Descrizione VHDL di componenti sequenziali

14 giugno 2003

1 Registri

I registri sono una famiglia di componenti utilizzati per la memorizzazione. Il loro funzionamento dipende quindi, oltre che dai segnali di ingresso, anche dalla loro storia passata. Questa proprietà è caratteristica dei circuiti sequenziali.

L'elemento di memoria minimale è quello che consente la memorizzazione di un bit. Due sono le famiglie principali che realizzano questo comportamento: i *flip-flop* ed i *latch*. Essi saranno utilizzati per costruire elementi di memoria più capaci e con funzionalità più complesse.

1.1 Flip-flop di tipo D

Il flip-flop di tipo D è caratterizzato da due linee di ingresso (nella fig. 1 indicati come d e clk) e da una linea di uscita (q). Il segnale di ingresso clk viene detto *clock* e governa la modifica dello stato del segnale di uscita: l'uscita q assume il valore dell'ingresso d solo in presenza di un fronte di salita del clock clk (cioè in corrispondenza di un cambiamento del segnale clk da 0 a 1). Ciò significa che se il clock non cambia di valore (e precisamente passando da 0 a 1), il segnale di ingresso d non avrà alcun effetto sull'uscita q. Per questo motivo questo flip-flop viene anche chiamato *edge triggered*.

Va notato che la forma del segnale di clock non viene in alcun modo specificata. Dal punto di vista puramente teorico, il clock potrebbe assumere qualsiasi forma d'onda. Tuttavia, tipicamente il clock è un segnale periodico che assume in un semiperiodo il valore 0 e nell'altro semiperiodo il valore 1. Esso serve a sincronizzare i componenti di uno stesso circuito ("detta i tempi"). Un comportamento come quello dell'aggiornamento dell'uscita del flip-flop viene detta *sincrono*, in quanto avviene in sincronia con il segnale di clock.

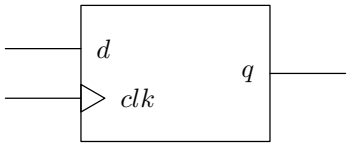

Figura 1: Rappresentazione schematica di un flip-flop.

La descrizione dell'entity VHDL corrispondente al componente in fig. 1 è:

```
entity FF_D is
  port (d, clk: in bit;
              q: out bit);
end FF_D;
```

Ci sono due modi per implementare in VHDL il comportamento di un flip-flop D:

```
architecture behav1 of FF_D is
begin
  process (clk)
  begin
    if (clk'event and clk='1') then
      q <= d;
    end if;
  end process;
end behav1;

architecture behav2 of FF_D is
  process -- no sensitivity list
  begin
    wait until clk'event and clk='1';
    q <= d;
  end process;
end behav2;
```

Entrambe le **architecture** descrivono il flip-flop in stile comportamentale e utilizzano il costrutto **process** per gestire la dipendenza da segnale di clock. L'**architecture** `behav1` fa uso della sensitivity list del processo per esplicitare la sincronizzazione del segnale di uscita, `q`, dal segnale di clock, `clk`, mentre L'**architecture** `behav2` non riposta alcun segnale in sensitivity list, ma lascia che il processo rimanga in attesa di un fronte di salita sul segnale `clk`.

Va notata l'espressione usata per descrivere il fronte di salita del clock: `clk'event and clk='1'`. `clk'event` indica l'attributo `event` del segnale `clk`. Esso assume il valore booleano *vero* solo nell'istante in cui il segnale cambia di valore. L'espressione `clk'event and clk='1'` va quindi letta

come: il valore del segnale `clk` è appena cambiato e il suo valore è 1. Tale situazione si verifica solo in presenza di un fronte di salita del segnale.

1.2 Registri a n bit

Componendo n flip-flop, si possono costruire registri a n bit: basta mettere i flip-flop in parallelo sullo stesso clock, come rappresentato in figura fig. 2.

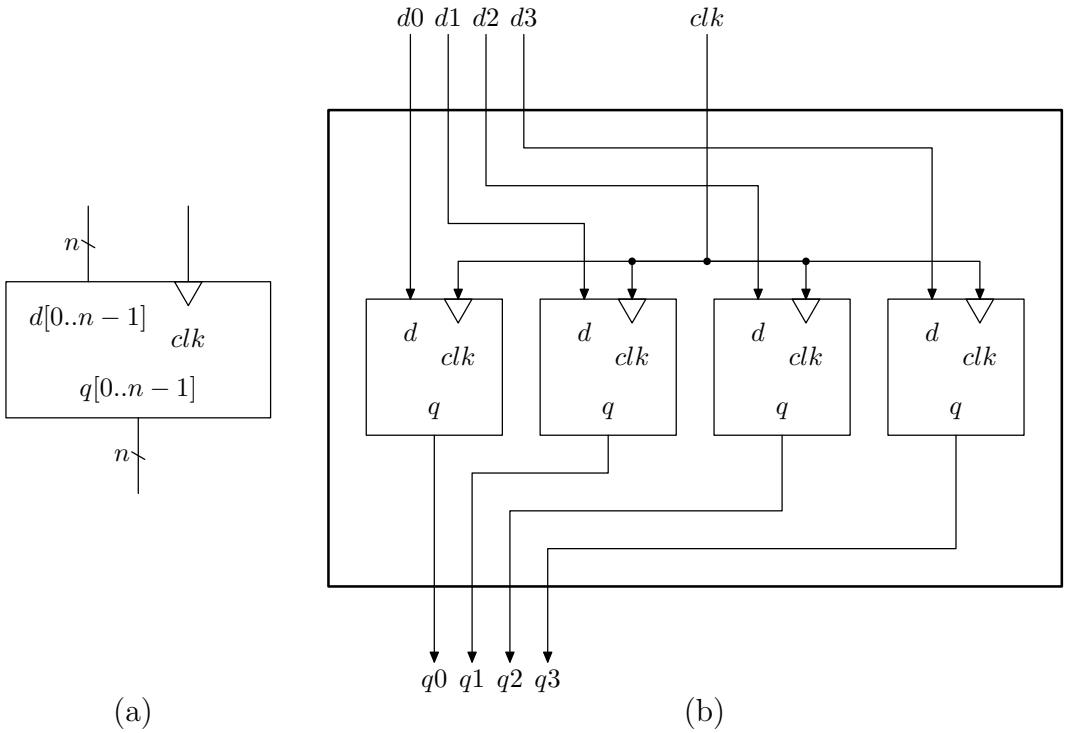

Figura 2: (a) Un registro a n bit. (b) Registro a 4 bit costruito utilizzando 4 flip-flop di tipo D.

Il circuito in fig. 2b può essere descritto in VHDL strutturle come segue:

```
entity REG4 is
  port (  d: in bit_vector(0 to 3);
          clk: in bit;
          q: out bit_vector(0 to 3));
end REG4;

architecture STRUCT of REG4 is
  component FF_D
    port (d, clk: in bit;
          q: out bit);
  end component;
```

```
begin
  u1: FF_D port map (d(0),clk,q(0));
  u2: FF_D port map (d(1),clk,q(1));
  u3: FF_D port map (d(2),clk,q(2));
  u4: FF_D port map (d(3),clk,q(3));
end STRUCT;
```

1.3 Flip-flop di tipo SR

Un altro tipo di flip-flop è il *flip-flop di tipo SR* (fig. 3), che deve il suo nome alla presenza di due segnali di ingresso, *set* e *reset*, la cui funzione è quella di impostare il segnale di uscita a 1 o a 0, rispettivamente.

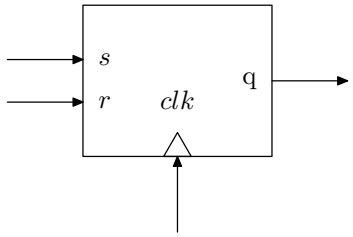

Figura 3: Rappresentazione schematica del flip-flop di tipo SR.

La descrizione dell'entity VHDL che descrive il componente in figura 3 è la seguente:

```
entity FF_set_reset is
  port ( s, r, clk: in std_logic;
                 q: out std_logic);
end FF_set_reset;
```

In un flip-flop di tipo SR le operazioni di set e reset sono sincrone. La seguente tabella delle verità descrive questo comportamento:

s	***r***	***clk***	***q***
0	0	↑	q
0	1	↑	0
1	0	↑	1
1	1	↑	**X**
-	-	0	q
-	-	1	q

Il simbolo ↑ indica il fronte di salita del segnale, mentre il simbolo "X" significa che il segnale non è definito (ciò accade quando i segnali di set e reset vengono attivati contemporaneamente).

Il comportamento descritto nella tabella precedente viene descritto dal seguente VHDL comportamentale:

```
architecture BEHAV of FF_set_reset is
begin
  process (clk)
  begin
    if (clk'event and clk='1') then
      if (s='1' and r='1') then
        q <= 'X';
      elseif (r='1') then
        q <= '0';
      elseif (s='1') then
        q <= '1';
      end if;
    end if;
  end process;
end BEHAV;
```

L'utilizzo del tipo `std_logic` anziché `bit` consente di specificare il comportamento indefinito in occasione della contemporaneità dell'attivazione del segnale di set e di quello di reset (nonché del fronte di salita).

1.4 Latch

La differenza tra un FF e un latch è che l'uscita del latch riproduce il valore del segnale d'ingresso per tutto il tempo in cui il segnale di clock assume il valore alto. Il segnale di clock assume pertanto il significato di un segnale abilitatore o inibitore e sarà per questo indicato con il nome di *enable* nel seguito.

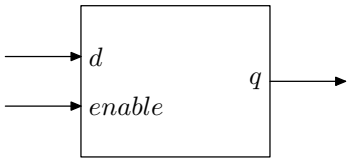

Figura 4: Latch di tipo D.

Il latch riportato in fig. 4 può essere descritto dalla seguente entity VHDL:

```
entity latch_D is
  port ( d, enable: in std_logic;
                 q: out std_logic);
end latch_D;
```

Il latch viene detto *trasparente* quando il segnale *enable* è alto e viene detto *opaco* quando il segnale *enable* è basso. Questo comportamento è descritto dal seguente codice VHDL:

```
architecture behaviour of latch_D is
begin
  process (enable)
  begin
    if (enable = '1') then
      q <= d;
    end if;
  end process;
end behaviour;
```

1.5 SR latch

Analogamente alla famiglia dei flip-flop, esistono diverse varianti per i latch. Fra queste, il latch di tipo SR, il quale viene controllato tramite un segnale di set e uno di reset (fig. 5). Data l'assenza di una segnale di clock, si tratta quindi di una rete asincrona.

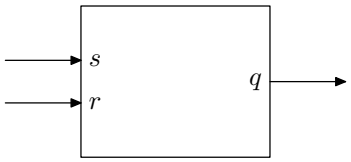

Figura 5: Latch di tipo SR

Il funzionamento è il seguente:

- quando s vale 1, q viene posto ad 1 (a tale valore rimane anche quando s torna a 0);
- quando r vale 1, q viene posto ad 0 (a tale valore rimane anche quando $reset$ torna a 0);
- quando sia s sia r valgono 0, il valore di q non cambia;
- quando sia s sia r valgono 1, il valore di q è indefinito.

Ciò viene descritto dalla seguente tabella di verità:

s	***r***	***q***
0	0	q
0	1	0
1	0	1
1	1	**X**

Il latch di tipo SR può essere descritto mediante il seguente codice VHDL:

```
entity latch_SR is
  port ( s, r: in std_logic;
           q: out std_logic);
end latch_SR;

architecture behaviour of latch_SR is
begin
  process (s, r)
  begin
    if (s = '1' and r = '0') then
      q <= '1';
    elseif (s = '0' and r = '1') then
      q <= '0';
    elseif (s = '1' and r = '1') then
      q <= 'X';
    end if;
  end process;
end behaviour;
```

1.6 Registro con reset asincrono

È possibile realizzare varianti dei componenti fin qui presentati limitando la sincronizzazione solo ad alcune operazioni. Per esempio, un flip-flop con reset asincrono è descritto dal seguente codice VHDL:

```
-- in questo caso il reset e' prioritario!
architecture behaviour of FF_asincr_reset is
begin
  process (clk, r)
  begin
    if (r='1') then
      q <= '0';
    elseif (clk'event and clk='1') then
      if (s='1') then
        q <= '1';
      end if;
    end if;
  end process;
end behaviour;
```

dove l'entity `FF_asincr_reset` è identica all'entity `FF_set_reset`

Da notare che il segnale `r` è stato aggiunto nella sensitivity list e che la sequenza dei costrutti **`if`** rispecchia la priorità data agli eventi.

2 Contatore

I registri sono elementi essenziali per costruire i contatori. Questi ultimi formano una famiglia di componenti sequenziali molto variegata. Nella versione più semplice (fig. 6), il contatore ha la caratteristica di incrementare periodicamente l'uscita di 1. L'uscita è costituita da una sequenza di bit e può pertanto essere utilizzata per denotare un numero in notazione binaria. Gli incrementi avvengono solo ogni fronte di clock.

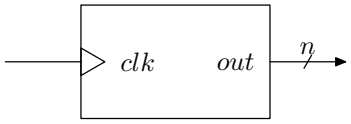

Figura 6: Contatore a n bit.

Ad esempio, nel caso di un contatore a 4 bit di uscita, ipotizzando che il segnale di *out* valga inizialmente "0000", avremmo questo andamento:

# fronti	***out***
0	"0000"
1	"0001"
2	"0010"
3	"0011"

e così via. Un contatore come quello visto nell'esempio, cioè con 4 bit di uscita, viene chiamato contatore modulo 16. Tale componente "conta" fino al numero binario 1111 (15 in base 10) e poi riparte da 0000 (0 in base 10). Un contatore modulo n ha perciò n stati.

Normalmente il contatore ha alcune ulteriori funzionalità, date dalla presenza di segnali addizionali. In fig. 7 ne sono riportati alcuni:

reset riporta l'uscita al valore minimo indicabile, e cioè 0 (che in base binaria si indica con una stringa di n zeri);

set riporta l'uscita al valore massimo indicabile, e cioè $2^n - 1$ (che in base binaria si indica con una stringa di n uno);

up/down indica al contatore in quale direzione contare: se ascendente oppure discendente;

enable indica al counter se contare oppure no nel corrente ciclo di clock;

data in/load porta in uscita la sequenza di bit immessa dal segnale *data in* una volta attivato il segnale di *load*. v

Le operazioni di set, reset e caricamento possono essere (in modo indipendente l'una dall'altra) in sincronia con il clock oppure in modo asicrono. Questo fatto dà luogo a molteplici varianti del componente contatore.

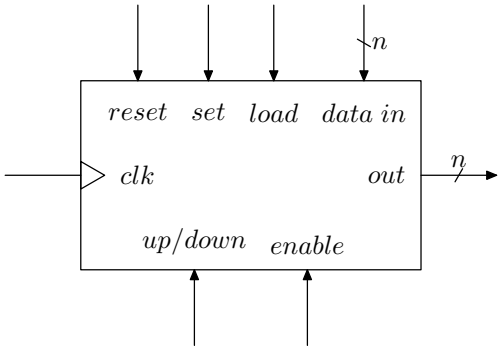

Figura 7: Contatore a n bit con segnali addizionali

Il seguente codice VHDL descrive un contatore a 8 bit con reset asincrono.

```
entity cont8 is
  port(clk, reset: in std_logic;
       outa: out std_logic_vector(0 to 7));
end cont8;

architecture rtl of cont8 is
  signal t: std_logic_vector(0 to 7);
begin

  process(clk, reset)
  begin
    if (reset = '1' ) then
      t <= "00000000";
    elseif (clk'event and clk = '1' ) then
        t <= t + "00000001";
    end if;
  end process;

  outa <= t;

end rtl;
```

Il segnale `t` viene utilizzato per gestire l'incremento. Esso appare sia a destra che a sinistra dell'operatore di assegnamento. Al suo posto non può perciò essere usato un segnale in modalità **`in`** perché non potrebbe essere a sinistra, né un segnale in modalità **`out`** perché non potrebbe essere a destra dell'operatore di assegnamento. Da notare l'uso di `std_logic` per poter effettuare le operazioni di somma su sequenze di bit.

La versione sincrona dell'operazione di reset può essere descritta in VHDL portando il test sul segnale `reset` all'interno del codice eseguito dopo il test sul fronte di salita del segnale `clk`. Inoltre, il segnale `reset` può essere tolto dalla sensitivity list.

Di seguito è riportato il codice VHDL di un contatore che include le modifiche sopra descritte, con l'aggiunta del segnale `updown` per governare la direzione del conteggio.

```
entity counter is
  port(clk, reset, updown: in std_logic;
             outa: out std_logic_vector(0 to 7));
end counter;

architecture sincr_reset of counter is
  signal t: std_logic_vector(0 to 7);
begin
  process(clock)
  begin
    if (clk'event and clk = '1') then
      if (reset = '1') then
        t <= "00000000";
      elseif (updown = '1') then
        t <= t + "00000001";
      else
        t <= t - "00000001";
      end if;
    end if;
  end process;

  q <= t;
end sincr_reset;
```

2.1 Contatore completo

Il contatore può essere descritto come composizione di componenti fin qui visti. La fig. 8 illustra questo concetto. Alla fine della catena di componenti, si trova un registro. La sua funzione è mantenere in memoria il valore in del segnale in uscita, *out* e di aggiornarlo ad ogni fronte di salita del segnale di clock, *clk*. Inoltre, vengono fornite le funzionalità di *set* e *reset*. Il segnale di ingresso del registro viene fornito da un multiplexer che, regolato dal segnale *load*, seleziona il valore dell'uscita del registro al battito di clock precedente opportunamente incrementato o il valore del segnale *data in*. L'incremento del valore di *out* viene effettuato da un addizionatore, che, oltre al segnale di *out*, riceve in ingresso un segnale con il valore da impostato dai segnali *enable* e *up/down* tramite due multiplexer in cascata. Il segnale *enable* posto

a zero, ha l'effetto di porre il segnale di incremento a zero, in modo da inibire la modifica del valore dell'uscita. Il segnale *up/down* alto pone il valore di incremento pari a "+1", mentre lo pone a "-1" se basso. Da notare che i valori "0", "+1" e "-1" devono essere rappresentati come una opportuna sequenza di n bit.

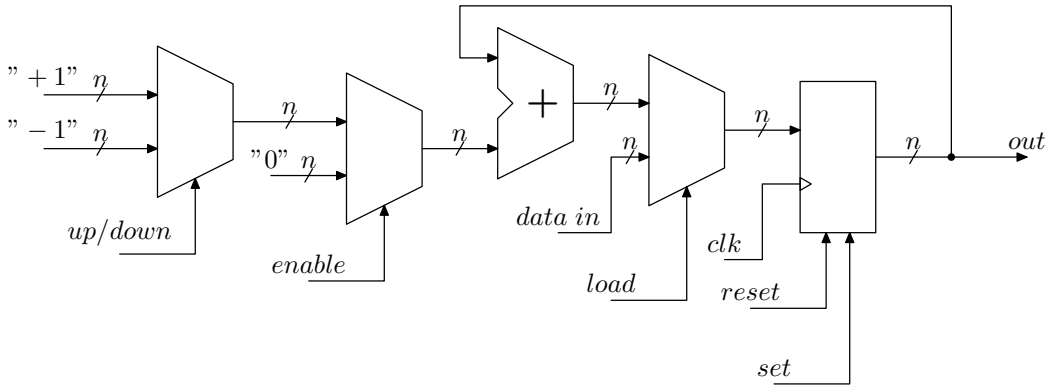

Figura 8: Visione sistemistica di un contatore.

Più in dettaglio, le funzionalità di *reset* e *set* possono essere realizzate ponendo il segnale di ingresso del registro ripettivamente in *and* con il segnale di *reset* invertito e in *or* con il segnale di *set*.

Questa descrizione del contatore può essere riportata quasi pedissequamente in VHDL, usando lo stile strutturale:

```
entity cont_full_sist is
  port(clk, set, reset, load, enable, updown: in std_logic;
       datain: in std_logic_vector(0 to 7);
       outa: out std_logic_vector(0 to 7));
end cont_full_sist;

architecture struct of cont_full_sist is

  component REG8                  -- registro da 8 bit
    port (
      clk, set, reset : in std_logic;
      d               : in std_logic_vector (0 to 7);
      q               : out std_logic_vector (0 to 7));
  end component;

  component MUX2_8                -- multiplexer a 2 ingressi da 8 bit
    port (
      a, b : in  std_logic_vector (0 to 7);
      sel  : in std_logic;
      q    : out std_logic_vector (0 to 7));
```

```
      -- q vale a se sel vale '1', b altrimenti
  end component;

  component ADDER_8               -- sommatore da 8 bit
    port (
      a, b : in std_logic_vector (0 to 7);
      r    : out std_logic_vector (0 to 7));
  end component;

  signal t1,t2,t3,t4,t5 : std_logic_vector (0 to 7);
  -- i seguenti segnali svolgono la funzione di costanti
  signal zero, piu_uno, meno_uno: std_logic_vector (0 to 7);

begin
-- outa non puo' essere usato in modalita' "in" come richiesto dalla
-- port map dell'adder: viene usato t3 al suo posto

  registro: REG8 port map (clk,set,reset,t1,t3);
  load_mux: MUX2_8 port map (datain,t2,load,t1);
  adder: ADDER_8 port map (t3,t4,t2);
  enable_mux: MUX2_8 port map (t5,zero,enable,t4);
  updown_mux: MUX2_8 port map (piu_uno,meno_uno,updown,t5);
  outa <= t3;
  -- NB: esiste un modo piu' corretto di indicare le costanti in VHDL,
  -- ma non e' stato visto a lezione
  zero <= "00000000";
  piu_uno <= "00000001";
  meno_uno <= "11111111";    -- "-1" in complemento a 2
end struct;
```

Da notare che i segnali interni `t1`–`t5` vengono usati per connettere i componenti, mentre i segnali `zero`, `piu_uno`, `meno_uno` sono usati impropriamente per definire dei segnali costanti (esistono costrutti più appropriati per descrivere valori costanti, ma non rientrano nel sottoinsieme di VHDL visto a lezione).

3 Registro a scorrimento

Il registro a scorrimento (*shift register*) è costituito da una catena di flip-flop in cascata. Nella sua versione più semplice (fig. 9), il registro a scorrimento ha il seguente funzionamento: ad ogni colpo di clock, ogni flip-flop passa il proprio valore al successivo ed il primo flip-flop della catena assume il valore del segnale di ingresso d. Il valore precedentemente memorizzato nell'ultimo flip-flop viene assegnato al segnale di uscita del registro a scorrimento, q.

Il componente in fig. 9 può essere descritto in VHDL dalla seguente entità:

Figura 9: Registro a scorrimento

```
entity ShiftRegister is
  port (
    d, clk : in std_logic;
    q : out std_logic);
end ShiftRegister;
```

Da notare che la lunghezza della catena di flip-flop che compongono il registro a scorrimento non compare in alcun modo nella dichiarazione di entity (e nemmeno nello schema in fig. 9). Il numero di flip-flop impiegati costituisce la capacità del registro a scorrimento e, secondo la descrizione sopra data, rappresenta il numero di colpi di clock che bisogna attendere perchè un segnale di ingresso venga presentato in uscita.

Il funzionamento di un registro a scorrimento a 4 bit può essere descritto come segue:

```
architecture behav of ShiftRegister is
  signal t : std_logic_vector(0 to 3);
begin
  process (clk)
  begin
    if clk'event and clk = '1' then -- fronte di salita del clock
      t <= d & t(0 to 2);
      q <= t(3);
    end if;
  end process;
end behav;
```

Il modello semplice del registro a scorrimento fin qui illustrato può essere arricchito di funzionalità. Ciò richiede, analogamente al caso del contatore, l'aggiunta di alcuni segnali addizionali:

reset imposta i valori dei flip-flop vengono impostati a '0';

set imposta i valori dei flip-flop vengono impostati a '1';

right/left indica in che direzione effettuate lo scorrimento;

enable abilita oppure disabilita lo scorrimento nel corrente ciclo di clock;

data in/load imposta nel registro la sequenza di bit immessa dal segnale *data in* una volta attivato il segnale di *load*;

data out porta all'esterno i valori di uscita di tutti i flip-flop, consentendo una lettura parallela del contenuto dello shift register.

Con l'aggiunta dei segnali *data in/load* e *data out*, il registro a scorrimento pouò essere utilizzato per la conversione seriale/parallela dell'elaborzione.

Come per il contatore, le operazioni sullo shift register possono essere sincrone o asincrone, a seconda della modalità con cui possono essere effettuate: se vengono permesse solo in coincidenza di un fronte di salita, vengono dette *sincrone*; se possono essere effettuate in ogni istante, vengono dette *asincrone*.

Il seguente codice VHDL descrive uno shift register a 4 bit con reset asincrono e load sincrono:

```
entity shift_reg is
  port (
    d, clk, load, reset : in  bit;
    datain              : in  bit_vector (0 to 3);
    q                   : out bit);
end shift_reg;

architecture reset_asincr of shift_reg is
  signal t : std_logic_vector(0 to 3);
begin
  process (clk,reset)
  begin
    if reset = '1' then
      t <= "0000";
    elseif clk'event and clk = '1' then -- fronte di salita del clock
      if load = '1' then
        t <= datain;
      else
        t <= d & t(0 to 2);
        q <= t(3);
      end if;
    end if;
  end process;
end reset_asincr;
```

L'operazione di scorrimento dei valori dei flip-flop può essere descritta anche tramite un costrutto di ciclo **for-loop**:

```
architecture reset_asincr of shift_reg is
  signal t : std_logic_vector(0 to 3);
begin
  process (clk,reset)
  begin
    if reset = '1' then
      t <= "0000";
```

```
    elseif clk'event and clk = '1' then -- fronte di salita del clock
      if load = '1' then
        t <= datain;
      else
        for i in 0 to 2 loop
          t(i+1) <= t(i);
        end loop;
        t(0) <= d;
        q <= t(3);
      end if;
    end if;
  end process;
end reset_asincr;
```

Il funzionamento del registro a scorrimento può essere descritto in termini di composizione di componenti standard elementari. Per esempio, in fig. 10 è illustrato uno schema a blocchi che descrive un registro a scorrimento a 4 bit con segnali di set e reset utilizzando quattro flip-flop di tipo D (dotati anch'essi di segnali *set* e *reset*). Esso può essere descritto in VHDL strutturale come segue:

```
entity sh_reg_sr is
  port (
    d, clk, set, reset : in  bit;
    q                  : out bit);
end sh_reg_sr;

architecture struct of sh_reg_sr is
  component ffd_sr
    port (
      d, clk, set, reset : in  bit;
      q                  : out bit);
  end component;
  signal t1, t2, t3      :     bit;

begin  -- struct
  reg0 : ffd_sr port map (d, clk, set, reset, t1);
  reg1 : ffd_sr port map (t1, clk, set, reset, t2);
  reg2 : ffd_sr port map (t2, clk, set, reset, t3);
  reg3 : ffd_sr port map (t3, clk, set, reset, q);
end struct;
```

Da notare che la modalità con cui operano i segnali `set` e `reset` dipendono dall'implementazione del componente `ffd_sr`: ad esempio, se `ffd_sr` ha il reset asincrono, anche `sh_reg_sr` lo avrà.

La fig. 11 illustra, mediante uno schema a blocchi, una realizzazione di un registro a scorrimento a 4 bit con caricamento di dati in parallelo. Questa azione è controllata dal segnale `load`, il quale, agendo su una serie di

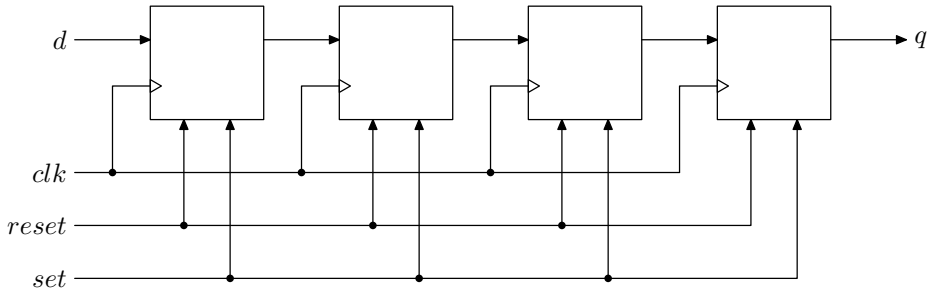

Figura 10: Schema a blocchi di un registro a scorrimento a 4 bit con segnali di set e reset.

multplexer, seleziona l'ingresso per i flip-flop: se `load` vale '0', il registro a scorrimento ha il funzionamento caratteristico, mentre se `load` vale '1', agli ingressi dei flip-flop vengono presentati i valori portati dal bus `datain`.

Questo circuito può essere descritto in VHDL strutturale come segue:

```
entity sh_reg_load is
  port (
    d, clk, load : in  bit;
    datain : in  bit_vector (0 to 3);
    q                  : out bit);
end sh_reg_load;

architecture struct of sh_reg_load is
  component ffd
    port (
      d, clk : in bit;
      q                  : out bit);
  end component;

  component mux
    port (
      a, b, sel: in bit;
      z: out bit);
  end component;

  signal t1, t2, t3, t4, t5, t6, t7: bit;

begin  -- struct
  reg0 : ffd port map (t4,clk,t1);
  reg1 : ffd port map (t5,clk,t2);
  reg2 : ffd port map (t6,clk,t3);
  reg3 : ffd port map (t7,clk,q);
  mux0 : mux port map (d,datain(0),load,t4);
  mux1 : mux port map (t1,datain(1),load,t5);
  mux2 : mux port map (t2,datain(2),load,t6);
```

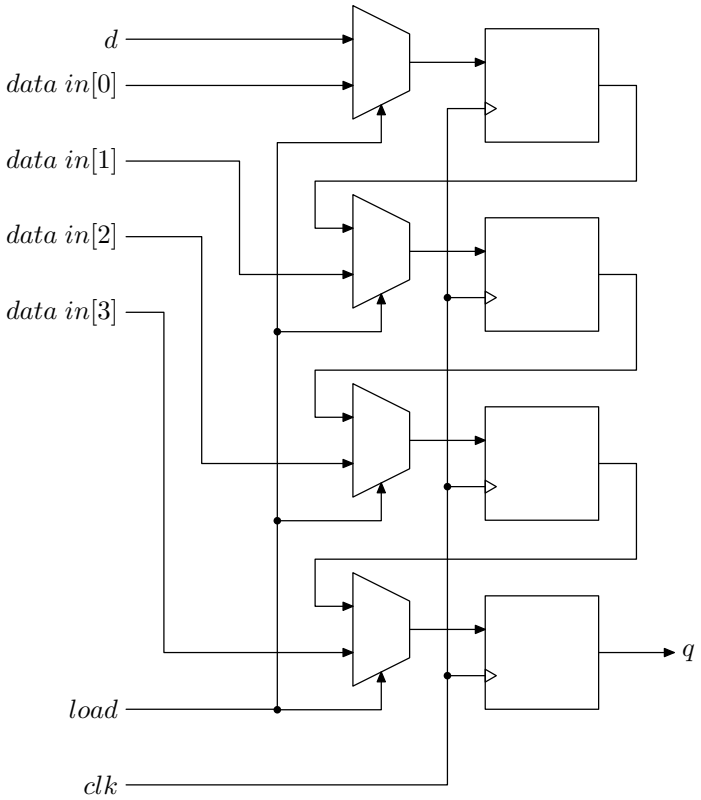

Figura 11: Schema a blocchi di un registro a scorrimento a 4 bit con caricamento in parallelo.

```
  mux3 : mux port map (t3,datain(3),load,t7);
end struct;
```