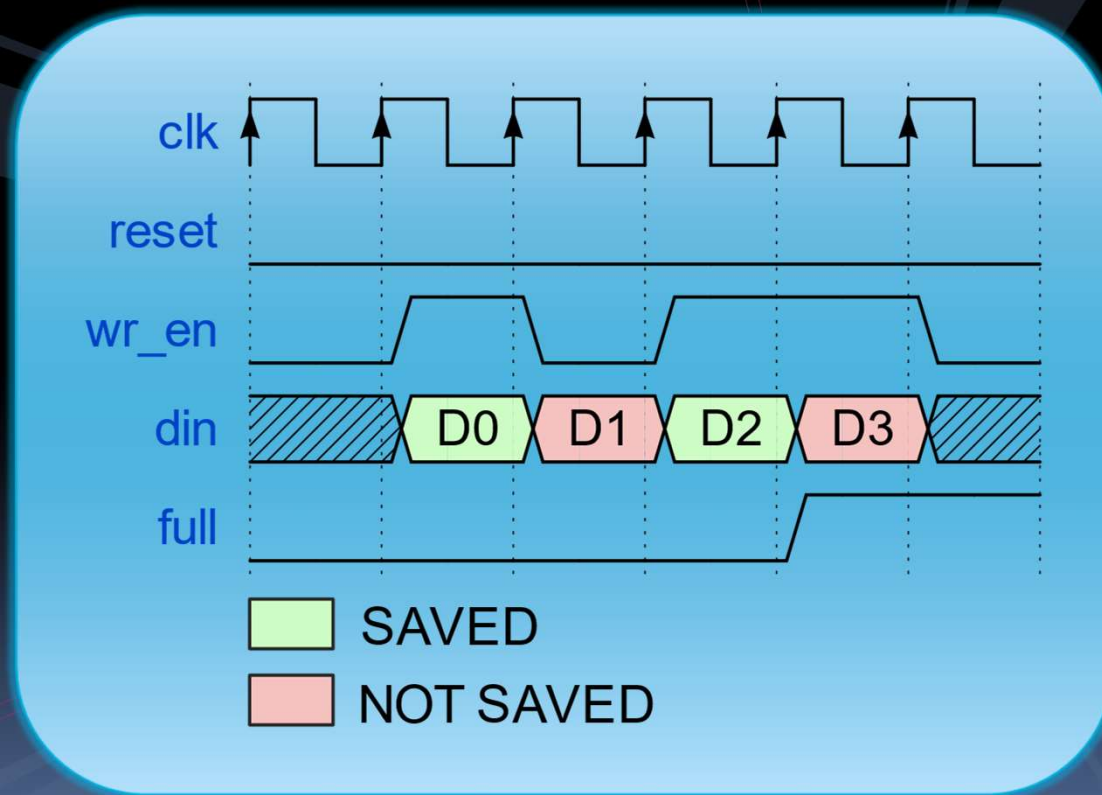
La porta di scrittura presenta questi segnali:

- **din**: Parola in ingresso da memorizzare
- **wr\_en**: Quando è attivo salva la parola al prossimo fronte positivo del clock
- **full**: Quando la memoria è piena e non può accettare più nuovi dati



# WR Waveform

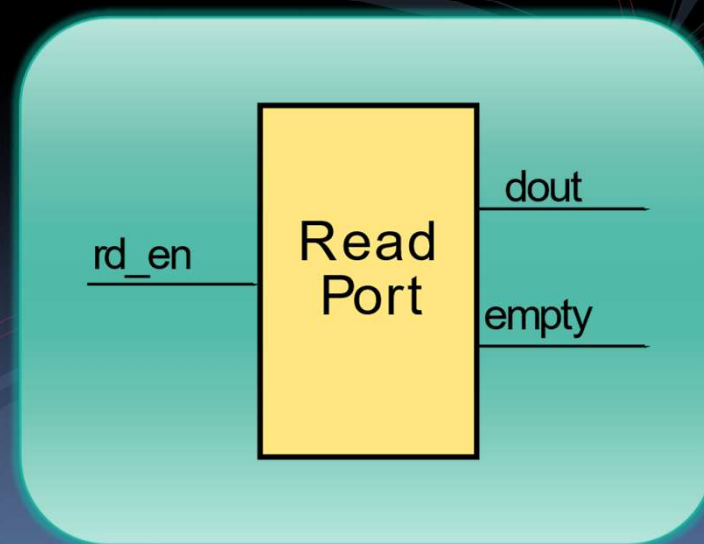
Qui un esempio di scrittura:



# RD Port

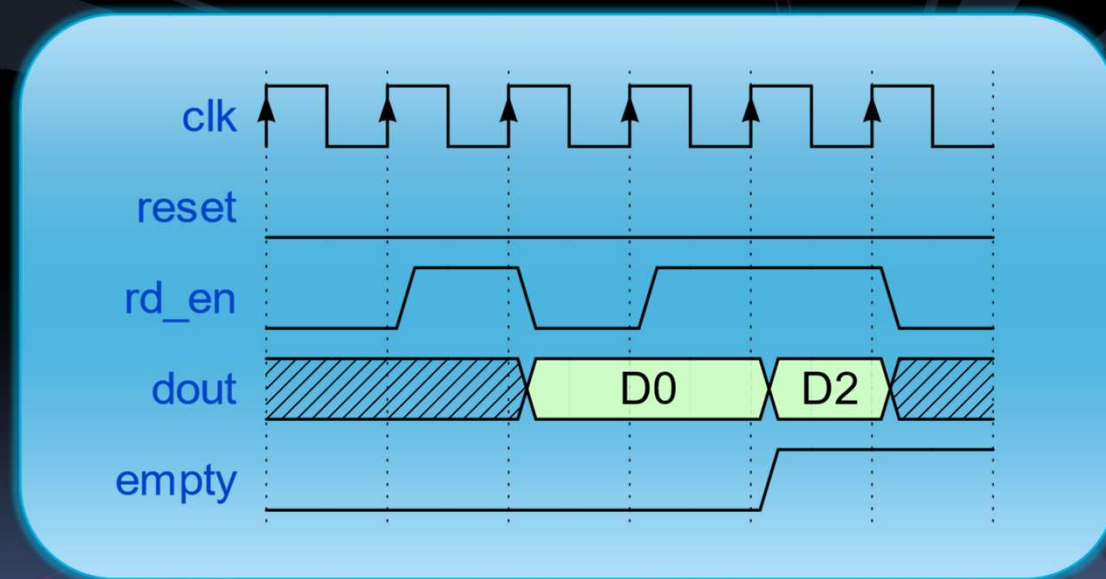
La porta di lettura presenta questi segnali:

- **dout**: Parola in uscita dalla memoria
- **rd\_en**: Quando è attivo richiede l'espulsione di una nuova parola memorizzata al successivo fronte di clock
- **empty**: Quando la memoria è vuota non è possibile richiedere nuove parole



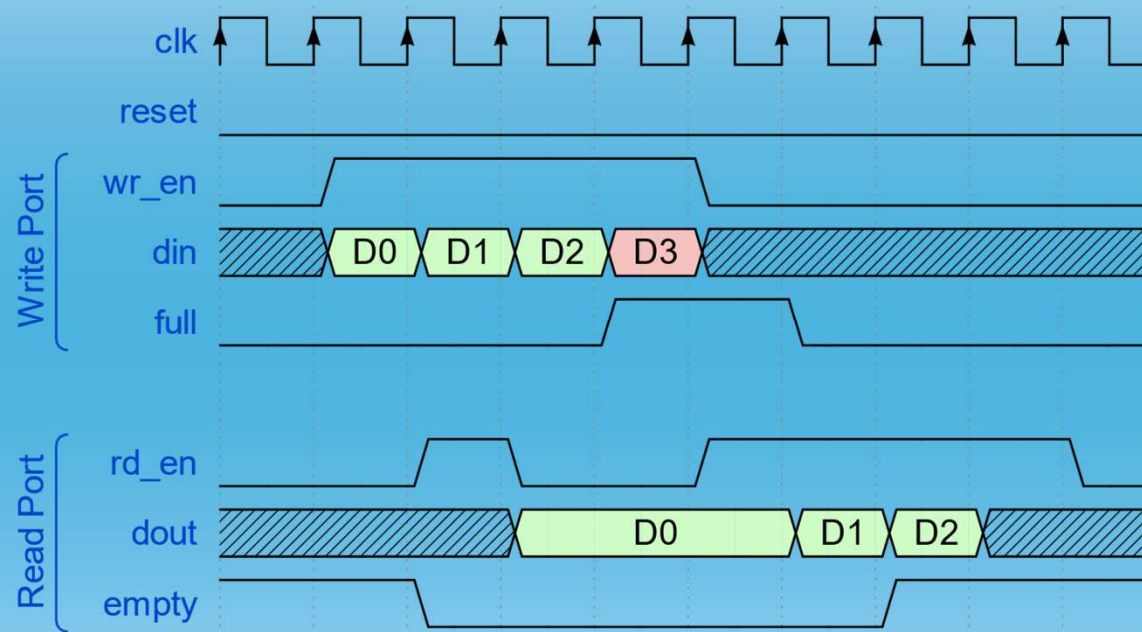
# RD Waveform

Qui un esempio di lettura:



# WR & RD Waveform

Qui un esempio di scrittura e lettura:







# Testo

Creare una entity «FIFO» con le seguenti porte:

```
entity FIFO is
  Generic(
    FIFO_WIDTH   : integer := 8;
    FIFO_DEPTH   : integer := 16
  );
  Port(
    reset        : in std_logic;
    clk          : in std_logic;

    din          : in std_logic_vector(FIFO_WIDTH-1 DOWNT0 0);
    dout         : out std_logic_vector(FIFO_WIDTH-1 DOWNT0 0);

    rd_en        : in std_logic;
    wr_en        : in std_logic;

    full         : out std_logic;
    empty        : out std_logic
  );
end FIFO;
```



# Testo

Nella costruzione della «FIFO» è importante notare i casi particolari:

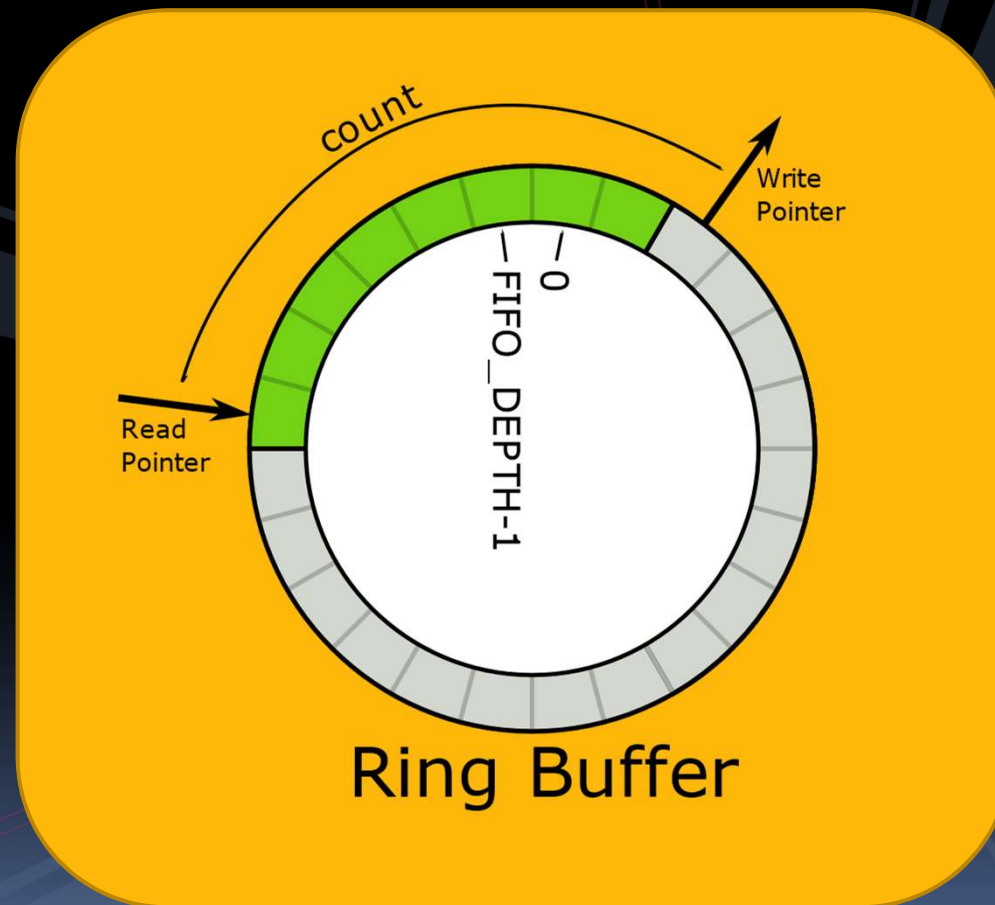
- Quando la memoria è piena anche se **wr\_en = '1'** non può scrivere un dato nuovo
- Quando la memoria è vuota anche se **rd\_en = '1'** non può espellere un dato perché non esiste
- Se **rd\_en = '1'** and **wr\_en = '1'** =>
  - Se **empty = '0'** and **full = '0'** allora il numero totale delle parole immagazzinate non cambia ma cambia il contenuto della memoria perché un valore vecchio è stato tolto e ne è stato inserito uno nuovo
  - Se **empty = '1'** allora solo la scrittura ha effetto
  - Se **full = '1'** allora solo la lettura ha effetto



# Tips

- Costruire la FIFO partendo da una memoria creata tramite un tipo personalizzato.
- Tenere traccia di dove scrivere e leggere tramite due «signal» che verranno usati come puntatori.
- Tenere traccia di quanti elementi sono stati immagazzinati tramite un signal.
- Alcune cose è utile farle sincrone altre no.
- Provare a farlo sia senza che con variabili per capire come possono modificare lo stile di scrittura

# Tips





# Note

Note:





# Consigli

**Attenzione:** di seguito alcune linee guida per arrivare ad una soluzione dell'esercizio.

Consiglio di non leggerli prima di aver pensato autonomamente ad una soluzione.

# Consigli

Cercare di creare un design pulito dove siano ben separate le sezioni di codice adibite ad una singola funzione.

Come si può vedere dall'esempio non compare all'interno del «rising\_edge(clk)» la sezione per la gestione dei segnali «empty» e «full». Probabilmente è più conveniente gestirli direttamente in data flow.

```
elsif rising_edge(clk) then

    -- Count Engine
    if ... then
        ...
    end if;

    -- Write Pointer
    if ... then
        ...
    end if;

    -- Read Pointer
    if ... then
        ...
    end if;

    -- Data in Engine
    if ... then
        ...
    end if;


    -- Data out Engine
    if ... then
        ...
    end if;

end if;
```



# Consigli

La gestione dei puntatori deve essere curata molto bene poiché anche un singolo errore lo si protrarrà fino al successivo reset. Visto che i puntatori devono andare ad indicizzare un array, conviene dichiararli direttamente come tipo integer (o suoi subtype), così da poterli utilizzare direttamente senza conversioni.



```
signal read_pointer, write_pointer: integer range 0 TO FIFO_DEPTH-1 := 0;  
signal count_word : integer range 0 TO FIFO_DEPTH := 0;
```



# Consigli

Per creare la memoria necessaria al «Ring Buffer» creare un nuovo tipo array e utilizzarlo come segnale. La memoria così creata sarà di tipo lineare ad indirizzi; gestendo i puntatori in modo che saltino dalla coda della memoria alla testa si crea un ring buffer.

```
type memory_type is array (0 TO FIFO_DEPTH-1) of std_logic_vector(din'RANGE);  
signal memory : memory_type;
```

```
if write_pointer = FIFO_DEPTH-1 then  
    write_pointer <= 0;  
else  
    write_pointer <= write_pointer + 1;  
end if;
```

```
if read_pointer = FIFO_DEPTH-1 then  
    read_pointer <= 0;  
else  
    read_pointer <= read_pointer + 1;  
end if;
```