Descrizione di registri, ROM, RAM in Verilog

▼ Creatore originale: @Gianbattista Busonera

In questa sezione verrà approfondito l'utilizzo del blocco always (modellazione comportamentale) per descrivere registri e memorie.

Registri

Nella maggioranza dei casi, considereremo Flip-Flop attivi sul fronte di salita/discesa del clock, chiamati edge-triggered Flip Flop. È anche possibile descrivere registri di tipo Latch, che sono attivi su un segnale di enable.

Esempio - Registro a 4 bit

Si vuole definire un registro da 4 bit per modellare circuiti di memoria tali che consentano una sola operazione alla volta, sia essa di lettura, di scrittura o di reset sul fronte di salita del clock.

Si vuole fare in modo che il segnale di `reset` (attivo con il valore basso) imposti il contenuto del registro a zero e che il segnale `read_write` si comporti come segue:

- se `read_write == 0`, allora voglio effettuare una scrittura;
- se `read_write == 1`, allora voglio effettuare una lettura.

▼ Componenti

Per tale scopo, sono necessari:

- una variabile di tipo `reg` da 4 bit, che sarà l'uscita del blocco `always`;
- un selettore di lettura/scrittura, chiamato `read_write`;
- un segnale di clock che sincronizza la scrittura e il reset;
- un segnale di reset (attivo al valore basso) che consente di impostare il valore nel registro a 0;
- un input `write_data` per la scrittura dei dati nel registro;
- un output `read_data` su cui sarà disponibile il dato memorizzato nel registro quando viene effettuata un'operazione di lettura, altrimenti si avrà alta impedenza, portando a un comportamento simile ad un buffer tri-state.

▼ Versione del codice sulle slide

Questo codice è presente nelle slide del corso, ma risulta errato.

Il problema è che l'operatore `assign` descrive bene una assegnazione combinatoria, ma non una sequenziale: infatti `read_data` assume il valore in storage ogni qual volta `rw = 1`, indipendentemente dal clock, mentre viene richiesta una lettura sul fronte di salita del clock.

```
module reg4b (output [3:0] read_data,
              input [3:0] write_data,
              input clk, rst, rw);

  reg [3:0] storage; // rappresenta il contenuto (memoria) del registro
```

```
    initial storage = 4'b0001; // SOLO PER TESTBENCH

    assign read_data = (rw ? storage: 4'bZ); // se il segnale read_write = 1 assegno
    // read_data il valore contenuto in memoria, diversamente alta impedenza.


    always @(posedge clk) begin // sul fronte di salita del clock...
        if(rst == 0) // se reset == 0, devo resettare il contenuto.
            storage = 0;
        else if(rw == 0)
            storage = write_data;
    end
endmodule
```

▼ Versione del codice corretta

```
module reg4b (read_data, write_data, clk, rst, rw);
    output reg [3:0] read_data; // uscita su 4 bit
    input [3:0] write_data; // ingresso su 4 bit
    input clk, rst, rw; // ingressi vari su 1 bit

    reg [3:0] storage;

    initial storage = 4'b0001; // SOLO PER TESTBENCH
    // SCRITTURA E RESET
    always @(posedge clk) begin
        if (rst == 0) begin // DIAMO PRIORITA' al reset
            storage = 0;
            read_data = 0;  // Reset anche dell'uscita
         end
        else if (rw == 0) begin // Scrittura → rw = 0
            storage = write_data;  // Scrittura nei "FF"
            read_data = 4'bZ; // Alta impedenza durante la scrittura
           end
        else
            read_data = storage;  // Lettura sincronizzata al fronte di salita
```

```
    end
endmodule
```

▼ Testbench e verifica (entrambe le versioni)

```
module reg4b_tb;

    reg clk;            // Clock
    reg rst;            // Reset
    reg rw;             // Read/Write control
    reg [3:0] write_data; // Input data to write to the register
    wire [3:0] read_data; // Output data from the register

    // Instanza del registro a 4 bit
    reg4b uut (
        .read_data(read_data),
        .write_data(write_data),
        .clk(clk),
        .rst(rst),
        .rw(rw)
    );

    // Generazione del clock (periodo 10 ns)
    always begin
        #5 clk = ~clk;  // Periodo di clock di 10 ns
    end

    // Stimoli di input
    initial begin
        // Inizializza i segnali
        clk = 0;
        rst = 1;  // Inizia con reset disabilitato
        rw = 0;   // Iniziamo con modalità scrittura
        write_data = 4'b0000; // Scriviamo un valore iniziale

        // Aspetta un po' prima di testare
        #10;
```

```
// Reset attivo per il primo test (rst=0)
rst = 0;  // Applicare il reset
#10;      // Attendere un ciclo di clock per completare il reset

// Fine del reset
rst = 1;  // Disabilitare il reset
write_data = 4'b1010; // Scrivere un nuovo dato (1010)

// Scrittura nel registro con rw = 0
rw = 0;  // Modalità scrittura
#10;     // Aspettare un ciclo di clock per la scrittura

// Lettura dal registro con rw = 1
rw = 1;  // Modalità lettura
#10;     // Aspettare un altro ciclo di clock

// Scrittura di un nuovo valore
write_data = 4'b1111;
rw = 0;  // Modalità scrittura
#10;     // Aspettare un ciclo di clock

// Lettura del nuovo valore
rw = 1;  // Modalità lettura
#10;     // Aspettare un altro ciclo di clock

// Reset del registro
rst = 0;  // Applicare il reset
#10;      // Attendere un ciclo di clock per il reset

// Verifica di lettura dopo il reset (dovrebbe essere 0000)
rst = 1;  // Disabilitare il reset
rw = 1;   // Modalità lettura
#10;      // Aspettare un ciclo di clock

// Finire la simulazione
$finish;
```

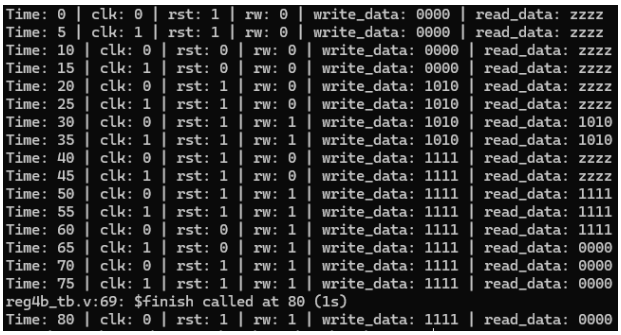
```
    end

    // Monitor personalizzato: stampa dopo ogni fronte di salita del clock con un p
    always @(posedge clk) begin
        #0.01; // Ritardo piccolo dopo il fronte di salita
        $display("Time: %0t | clk: %b | rst: %b | rw: %b | write_data: %b | read_data
                 $time, clk, rst, rw, write_data, read_data);
    end
endmodule
```

1. Nella versione del codice presente sulle slide si ottiene il seguente output:

   Si nota facilmente come, nel caso della riga in cui `time=30`, ci aspettavamo che `read_data` fosse pari a `zzzz`, in quanto il clock è allo stato basso e precedentemente `read_data` era impostato a `zzzz`.

   Ciò significa che **ho letto anche se il clock era allo stato basso**, che è un comportamento non voluto.

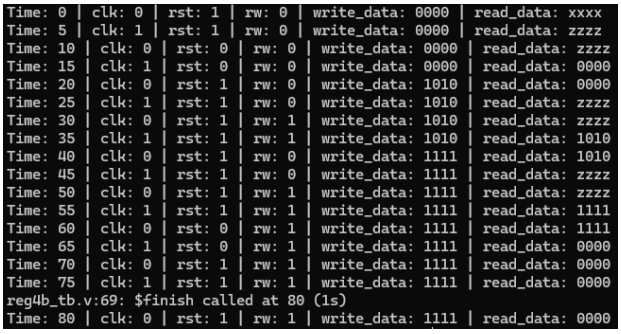


Output della prima versione del codice
N.B. il reset è attivo basso!

2. Nella versione corretta del codice si ottiene il seguente output:

   Si noti come, al tempo `time=40`, troviamo `read_data=1010`, che risulta diverso da `write_data=1111`, come ci si aspetta, essendo sul fronte di discesa del clock, e per cui il valore di `read_data` non dovrebbe essere modificato.

   Infatti, il valore verrà letto al tempo `time=55`, dove abbiamo che `clk=1` e che `rw=1`, portando all'effettiva lettura del valore in scrittura.



Output della seconda versione del codice
N.B. il reset è attivo basso!

▼ Esempio - Registro parametrico

Si vuole realizzare un registro parametrico a numero di bit variabile (N).

Si vuole fare in modo che si possa leggere, scrivere e resettare solo sul fronte di salita del clock.

A differenza del caso precedente, si propone solo l'implementazione del creatore della pagina, in quanto sulle slide è presente lo stesso errore già visto.

```
module #(parameter N = 4) regNb (
    output reg [N-1:0] read_data,  // Dichiarato come 'reg' perché assegnato in un
    input [N-1:0] write_data,
    input clk, rst, rw
);
    reg [N-1:0] storage; // Utile per la memoria!

    always @(posedge clk) begin
        if (!rst) begin
            storage = 0;
            read_data = 0;  // Reset anche dell'uscita
        end
        else if (!rw) begin
            storage = write_data;  // Scrittura
            read_data = {N{1'bz}}; // Alta impedenza durante la scrittura
            // CREA UN VETTORE D N bit con tutti Z
        end
        else
            read_data = storage;  // Lettura sincronizzata al fronte di salita
    end
endmodule
```

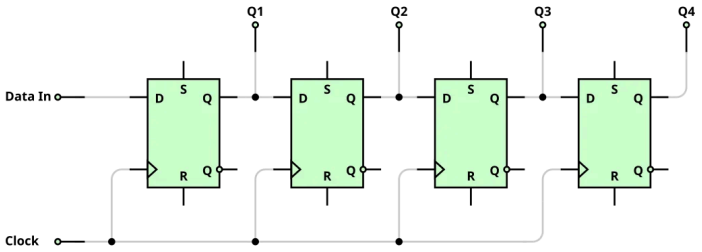
▼ Esempio - Registro a 4 bit con FF-D

Potremmo descrivere un registro a 4 bit mediante modellazione strutturale, assumendo di avere un componente Flip-Flop D (`ffD`) del tipo:

```
module ffD(output reg Q, input D, clk);
    always @(posedge clk)
```

```
    Q=D;
endmodule
```

Si possono, quindi, utilizzare 4 componenti `ffD` per implementare un registro a 4 bit.



Implementazione di un registro a 4 bit con FF-D

```
module reg4b(inout[3:0] out, input [3:0] in, clk, rw);
   ffD f3(out[3], rw ? out[3] : in[3], clk);
   ffD f2(out[2], rw ? out[2] : in[2], clk);
   ffD f1(out[1], rw ? out[1] : in[1], clk);
   ffD f0(out[0], rw ? out[0] : in[0], clk);
endmodule
```

In scrittura (`rw == 0`) si propaga l'input (`in`) agli ingressi `D` dei FF.

In lettura (`rw == 1`) scriviamo nuovamente in uscita il valore precedentemente memorizzato.

Visto che nella modellazione strutturale non è possibile utilizzare costrutti condizionali, è necessario utilizzare un costrutto condizionale ternario per selezionare l'ingresso `D` dei FF.

⚠️ La versione illustrata differisce leggermente da quella implementata sulle slide, ma la reputiamo più immediata per comprendere che l'uscita resta invariata, mentre l'ingresso è fornito dalla vecchia uscita nel caso di lettura (mantenendo in memoria) o dal nuovo input nel caso in cui si debba scriver

Si riporta anche il codice delle slide per completezza, pur sapendo che contiene un errore.

```
module reg4b(output [3:0] out, input [3:0] in, clk, rw);
   wire [3:0] Q;

   ffD f3(Q[3], rw ? Q[3] : in[3], clk);
   ffD f2(Q[2], rw ? Q[2] : in[2], clk);
   ffD f1(Q[1], rw ? Q[1] : in[1], clk);
   ffD f0(Q[0], rw ? Q[0] : in[0], clk);

   assign out <= Q; // non si può fare questa assegnazione non bloccante
   // assign out = Q; è corretto
endmodule
```

In questo caso, il docente ha preferito utilizzare una variabile di appoggio Q probabilmente per aggiungere della logica combinatoria tra l'uscita Q e l'uscita out.

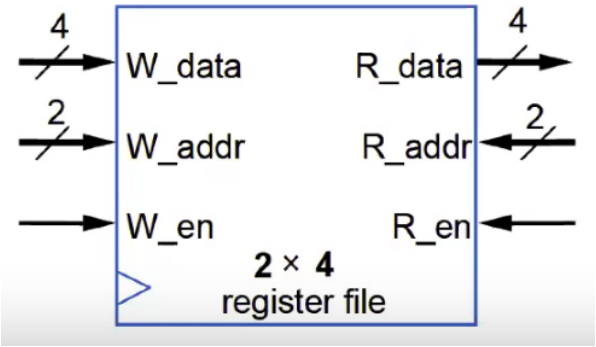
Risulta essere leggermente più verboso, ma segue lo stesso principio del codice implementato in precedenza.

Register File

Il Register File è un blocco di tanti registri (matrice di registri) che possono avere almeno una porta di lettura (solitamente almeno 2) e almeno una di scrittura.

Un esempio di Register File è descritto in figura, ed è formato da:

- 4 registri da 4 bit ciascuno;
  - Di conseguenza, saranno necessari $\log_2(4) = 2$ bit di indirizzo (R_{address} e W_{address});
  - W_{data} e R_{data} sono su 4 bit rispettivamente
- un segnale di abilitazione per la scrittura;
- un segnale di abilitazione per la lettura.



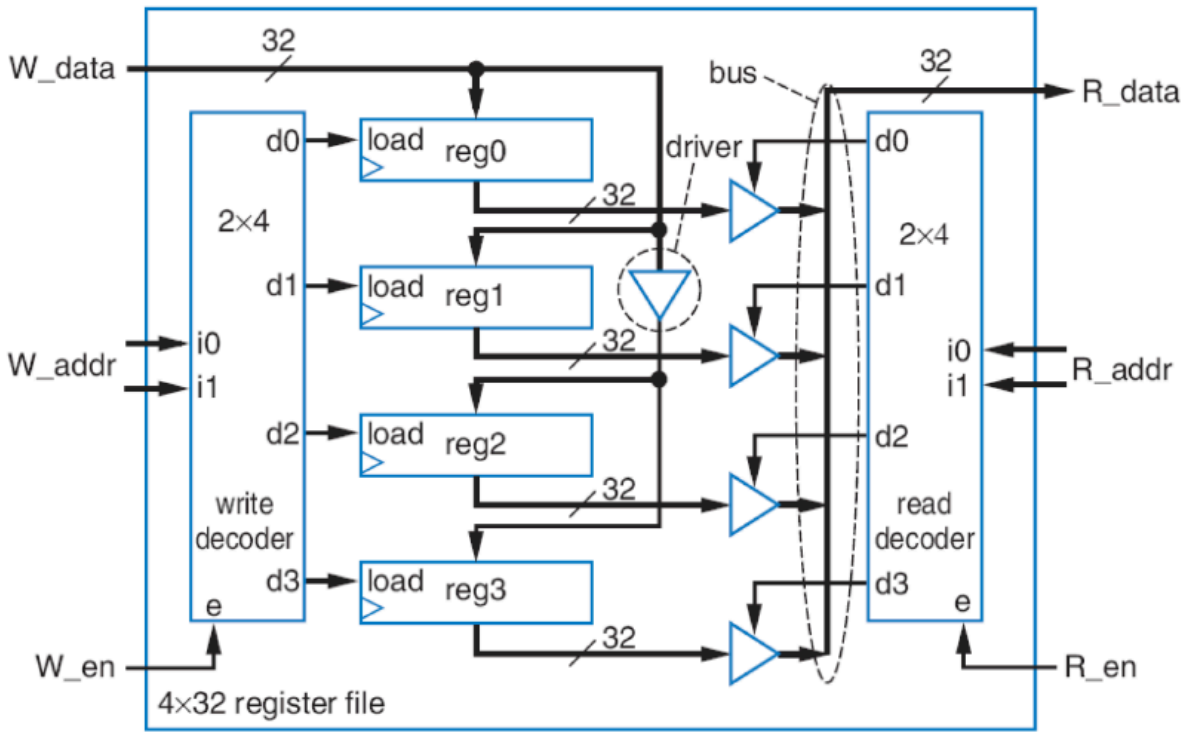
Esempio di Register File

▼ Spiegazione generica Register File 4×32

Facciamo una breve spiegazione del funzionamento del Register File 4×32:

- caratteristiche generali:
  - parallelismo dei dati su 32 bit;
  - il numero di registri è 4.
- scrittura:
  - è necessario un decodificatore di scrittura che, dato l'indirizzo su 2 bit (in quanto ci sono 4 registri in totale), seleziona quale dei 4 registri attivare per la scrittura;
- lettura:
  - è necessario un altro decodificatore che, dato l'indirizzo su 2 bit, seleziona quale dei 4 buffer tri-state collegati ai registri abilitare.
  - nel caso in cui il buffer tri-state sia abilitato, quest'ultimo lascia passare la parola memorizzata nel rispettivo registro, altrimenti imposta l'alta impedenza sul bus.

⚠️ Ricordiamo cosa succede nel caso di combinazioni logiche con buffer tri-state.



Visualizzazione del circuito interno generico per un Register File 4×32

▼ Implementazione strutturale del Register File 4×32

Si immagini un file che descrive il modulo reg32en, il quale dovrà comportarsi come un registro con:

- uscita di lettura;
- ingresso di scrittura;
- condizione di abilitazione a lettura;
- condizione di abilitazione alla scrittura;
- clock e reset.

I due decoder sono presenti poiché, in base al valore dell'indirizzo `r_addr`, definiscono quale degli elementi va effettivamente letto/scritto.

In tal senso, si riporta anche l'implementazione strutturale, con:

- 4 registri da 32 bit con output `enable`;
- 2 decoder 2:4 con segnale di `enable` per abilitare letture e scritture nei singoli registri interni, per cui:
  - `enable` attivo abilita l'uscita decodificata dai decoder;
  - `enable` inattivo imposta tutte le uscite a 0.

```
module reg4×32 (r_data , w_data, r_addr, w_addr, r_en, w_en, clk, rst);
    output [31:0] r_data;
    input [31:0] w_data;
    input [1:0] r_addr, w_addr;
    input r_en, w_en, clk, rst;

    wire w_r3, w_r2, w_r1, w_r0, r_r3, r_r2, r_r1, r_r0;
    dec2×4en r_dec (r_addr[1], r_addr[0], r_en, r_r3, r_r2, r_r1, r_r0);
    dec2×4en w_dec (w_addr[1], w_addr[0], w_en, w_r3, w_r2, w_r1, w_r0);

    reg32en r0 (r_data, w_data, r_r0, w_r0, clk, rst);
    reg32en r1 (r_data, w_data, r_r1, w_r1, clk, rst);
    reg32en r2 (r_data, w_data, r_r2, w_r2, clk, rst);
    reg32en r3 (r_data, w_data, r_r3, w_r3, clk, rst);
endmodule
```

▼ Implementazione Register File 8×8 con 2 porte di lettura e 1 di scrittura

Si voglia implementare un Register file con 8 registri da 8 bit ciascuno. Si faccia in modo che ci siano 2 porte di lettura e 1 porta di scrittura, e che, se reset è attivo basso in modalità sincrona, si pulisca completamente il Register File.

Si vuole fare in modo che si possa leggere indipendentemente dal segnale di clock e che si possa selezionare il segnale `r1r2w` per selezionare l'operazione da effettuare, e che si possa scrivere e resettare solo sul fronte di salita del clock, come segue:

- se `r1r2w=1`, si legge dalla porta di lettura 1;
- se `r1r2w=2`, si legge dalla porta di lettura 2;
- se `r1r2w=0`, si scrive sulla linea selezionata.

```
module regFile (data_out1, data_out2, data_in, clk, rst, r1r2w, address);
   // DICHIARAZIONI
   output [7:0] data_out1, data_out2;
   input [7:0] data_in;
   // Poichè si può leggere o scrivere solo da una porta per volta
   // non ha senso utilizzare 3 indirizzi differenti come ha fatto lui.
   // io userò solo un indirizzo "address", anzichè 3 distinti.
   input [2:0] address; // servono 3 bit per selezionare le 8 parole.
   input clk, rst;
   input [1:0] r1r2w; // Segnale di selezione

   reg [7:0] storage [7:0]; // matrice 8×8 → 8 registri da 8 bit ciascuno

   assign data_out1 = (r1r2w == 1) ? storage[address] : 'bZ;
   assign data_out2 = (r1r2w == 2) ? storage[address] : 'bZ;
   // Le due assegnazioni continue sopra, nel caso in cui non si sia stato
   // selezionato correttamente la porta di lettura, mettono l'uscita in alta
   // impedenza su tutti i bit disponibili

   always @(posedge clk) begin
      if(rst == 0) // Se ho deciso di resettare
         storage = 0; // azzera tutti i registri in automatico su fronte di salita
      else if(r1r2w == 0)
         storage[address] = data_in; // WRITE solo su fronte di salita
   end
endmodule
```

Random Access Memory (RAM)

Una memoria RAM è una memoria sia leggibile che scrivibile in un tempo unitario attraverso degli indirizzi. Da qui il nome Random Access Memory, a discapito delle memorie ad accesso sequenziale.

Praticamente è identica a un Register File, ma con alcune piccole differenze:

- una RAM ha solitamente una sola porta condivisa per lettura e scrittura;
- una RAM è in genere più grande delle 512 o 1024 parole dei Register File;

- una RAM memorizza i bit in maniera più efficiente dei comuni Flip Flop e, oltretutto, è solitamente implementata su chip di forma quadrata per ridurre le lunghezze delle connessioni più lunghe.

▼ Implementazione RAM parametrica

Si vuole implementare una RAM parametrica con una porta condivisa per lettura e scrittura.

Da specifica vogliamo che:

- le operazioni di lettura e scrittura devono essere sincrone al fronte di salita del clock;
- la RAM è abilitata solo quando il “chip select” `cs` è attivo alto;
- se “write enable” `we = 1` e `cs = 1`, si può scrivere;
- se “output enable” `oe = 1` e `cs = 1`, possiamo leggere, assicurandoci che `we = 0` , poiché l’operazione di lettura può essere eseguita solo quando non è in esecuzione l’operazione di scrittura, in modo da evitare inconsistenze di qualsiasi tipo.

```
module RAM(data, address, clk, cs, we, oe);
    // DICHIARAZIONE PARAMETRI
    parameter DATA_WIDTH = 8; // parallelismo dei dati paria a 8 di default
    parameter ADDR_WIDTH = 8; // indirizzi su 8 bit di default
    // Poichè abbiamo indirizzi su 8 bit ⇒ possiamo scrivere 2^8 parole

    parameter WORDS_COUNT = 1 << ADDR_WIDTH; // shift a sinistra di "1"
    // tante volte quanto è il valore di "ADDR_WIDTH" così che WORDS_COUNT = 
    // DICHIARAZIONE INGRESSI

    inout [DATA_WIDTH-1:0] data; // bus condiviso per leggere e scrivere con para
    // selezionato in precedenza
    input [ADDR_WIDTH-1:0] address;
    input clk, cs, we, oe;

    // DEFINISCO LA VERA E PROPRIA MEMORIA DI DATA_WIDTH * WORDS_COU
    reg [DATA_WIDTH-1:0] mem [WORDS_COUNT-1:0];
    // Definisco una variabile ausiliaria per leggere altrimenti dovrei
```

```
    // istanziare data come reg (a causa dell'always)!
    reg [DATA_WIDTH-1:0] data_out;
    // Definisco un'altra variabile ausiliaria
    reg oe_r; // output enable registrato... Ci sarà un flip flop
    // che contiene tale valore sulla base della condizione di lettura.

    /// SCRITTURA
    always @(posedge clk) begin
        if(cs && we) // se chip select = 1 (ram attiva) e write enable = 1 (scrittura)
            mem[address] = data; // memorizzo nell'apposito indirizzo la parola
    end
    /// LETTURA
    always @(posedge clk) begin
        if(cs && oe && !we) begin // se sono nella condizione di poter leggere
            data_out = mem[address]; // assegno al registro temporaneo data_out il v
            // letto
            oe_r = 1; // segno nella variabile ausiliaria che ho registrato il valore letto
        end
        else
            oe_r = 0; // non ho letto effettivamente il valore
    end

    assign data = (cs && oe_r && !we) ? data_out : 'bZ;
    // Assegnazione continua che mi consente di memorizzare data_out nel caso i
    // abbia effettivamente letto qualcosa, altrimenti alta impedenza.

    // Nota come non potevamo usare "oe" direttamente al posto di "oe_r",
    // sarebbe stato infatti possibile leggere da data_out anche nel caso in cui
    // non avessimo ancora letto dal blocco always!
    // e, oltretutto, si fa in modo che SOLO sul fronte di salita del clock
    // si legga effettivamente grazie al fatto che oe_r può valere 1 solo
    // durante un fronte di salita del clock.
endmodule
```

Read Only Memory (ROM)

Una memoria ROM è una memoria che può essere letta, ma non modificata, per cui è necessario un solo bus di lettura non condiviso `e`, e per cui di conseguenza non servono segnali di selezione read/write come la coppia `oe` e `we`.

Rispetto alle RAM, ci sono alcuni vantaggi, analizzati meglio nella sezione di teoria relativa:

- maggiore compattezza, dovuta al fatto che devono solo essere lette;
- essendo una memoria non volatile, può essere più veloce di alcuni tipi di RAM;
- bassa potenza, e non serve alimentazione per conservare i bit.

Chiaramente, si utilizza una ROM rispetto a una RAM se i dati da memorizzare cambiano raramente o, meglio, non cambiano proprio.

Inizializzazione delle memorie

Verilog consente di leggere da file per inizializzare vettori o matrici.

In particolare, si possono leggere file in formato:

- esadecimale tramite `syscall $readmemh`;
- binario tramite `syscall $readmemb`.

```
$readmemh("hex_file.txt", mem_array, [start_address], [end_address]);
// gli ultimi due parametri sono opzionali e ci consentono di inizializzare
// una sottoparte del vettore o della matrice
$readmemb("bin_file.txt", mem_array, [start_address], [end_address]);
```

Nel caso in cui il contenuto del file non sia coerente con la dimensione della memoria, la lettura viene troncata fino all'ultima locazione di memoria disponibile.

⚠️ I valori nei file di input possono essere separati:

- Singoli spazi ‘ ‘;
- TAB: '	‘;
- Carattere a capo: `\n`;

Tutti questi separatori possono anche essere mischiati tra loro.

Nel file di input si possono aggiungere dei commenti con ‘`//`’.

▼ Esempi - Inizializzazione

1. Primo esempio:

```
reg [15:0] es1 [0:3];
$readmemh("es1_file.mem", es1);
// es1 è un registro che può
// contenere 4 parole da 16 bit
```

```
dead // Commento
be ef
0 a 0 a
1234
```

2. Secondo esempio:

```
reg [2:0] es1 [0:5];
$readmemb("es2_file.mem", es2);
// es2 è un registro che può
// contenere 6 parole da 3 bit
```

```
011 101 111
111
001 101
```

3. Terzo esempio:

   Si noti come viene specificato l’indirizzo di partenza!
   La stringa
   `dead` verrà inserita in `es[4]` e le successive all’indirizzo 5,6,7, e così via.

```
reg [15:0] es3 [0:255];
$readmemh("es3_file.mem", es3, 4
// es3 è un registro che può
// contenere 256 parole da 16 bit
```

```
dead beef aaaa
casu 0110 aabb
9876 5432 1001
```

⚠️ Si noti come si vuole fare in modo che la riga 0 sia la più significativa! Facendo così, possiamo trovare in memoria il file caricato in ordine.

▼ Implementazione ROM parametrica

In questo frammento di codice, verrà utilizzata per la prima volta la parola chiave `initial`, la quale permette di eseguire delle istruzioni (che possano essere letture da file, blocchi always...) solo all'inizio della simulazione del modulo.

```
module ROM (data, // data output
                  clk, // clock input
                  address, // address input
                  re, // Read enable, analogo a output enable
                  cs); // chip select per abilitazione ROM

   // DICHIARAZIONE PARAMETRI
   parameter DATA_WIDTH = 8; // parallelismo dei dati paria a 8 di default
   parameter ADDR_WIDTH = 8; // indirizzi su 8 bit di default
   // Poichè abbiamo indirizzi su 8 bit ⇒ possiamo leggere 2^8 parole
   parameter WORDS_COUNT = 1 << ADDR_WIDTH; // shift a sinistra di "1"
   // tante volte quanto è il valore di "ADDR_WIDTH" così che WORDS_COUNT = 1

   // DICHIARAZIONE INGRESSI
   output [DATA_WIDTH-1:0] data; // bus per leggere con parallelismo pari a data
   input [ADDR_WIDTH-1:0] address;
   input clk, cs, re;

   // DEFINISCO LA VERA E PROPRIA MEMORIA DI DATA_WIDTH * WORDS_COUN
   reg [DATA_WIDTH-1:0] mem [0:WORDS_COUNT-1];
   reg re_r; // di nuovo variabile ausiliaria per riprendere il codice RAM

   // INIZIALIZZAZIONE DELLA ROM
   initial
      $readmemb ("input.data", mem); // lettura da file
   // Solo ad inizio simulazione viene memorizzato nel registro mem il contenuto
```

```
    // del file 'input.data'

    // LETTURA
    always @(posedge clk) : MEM_READ // etichetta 'MEM_READ' utile per debug
    begin
        if(cs && re)
            re_r = 1;
        else
            re_r = 0;
    end

    assign data = (cs && re_r) ? mem[address] : 'bZ;
    // Analogo a quanto visto e spiegato in precedenza.
endmodule
```