

5

Esercitazione 5 + iverilog e gtkwave

[Installazione iverilog e gtkwave](#)

[Esercizio 1 - sommatore completo](#)

[Soluzione 1 - Porte logiche](#)

[Soluzione 2 - Assegnazione continua](#)

[Soluzione 3 - Modellazione comportamentale](#)

[Implementazione testbench e verifica](#)

[Esercizio 2 - multiplexer](#)

[Soluzione 1 - Assegnazione continua](#)

[Soluzione 2 - Modellazione comportamentale](#)

[Testbench e verifica](#)

[Esercizio 3 - latch trasparente](#)

[Soluzione](#)

[Testbench](#)

[Esercizio 4 - FSM](#)

[Soluzione](#)

[Testbench](#)

[GTKWave e visualizzazione forme d'onda](#)

[Esercizio 5 - Shift-Register completo](#)

[Soluzione](#)

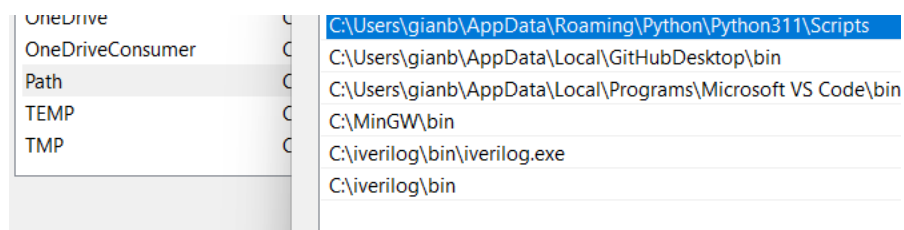
[Esercizi extra](#)

▼ Installazione iverilog e gtkwave

▼ Creatore originale: @Gianbattista Busonera

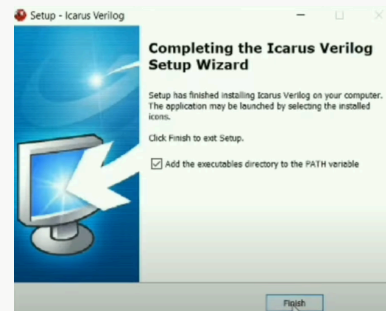
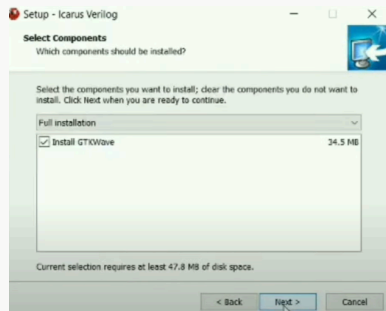
LINKS - [DOWNLOAD](#) | [TUTORIAL](#)

Per utilizzarlo su Windows occorre aggiungere alla variabile di sistema `$PATH` il path relativo a iverilog e gtkwave.



Esempio di variabile \$PATH modificata

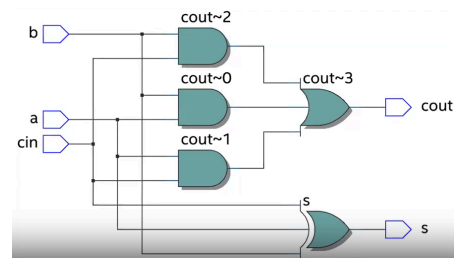
- ⚠ Durante l'installazione di iverilog vi chiederà se volete installare anche gtkwave e se volete aggiungerle alle variabili di sistema... Cliccate sì, facilita di tanto l'installazione.
Fatto ciò, riavviate il computer!



Esercizio 1 - sommatore completo

▼ Creatore originale: @Gianbattista Busonera

Si supponga di voler implementare un componente che realizza un sommatore, come in [figura](#).



Sommatore da realizzare

▼ Soluzione 1 - Porte logiche

```
module fadd (cout, s, a, b, cin);
    output cout, s;
    input a, b, cin;
    wire and_b_cin;
    wire and_b_a;
    wire and_a_cin;

    and(and_b_cin, b, cin);
    and(and_b_a, b, a);
    and(and_a_cin, a, cin);
    or(cout, and_b_cin, and_b_a, and_a_cin);
    xor(s, a, b, cin);
endmodule
```

▼ Soluzione 2 - Assegnazione continua

```
module fadd (cout, s, a, b, cin);
    output cout, s;
```

```

input a, b, cin;

assign s = a^b^cin;
assign cout = (a&b)|(a&cin)|(b&cin);
endmodule

```

▼ Soluzione 3 - Modellazione comportamentale

Si ricordi che stiamo modellando un circuito combinatorio e, pertanto, l'uscita cambia ogni qual volta cambiano gli ingressi.

E' pertanto necessario inserire nella lista di sensibilità tutti gli ingressi del nostro circuito combinatorio (a, b, cin).

```

module fadd (s, cout, a, b, cin);
    output reg s, cout;
    input a,b,cin;

    always @(*) begin
        s = a^b^cin;
        cout = (a & b) | (a & cin) | (b & cin);
    end
endmodule

```

▼ Implementazione testbench e verifica

```

`timescale 1ns/1ps

module fadd_tb;
    // segnali per collegarsi al modulo fadd
    reg a, b, cin;
    wire s, cout;
    // istanza del modulo da testare
    fadd uut (
        .s(s),
        .cout(cout),
        .a(a),
        .b(b),
        .cin(cin)
    );
    // procedura di test
    initial begin
        $display(" a | b | cin | s | cout ");
        $display("-----");
        // test per tutte le combinazioni
        for (integer i = 0; i < 8; i = i + 1) begin
            {a, b, cin} = i[2:0]; // assegna valori da 000 a 111
            #1; // attesa di 1ns per propagazione
            $display(" %b | %b | %b | %b | %b", a, b, cin, s, cout);
        end
        $finish; // termine modellazione
    end

endmodule

```

Come sfruttare un testbench

Bisogna salvare (per semplicità) nella stessa cartella i file fadd.v e fadd_tb.v (dove l'estensione .v indica che il codice è scritto in verilog). Una volta fatto ciò e, verificata la corretta installazione di iverilog occorre aprire il terminale (cmd) nella cartella di riferimento e digitare:

```
iverilog -o fadd_tb fadd.v fadd_tb.v
vvp fadd_tb
```

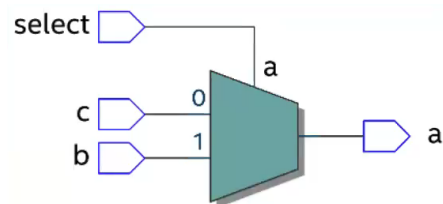
Il primo comando compilerà il testbench dandogli nome "fadd_tb", il secondo comando lo eseguirà e, a schermo, vedremo:

```
PS C:\Users\gianb\Desktop\UNI\ELAP\Lab\E05\ES01> iverilog -o fadd_tb fadd.v fadd_tb.v
PS C:\Users\gianb\Desktop\UNI\ELAP\Lab\E05\ES01> vvp fadd_tb
a | b | cin | s | cout
-----
0 | 0 | 0 | 0 | 0
0 | 0 | 1 | 1 | 0
0 | 1 | 0 | 1 | 0
0 | 1 | 1 | 0 | 1
1 | 0 | 0 | 1 | 0
1 | 0 | 1 | 0 | 1
1 | 1 | 0 | 0 | 1
1 | 1 | 1 | 1 | 1
fadd_tb.v:30: $finish called at 8000 (1ps)
```

Esercizio 2 - multiplexer

▼ Creatore originale: @Gianbattista Busonera

Si implementi un multiplexer 2→1, come in figura.



Multiplexer 2→1 da realizzare

▼ Soluzione 1 - Assegnazione continua

Soluzione funzionante, ma non ottima, in quanto utilizza il costrutto condizionale ternario presente anche in C. Non si può fare diversamente in quanto gli if, case, etc. sono utilizzabili solo nei costrutti `always` e `initial`.

```
module mux2x1 (a, b, c, select);
    output a;
    input b, c, select;

    assign a = (select ? b : c); // se select = 1 a = b, se select = 0, a = c
    // operatore ternario
endmodule
```

▼ Soluzione 2 - Modellazione comportamentale

Soluzione decisamente migliore dal punto di vista della leggibilità, in quanto più vicino al nostro modo di scrivere codice:

```
module mux2x1 (output reg a, input b, input c, input select);
    always @(*) begin
```

```

        if(select == 1)
            a = b;
        else
            a = c;
        end
    endmodule

```

▼ Testbench e verifica

```

`timescale 1ns/1ps

module mux2x1_tb;

    // segnali per collegarsi al modulo
    reg in0, in1, sel;
    wire out;

    // istanza del modulo da testare
    mux2x1 uut (
        .out(out),
        .in1(in1),
        .in0(in0),
        .sel(sel)
    );

    initial begin
        $display("in1 in0 sel | out");
        $display("-----");

        // test per tutte le 8 combinazioni
        for (integer i = 0; i < 8; i = i + 1) begin
            {in1, in0, sel} = i[2:0]; // assegna ogni combinazione
            #1; // attesa per stabilizzazione
            $display(" %b  %b  %b | %b", in1, in0, sel, out);
        end

        $finish;
    end

endmodule

```

In uscita, otteniamo la [tabella a lato](#).

Come previsto:

- se `sel = 0`, `out = c` ;
- se `sel = 1`, `out = b` .

b	c	sel	out
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Tabella risultante

Esercizio 3 - latch trasparente

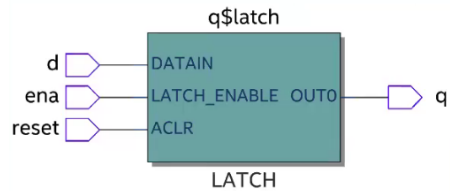
▼ Creatore originale: @Gianbattista Busonera

Si chiede sostanzialmente:

Se enableA è uguale a 1, l'uscita q deve diventare pari a d;

Se il reset è attivo (pari a 1), l'uscita q deve diventare pari a 0.

Il reset deve essere di tipo asincrono!



▼ Soluzione

```
module dlatch (q, d, ena, reset);
    output reg q;
    input d;
    input ena;
    input reset;
    // DEVO NECESSARIAMENTE USARE UN BLOCCO ALWAYS
    always @(d or ena or reset) begin
        if(reset) // il reset ha priorità rispetto al valore dell'ingresso!
            q = 0;
        else if(ena) // solo se enable a = 1 ⇒ q = d;
            q = d;
    end
endmodule
```



Dapprima inserire d nei parametri di sensibilità potrebbe sembrare insensato ma immaginiamo il caso:

- reset = 0; (costante)
- enableA = 1; (costante)
- d cambia da 0 a 1;

In questo caso sarebbe necessario che l'uscita q vari come è variato d.

▼ Testbench

```
`timescale 1ns/1ps

module tb_dlatch();
    // Segnali di test
    reg d;
    reg ena;
    reg reset;
    wire q;

    // Istanza del modulo da testare
    dlatch uut (
        .q(q),
```

```

        .d(d),
        .ena(ena),
        .reset(reset)
    );

    // Inizializzazione
    initial begin
        $dumpfile("dlatch_waveform.vcd");
        $dumpvars(0, tb_dlatch);

        // Inizializzazione ingressi
        d = 0;
        ena = 0;
        reset = 1;

        // Test del reset
        #10 reset = 0;

        // Test enable con d=0
        #10 ena = 1;
        #10 d = 1;
        #10 d = 0;

        // Test disable
        #10 ena = 0;
        #10 d = 1; // q non dovrebbe cambiare
        #10 d = 0;

        // Test enable di nuovo
        #10 ena = 1;
        #10 d = 1;

        // Test reset asincrono
        #10 reset = 1;
        #10 reset = 0;

        // Fine simulazione
        #20 $finish;
    end

    // Monitor per visualizzare i cambiamenti
    initial begin
        $monitor("Time = %0t: reset = %b, ena = %b, d = %b, q = %b",
            $time, reset, ena, d, q);
    end
endmodule

```

```

Time = 0: reset = 1, ena = 0, d = 0, q = 0
Time = 10000: reset = 0, ena = 0, d = 0, q = 0
Time = 20000: reset = 0, ena = 1, d = 0, q = 0
Time = 30000: reset = 0, ena = 1, d = 1, q = 1
Time = 40000: reset = 0, ena = 1, d = 0, q = 0
Time = 50000: reset = 0, ena = 0, d = 0, q = 0
Time = 60000: reset = 0, ena = 0, d = 1, q = 0
Time = 70000: reset = 0, ena = 0, d = 0, q = 0
Time = 80000: reset = 0, ena = 1, d = 0, q = 0
Time = 90000: reset = 0, ena = 1, d = 1, q = 1
Time = 100000: reset = 1, ena = 1, d = 1, q = 0
Time = 110000: reset = 0, ena = 1, d = 1, q = 1
dlatch_tb.v:50: $finish called at 130000 (1ps)

```

Esercizio 4 - FSM

▼ Creatore originale: @Gianbattista Busonera



Si rimanda link alla generica realizzazione di una FSM: [LINK](#)

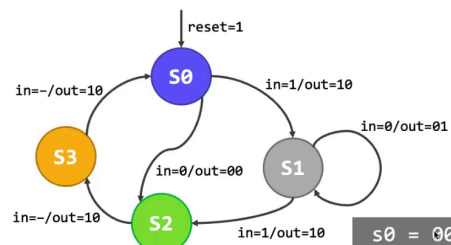
Si voglia descrivere in verilog una macchina a stati con quattro stati codificati come segue:

s0	00
s1	01
s2	10
s3	11

Si riporta inoltre l'FSM da realizzare:

La macchina a stati presenta un reset sincrono attivo alto, due bit di uscita (out) e un bit di ingresso (in).

Si noti che si tratta di una FSM di Mealy in quanto l'uscita non dipende solamente dallo stato corrente ma anche dal valore corrente dell'ingresso.



▼ Soluzione

```

module es4(out, clk, reset, in);
    output reg [1:0] out; // serve farla di tipo reg in quanto la utilizzeremo
    // nei blocchi always!
    input clk, reset, in;
    // SOLO PER COMODITA' decidiamo di utilizzare dei parametri costanti per descrivere
    // gli stati:
    parameter s0 = 2'b00,
               s1 = 2'b01,
               s2 = 2'b10,
               s3 = 2'b11;

    // Descriviamo ora la memoria (FF) che contiene lo stato corrente.
    reg [1:0] cs; // current state
    // è però necessario anche utilizzare una variabile d'appoggio che descriva lo stato
    // futuro, di tipo reg poichè finirà nel blocco always
    reg [1:0] ns; // next state

```



```

// BLOCCO ALWAYS CHE DESCRIVO LO STATO CORRENTE che cambia solo su fronte del clock!
always @(posedge clk) begin
    if(reset) cs = s0; // reset sincrono con priorità allo stato 00
    else cs = ns; // se non devo fare reset passo al next state sul fronte del clock.
end

// BLOCCO ALWAYS CHE CALCOLA IL NEXT STATE
// si ricordi che lo stato futuro dipende solo dal valore degli ingressi e dallo
// stato corrente
always @(in or cs) begin
    case(cs)
        s0: begin
            if(in) ns = s1;
            else ns = s2;
        end
        s1: begin
            if(in) ns = s2;
            else ns = s1;
        end
        s2: ns = s3; // indipendentemente dall'ingresso
        s3: ns = s0; // indipendentemente dall'ingresso
    endcase
end

// BLOCCO ALWAYS CHE DESCRIVE L'USCITA
// Si ricordi che l'uscita dipende solo dal valore corrente dell'ingresso
// e dallo stato corrente.
always @(in or cs) begin
    case(cs)
        s0: begin
            if(in) out = 2'b10;
            else out = 2'b00;
        end
        s1: begin
            if(in) out = 2'b10;
            else out = 2'b01;
        end
        s2: out = 2'b10; // indipendentemente dall'ingresso
        s3: out = 2'b10; // indipendentemente dall'ingresso
    endcase
end
endmodule

```

Potenzialmente potevamo unire le uscite e gli stati futuri nello stesso blocco always ma, per maggiore chiarezza, ho preferito separarli.



Se anche uno degli stati non fosse previsto, converrebbe inserire una condizione di default nel codice così da coprire in ogni caso tutti i casi!

▼ Testbench

Si riporta il codice dichiaratamente generato per il testbench:

```

`timescale 1ns / 1ps

module tb_es4;

    // Segnali
    reg clk;
    reg reset;
    reg in;
    wire [1:0] out;

    // Istanziamento del modulo
    es4 uut (
        .out(out),
        .clk(clk),
        .reset(reset),
        .in(in)
    );

    // Stato corrente interno (accesso diretto per debug)
    wire [1:0] state = uut.cs;

    // Clock con periodo 10ns
    always #5 clk = ~clk;

    // Monitoraggio ogni 5ns
    always #5 begin
        #0.1 // leggero ritardo per garantire aggiornamento variabili
        $display("t=%0dns | clk=%b | reset=%b | in=%b | state=%02b | out=%02b",
            $time, clk, reset, in, state, out);
    end

    // Sequenza di test sincrona
    initial begin
        // Inizializzazione
        clk = 0;
        reset = 1;
        in = 0;

        #10; // t = 10ns
        reset = 0;

        // Test transizioni con input cambiati tra i fronti
        in = 1; // S0 → S1
        @(posedge clk);

        in = 1; // S1 → S2
        @(posedge clk);

        in = 0; // S2 → S3
        @(posedge clk);

        in = 1; // S3 → S0
        @(posedge clk);
    end
endmodule

```

```

in = 0; // S0 → S2
@(posedge clk);

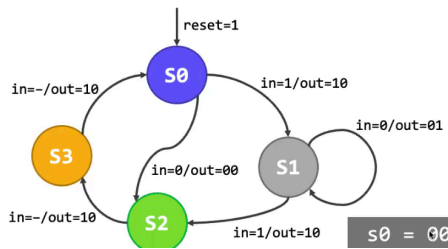
@(posedge clk); // S2 → S3

$display("Fine test.");
$finish;
end

endmodule

```

Che da il seguente output:



t=5ns	clk=1	reset=1	in=0	state=xx	out=xx
t=10ns	clk=0	reset=0	in=1	state=00	out=00
t=15ns	clk=1	reset=0	in=1	state=00	out=10
t=20ns	clk=0	reset=0	in=1	state=01	out=10
t=25ns	clk=1	reset=0	in=1	state=01	out=10
t=30ns	clk=0	reset=0	in=0	state=10	out=10
t=35ns	clk=1	reset=0	in=0	state=10	out=10
t=40ns	clk=0	reset=0	in=1	state=11	out=10
t=45ns	clk=1	reset=0	in=1	state=11	out=10
t=50ns	clk=0	reset=0	in=0	state=00	out=00
t=55ns	clk=1	reset=0	in=0	state=00	out=00
t=60ns	clk=0	reset=0	in=0	state=10	out=10
t=65ns	clk=1	reset=0	in=0	state=10	out=10

Tempo	clk	reset	in	Stato	Out	✅ Stato atteso	✅ Uscita attesa
5ns	1	1	0	00	00	S0 (da reset)	00
10ns	0	0	1	00	10	S0	10
15ns	1	0	1	01	10	S1	10
20ns	0	0	1	01	10	S1	10
26ns	1	0	0	10	10	S2	10
31ns	0	0	0	10	10	S2	10
36ns	1	0	1	11	10	S3	10
41ns	0	0	1	11	10	S3	10
46ns	1	0	0	00	00	S3 → S0	00
51ns	0	0	0	00	00	S0	00
56ns	1	0	0	10	10	S0 + in=0 ⇒ S2	10
61ns	0	0	0	10	10	S2	10

▼ GTKWave e visualizzazione forme d'onda

Per visualizzare le forme d'onda è necessario inserire il seguente frammento di codice al testbench precedente e ricompilare:

```
initial begin
    $dumpfile("waveform.vcd"); // Nome del file da creare
    $dumpvars(0, es04); // Dump di tutte le variabili del modulo "es04"
end
```

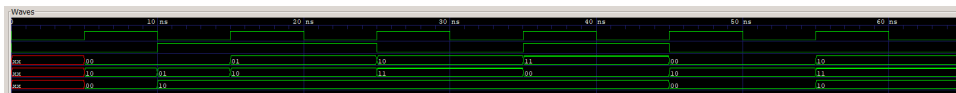
E dovremmo ottenere, dopo la ricompilazione e riesecuzione il seguente output:

```
PS C:\Users\gianb\Desktop\UNI\ELAP\Lab\E05\ES04> iverilog -o es04_tb es04.v es04_tb.v
PS C:\Users\gianb\Desktop\UNI\ELAP\Lab\E05\ES04> vvp es04_tb
VCD info: dumpfile waveform.vcd opened for output.
t=5ns | clk=1 | reset=1 | in=0 | state=00 | out=00
t=10ns | clk=0 | reset=0 | in=1 | state=00 | out=10
t=15ns | clk=1 | reset=0 | in=1 | state=01 | out=10
t=20ns | clk=0 | reset=0 | in=1 | state=01 | out=10
t=26ns | clk=1 | reset=0 | in=0 | state=10 | out=10
t=31ns | clk=0 | reset=0 | in=0 | state=10 | out=10
t=36ns | clk=1 | reset=0 | in=1 | state=11 | out=10
t=41ns | clk=0 | reset=0 | in=1 | state=11 | out=10
t=46ns | clk=1 | reset=0 | in=0 | state=00 | out=00
t=51ns | clk=0 | reset=0 | in=0 | state=00 | out=00
t=56ns | clk=1 | reset=0 | in=0 | state=10 | out=10
t=61ns | clk=0 | reset=0 | in=0 | state=10 | out=10
```

Scrivendo dunque su console:

```
gtkwave waveform.vcd
```

Si aprirà il programma gtkwave e, inserendo i segnali d'interesse nel grafico vediamo:



Esercizio 5 - Shift-Register completo

Si richiede di implementare in Verilog un componente adatto a realizzare la funzionalità di uno shift-register parametrico completo (con uscita q) con reset asincrono tale per cui:

- ctrl = 0 → nessuna uscita modificata
- ctrl = 1 → shift a destra, data[N-1] entra come MSB
- ctrl = 2 → shift a sinistra, data[0] entra come LSB
- ctrl = 3 → caricamento parallelo di data[N-1:0]

▼ Soluzione

```
module shift_register(q, clk, reset, ctrl, data);
    parameter N = 8;
    output [N-1:0] q;
    input clk, reset;
    input [1:0] ctrl;
    input [N-1:0] data;

    reg [N-1:0] mem; // DEFINISCO LA MEMORIA DEL MIO CIRCUITO
```

```

always @(posedge clk or posedge reset) begin
  if(reset)
    mem = 0;
  else begin
    case (ctrl)
      2'b00 : begin
        // no operation
      end
      2'b01 : begin // SHIFT A DESTRA
        mem = {data[N-1], mem[N-1:1]}; // perdo l'ultimo bit e inserisco data[N-1] come
        // MSB come da specifica
      end
      2'b10 : begin // SHIFT A SINISTRA
        mem = {mem[N-2:0], data[0]}; // perdo il primo bit e inserisco data[0] come LSB
      end
      2'b11 : begin
        mem = data;
      end
    endcase
  end
end

assign q = mem; // la mia uscita prende il valore della memoria.
endmodule

```

Esercizi extra

Non farò gli esercizi aggiuntivi perchè sono bene o male già stati trattati (FF e ALU) nella sezione di teoria relativa al linguaggio verilog e sono anche più semplici degli esercizi in precedenza.