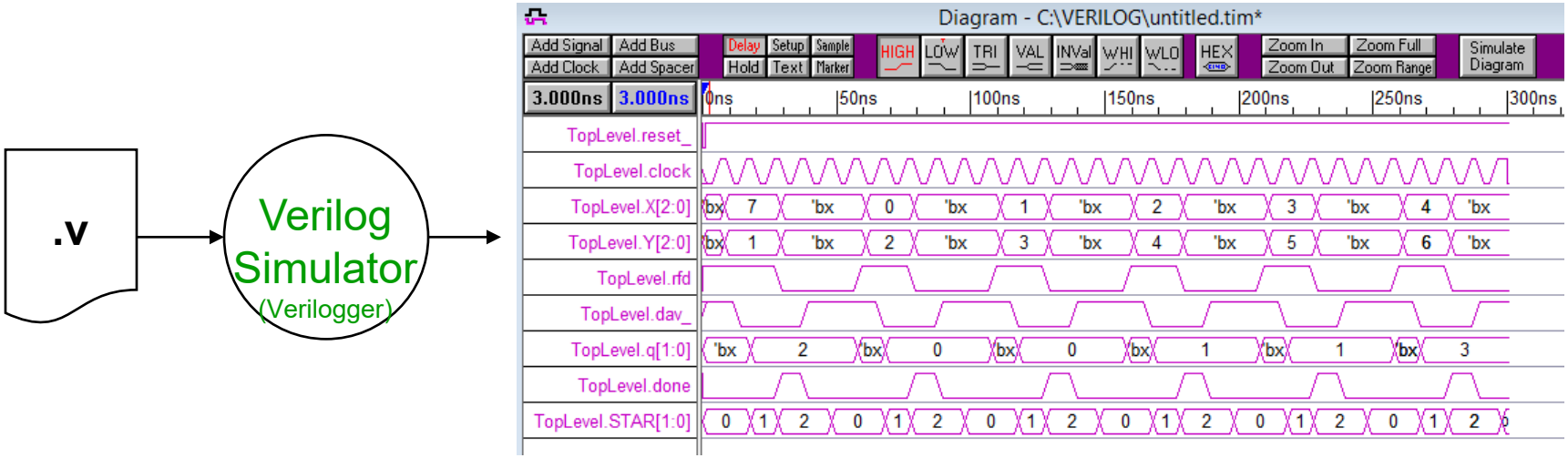
Esercitazione 1
Verilog Tools

Analisi del Diagramma Temporale di una rete logica



File Verilog

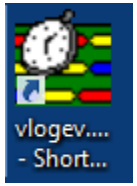
Tool

Diagramma Temporale

Istallazione di Verilogger

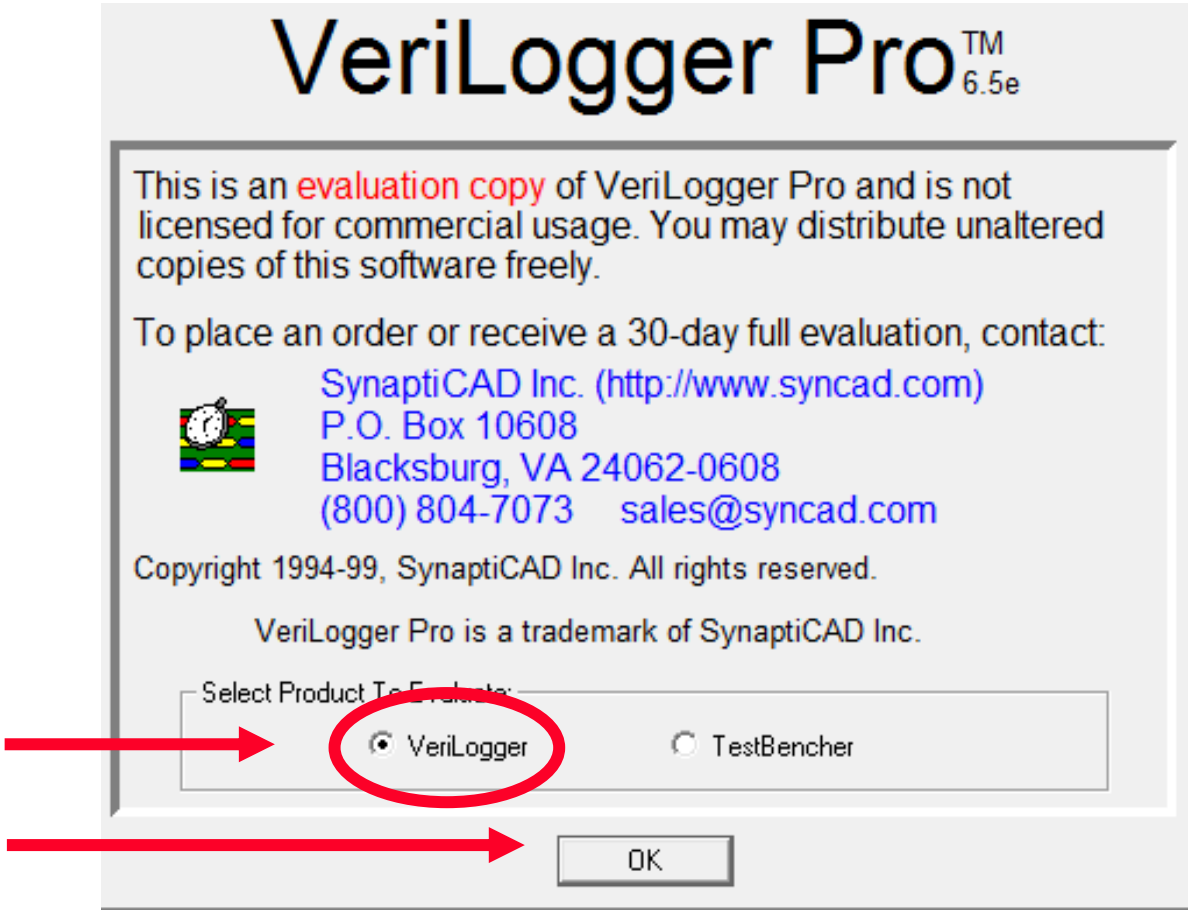
- Distribuibile gratuitamente

- Istruzioni per l'istallazione:
  - Sul proprio PC (Windows)
    - Scaricare: https://arcal.diism.unisi.it/tools1/verilog.exe
    - Eseguire (verrà creato il folder C:\VERILOG)
    - Crearsi un link (es. dal Desktop) al file C:\VERILOG\vlogev.exe
  - In Laboratorio
    - (gia' istallato) → individuare l'icona relativa

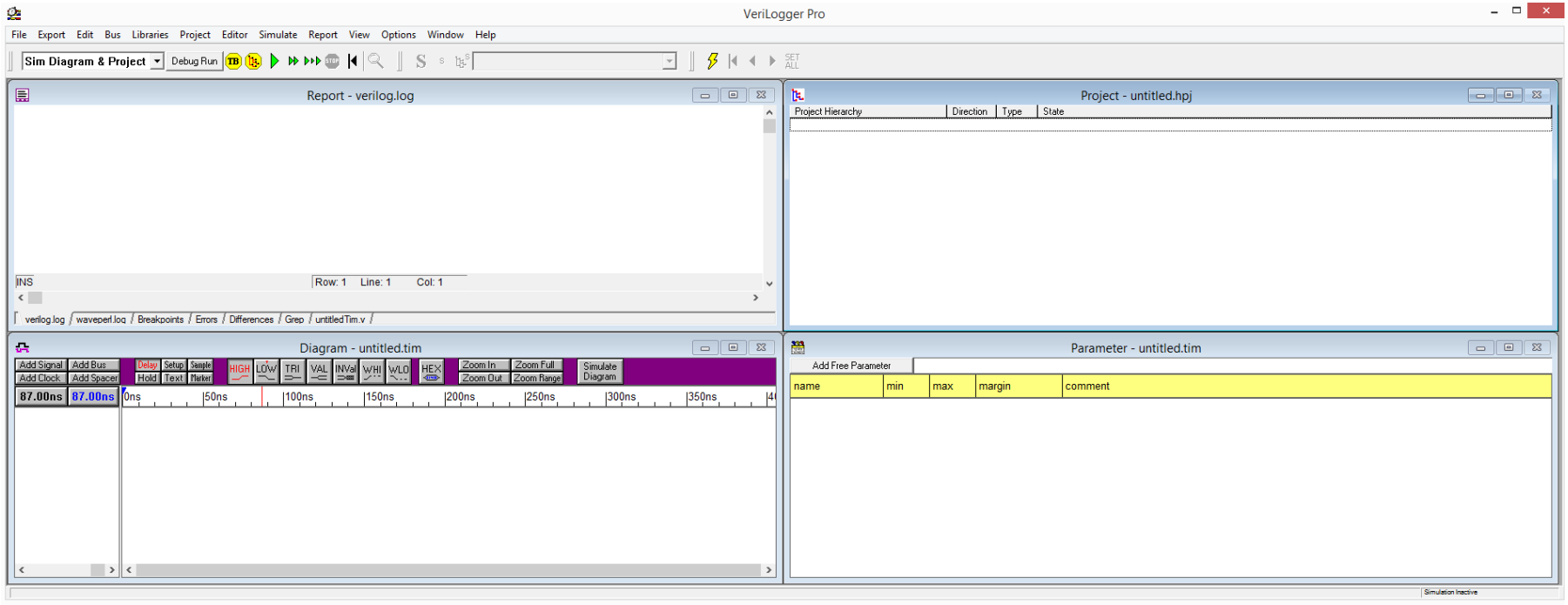


Uso di Verilogger: un primo esempio

- Lanciare il programma Verilogger (vlogev.exe)
  - Selezionare nella prima finestra "VeriLogger"

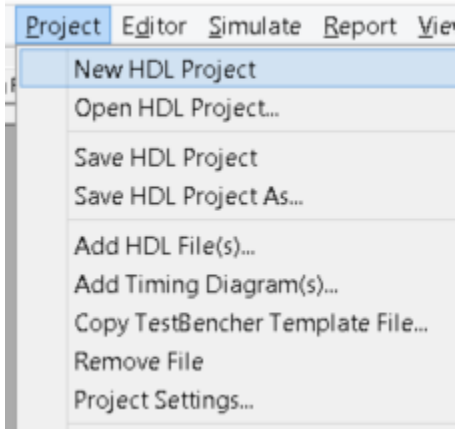


Verilogger: schermata iniziale

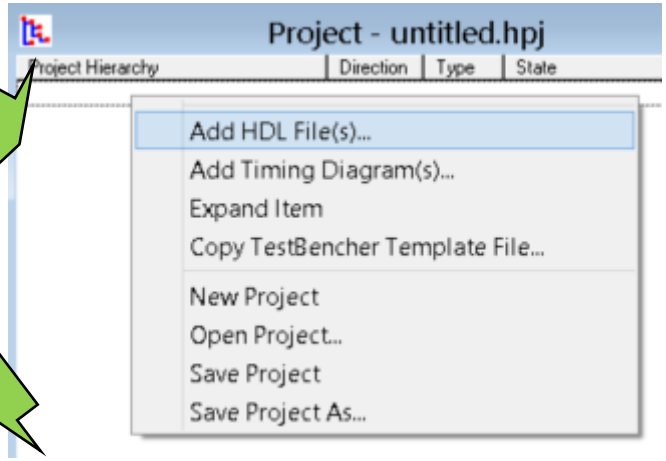


- Sono presenti 4 finestre
  - Report
  - Project
  - Parameter
  - Diagram

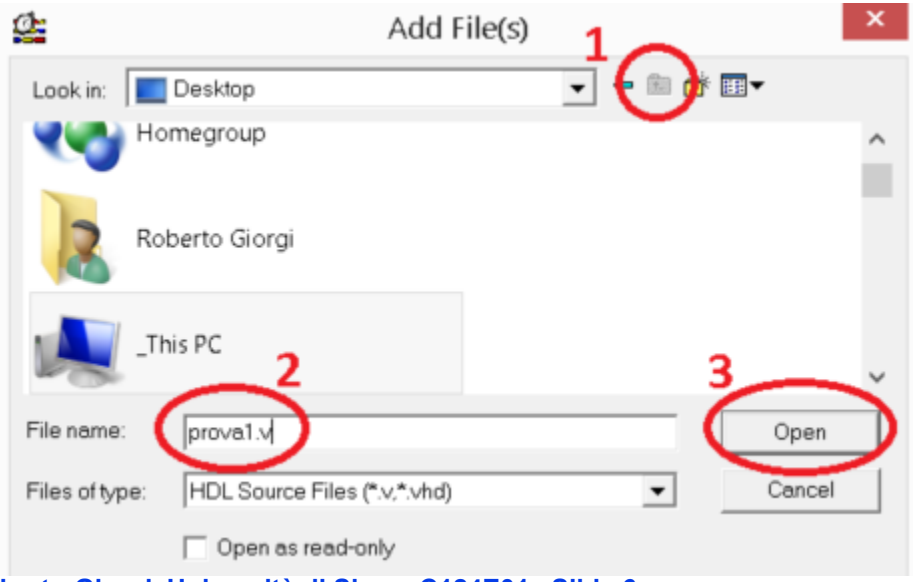
Creazione di un progetto VERILOG



- Selezionare: Project/New_HDL_Project



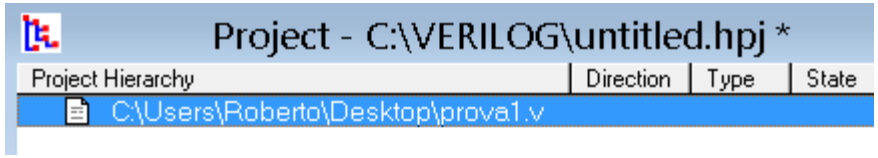
- Poi: tasto destro del mouse nella finestra "Project" e selezionare: "Add HDL File(s)"



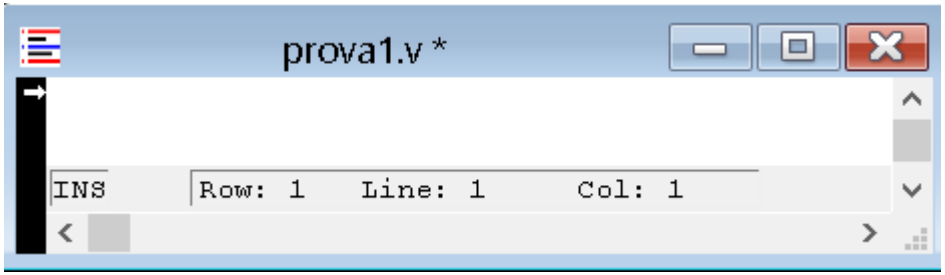
- 1: scegliere la cartella di lavoro (es. Desktop)
- 2: scrivere il nome del file Verilog (es. Prova1.v)
- 3: aprire (Open) → il file viene creato (se non esiste)

Inserimento codice VERILOG

- Confermare la creazione del file
- Fare doppio click sul riga del file HDL (verilog)



- Inserire il codice Verilog
  (o fare copy-paste di esempi esistenti)



- Inserire ad esempio il codice di una porta NAND
  (v. codice di esempio nella slide successiva)
- Salvare ! → CTRL+S sulla finestra "prova1.v" e
  → CTRL+S sulla finestra "Project"
  (es. Assegnare al progetto il nome "my1prj.hpj")

Esempio 1: Porta nand

```
`timescale 1ns / 1ps
//test the NAND gate
module encoder_testbench; //module with no ports
  reg A, B;
  wire C;

  //instantiate your circuit
  some_logic_component S1(C, A, B);

  //Behavioral code block generates stimulus to test circuit
  initial
    begin
      A = 1'b0; B = 1'b0;
      #50; $display("A = %b, B = %b, NAND output C = %b \n", A, B, C);
      A = 1'b0; B = 1'b1;
      #50 $display("A = %b, B = %b, NAND output C = %b \n", A, B, C);
      A = 1'b1; B = 1'b0;
      #50 $display("A = %b, B = %b, NAND output C = %b \n", A, B, C);
      A = 1'b1; B = 1'b1;
      #50 $display("A = %b, B = %b, NAND output C = %b \n", A, B, C);
    end
endmodule
```

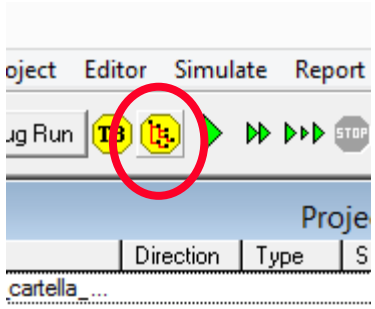
TESTBENCH

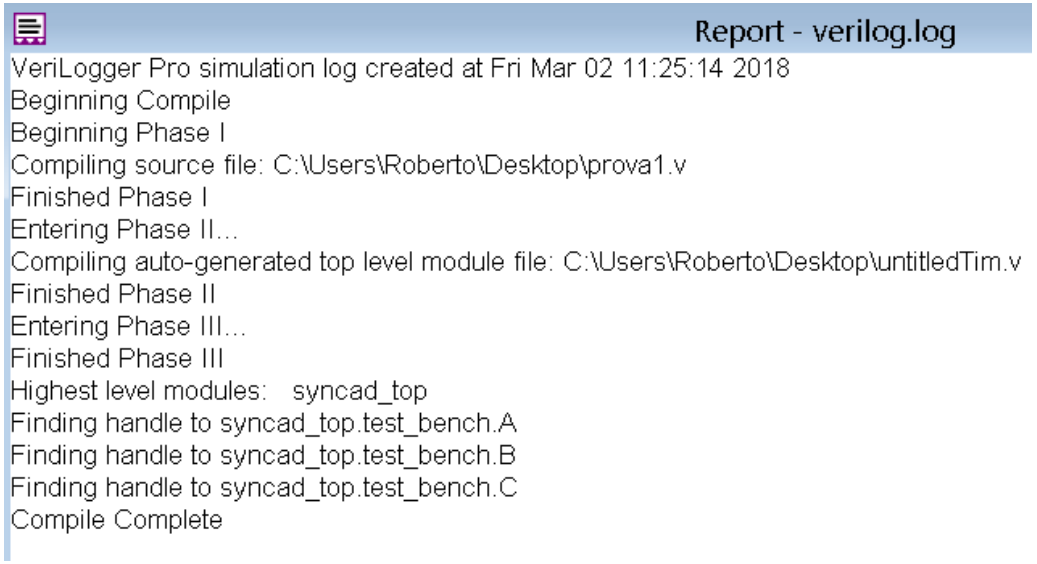
```
//create a NAND gate out of an AND and an Inverter
module some_logic_component (c, a, b);
      // declare port signals
  output c;
  input a, b;
     // declare internal wire
  wire d;
     //instantiate structural logic gates
  and a1(d, a, b); //d is output, a and b are inputs
  not n1(c, d);    //c is output, d is input
endmodule
```

ARCHITECTURAL MODULE

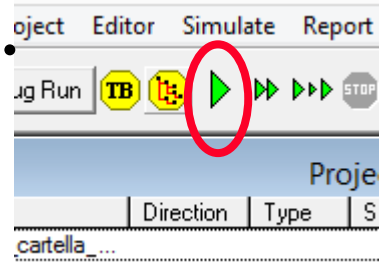
Compilazione e generazione del diagramma temporale

- **Compilare**



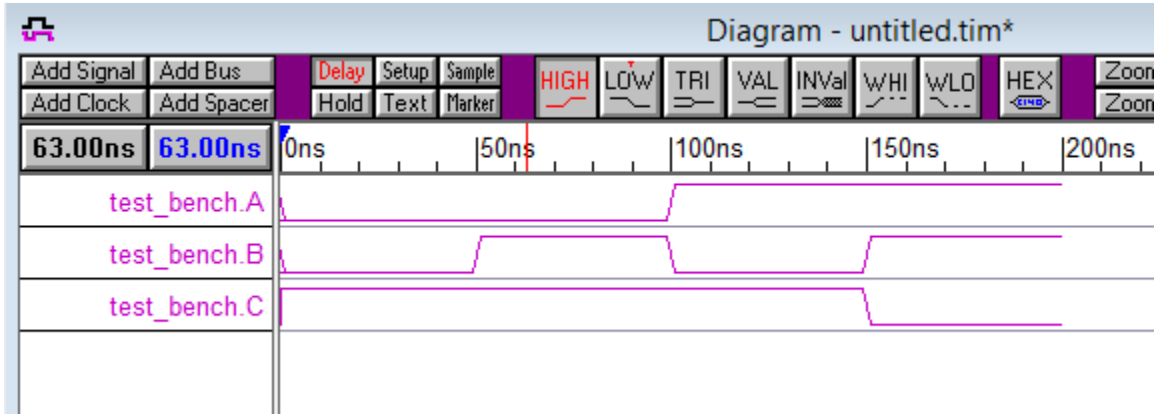


- **Generare il diagramma temporale**



(risultato nella slide successiva)

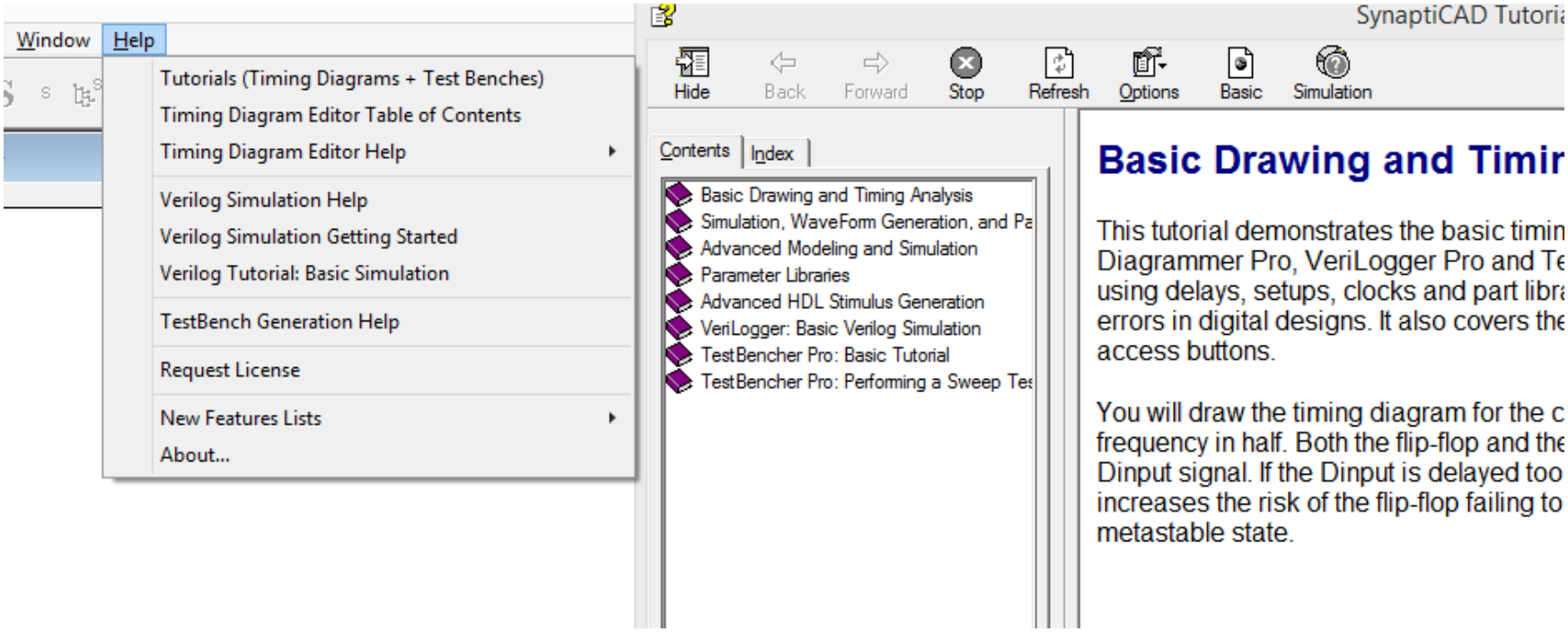
NAND: diagramma e report



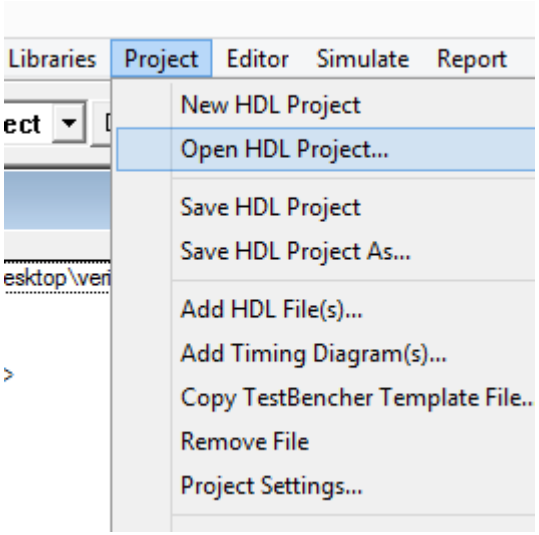
Report - C:\VERILOG\verilog.log

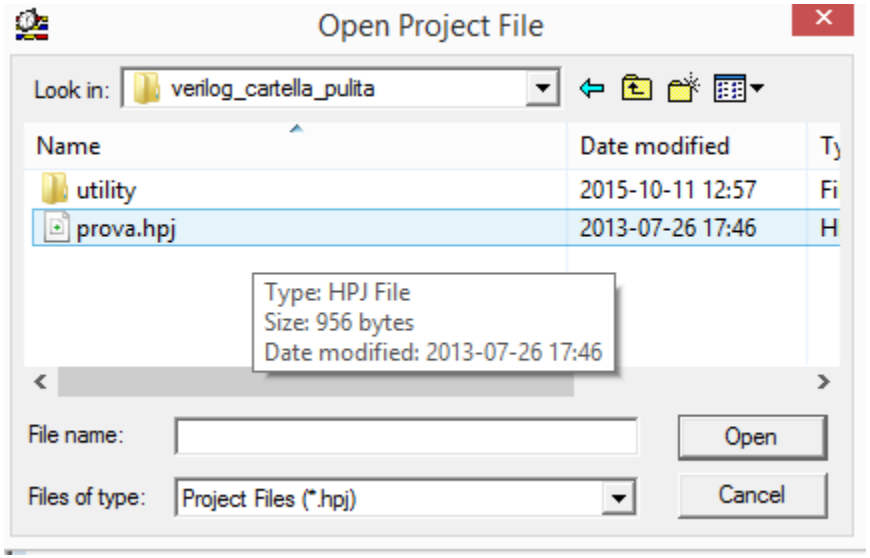
```
Finding handle to syncad_top.test_bench.C
Compile Complete
.
Running...
A = 0, B = 0, Nand output C = 1

A = 0, B = 1, Nand output C = 1

A = 1, B = 0, Nand output C = 1

A = 1, B = 1, Nand output C = 0

0 Errors, 0 Warnings
Compile time = 0.02000, Load time = 0.28400, Execution time = 0.02400
```

Ulteriore documentazione dettagliata

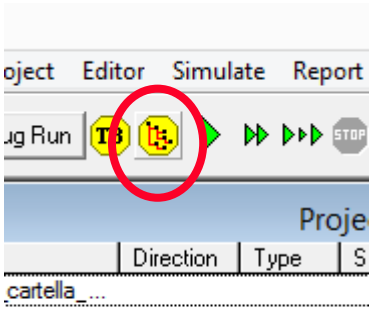


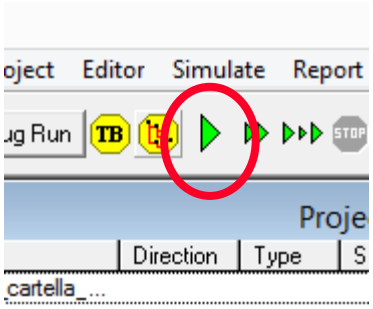
Esempio 2: selezione di un progetto esistente

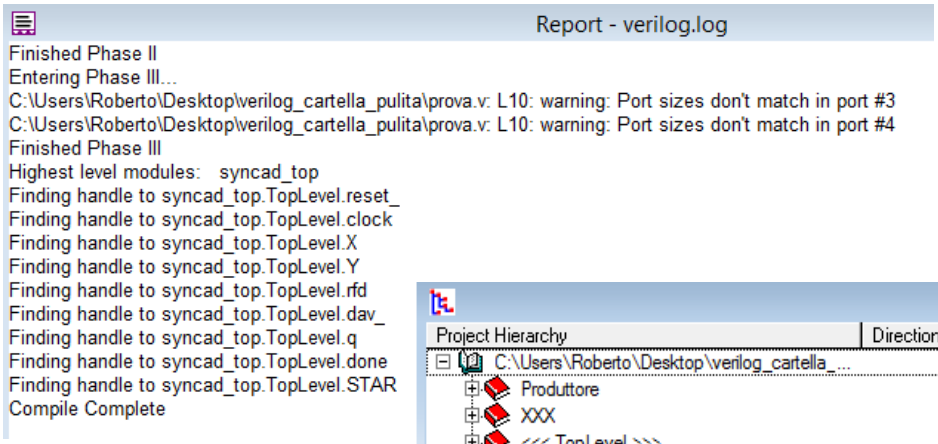


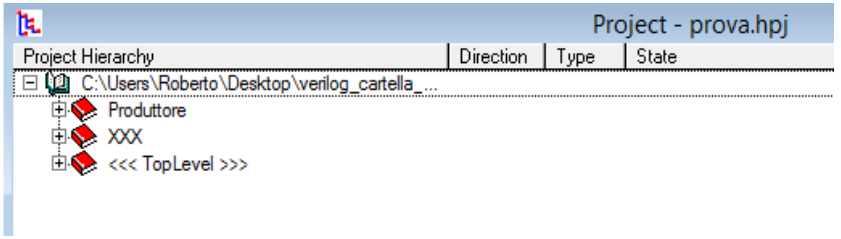


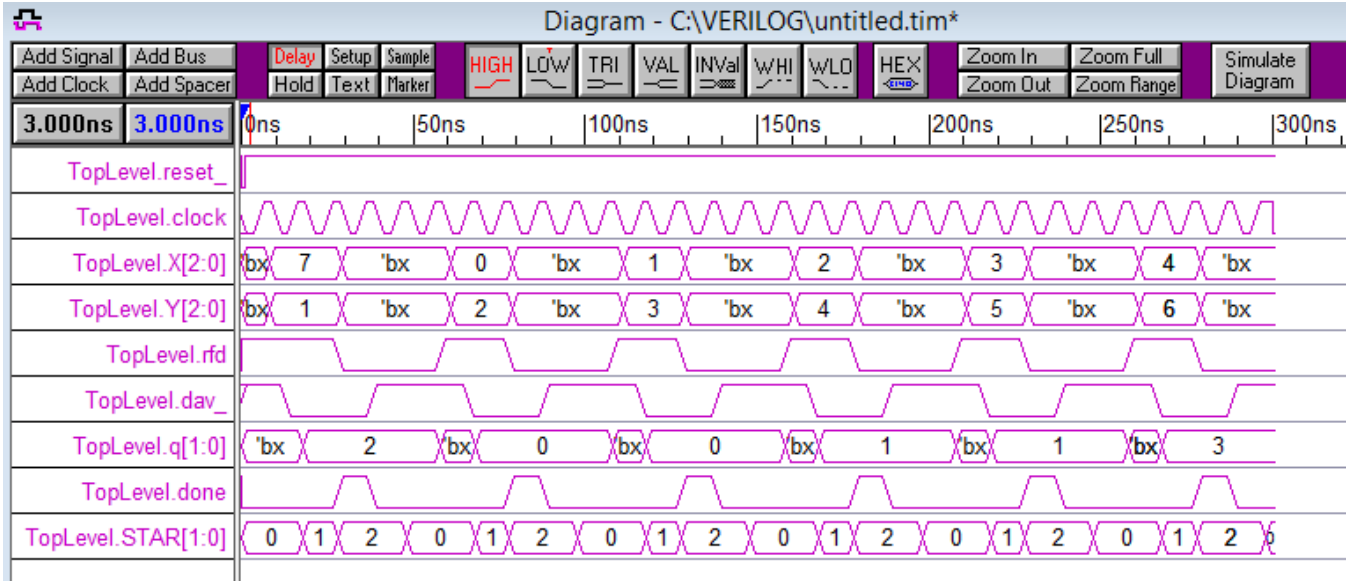
Esempio 2 (segue): compilazione e generazione diag. temporale







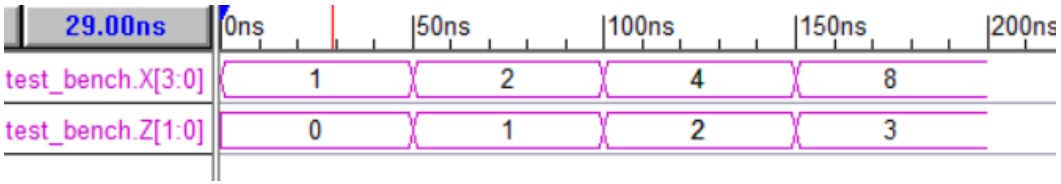




Esempio 3: Encoder a 4-bit

```
`timescale 1ns / 1ps
//test a 4-bit ENCODER
module test_bench;
  reg [3:0] X;
  wire[1:0] Z;

  //instantiate your circuit
  encoder ENC(X,  Z);

  //Behavioral code block generates stimulus to test circuit
  initial
    begin
      X = 4'b0001;
      #50; $display("X = %b, ENCODER output Z = %b \n", X, Z);
      X = 4'b0010;
      #50 $display("X = %b, ENCODER output Z = %b \n", X, Z);
      X = 4'b0100;
      #50 $display("X = %b, ENCODER output Z = %b \n", X, Z);
      X = 4'b1000;
      #50 $display("X = %b, ENCODER output Z = %b \n", X, Z);
end
endmodule
```

```
module encoder(x,  z);
   input [3:0] x;
   output[1:0] z; reg[1:0] z;

   always @(x) casex(x)
      4'b0001: z<=2'b00;
      4'b0010: z<=2'b01;
      4'b0100: z<=2'b10;
      4'b1000: z<=2'b11;
      default: z<=2'bzz;
   endcase
endmodule
```



Esercizi

- Provare a descrivere in Verilog le reti combinatorie notevoli che abbiamo visto a lezione

ESERCIZI in LABORATORIO – RETI DI MEALY/MOORE (ESERCITAZIONE/LAB #02) – versione aggiornata del 16/10/2020

OBIETTIVI

1) **Capire la sintesi delle reti di Mealy e Moore**
2) **Uso del simulatore VERILOGGER**

1) Capire la sintesi delle reti di Mealy e Moore

Es. n.6 e n.7 del 04/11/2016: https://arcal.diism.unisi.it/esercizi1/c1161104-sol.pdf

6) [8] Sintetizzare una rete sequenziale utilizzando il modello di Mealy-Ritardato con un ingresso X su un bit e una uscita Z su due bit che funziona nel seguente modo: devono essere riconosciute le sequenze non interallacciate 1,1,0,0 e 0,1,0,1; l'uscita Z[1] va a 1 se si presenta una delle due sequenze mentre Z[0] dice quale sequenza si e' presentata (Z[0]=1 se si presenta 1,1,0,0; Z[0]=0 altrimenti). Rappresentare la macchina a stati finiti per tale rete di Mealy-Ritardato, la tabella delle transizioni, le equazioni booleane delle reti CN1 e CN2 e il circuito sequenziale sincronizzato basato su flip-flop D.

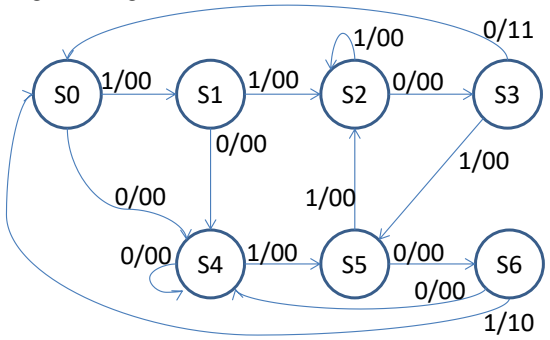
7) [8] Descrivere e sintetizzare in Verilog la rete sequenziale descritta nell'esercizio 6. Tracciare il diagramma di temporizzazione come verifica della correttezza dell'unità XXX (il modulo TopLevel e' riportato in calce). Nota: si puo' svolgere l'esercizio con ausilio del simulatore oppure su carta e salvare una copia dell'output e del programma nella cartella ARCALxxxxxxx

```
module TopLevel;
reg reset_;initial begin reset_=0; #22 reset_=1; #400; end
reg clock ;initial begin clock=0; forever #5 clock <=(!clock); end
reg X;
wire [1:0] Z;
wire [2:0] STAR=Xxx.STAR;
wire Z1=Xxx.z[1];
wire Z0=Xxx.z[0];
initial begin X=0;
wait(reset_==1); #5
  @(posedge clock); X<=1; @(posedge clock); X<=1; @(posedge clock); X<=0; @(posedge clock); X<=0;
  @(posedge clock); X<=0; @(posedge clock); X<=0; @(posedge clock); X<=0; @(posedge clock); X<=0;
  @(posedge clock); X<=1; @(posedge clock); X<=1; @(posedge clock); X<=0; @(posedge clock); X<=0;
  @(posedge clock); X<=0; @(posedge clock); X<=1; @(posedge clock); X<=0; @(posedge clock); X<=1;
  @(posedge clock); X<=0; @(posedge clock); X<=0; @(posedge clock); X<=0; @(posedge clock); X<=0;
  $finish;
end
XXX Xxx(X,Z,clock,reset_);
endmodule
```

ESERCIZIO 6

In corrispondenza del pattern $X_{t-3},X_{t-2},X_{t-1},X_t = 1,1,0,0$ ottengo → $Z_{t+1} = 11$; (ricordare che e' richiesto Mealy ritardato). In corrispondenza del pattern $X_{t-3},X_{t-2},X_{t-1},X_t = 0,1,0,1$ ottengo → $Z_{t+1} = 10$.

Diagramma degli Stati



Schema di riferimento per Mealy Ritardato

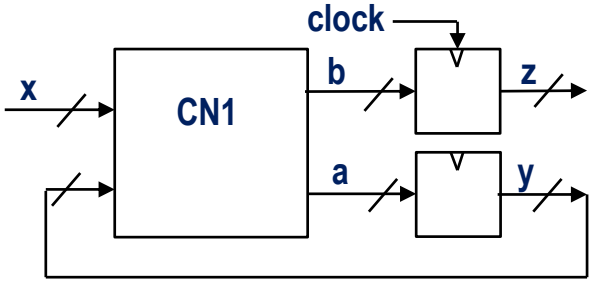


Tabella delle Transizioni (Tabella degli Stati)

TABELLA DELLE TRANSIZIONI

STATO ATTUALE \ x	0	1
S0	S4/00	S1/00
S1	S4/00	S2/00
S2	S3/00	S2/00
S3	S0/11	S5/00
S4	S4/00	S5/00
S5	S6/00	S2/00
S6	S4/00	S0/10

STATO SUCCES-SIVO/USCITA

Scelta della codifica degli Stati

STATO \ CODIFICA $y_2y_1y_0$	
S0	000
S1	001
S2	011
S3	010
S4	100
S5	101
S6	111
X	110

(completare il n. di stati a una potenza di 2)

Rappresentazione della Tabella degli Stati in Formato più comodo per la sintesi

OVVERO

y_1y_0 \ y_2x	00	01	11	10
00	S4/00	S1/00	S5/00	S4/00
01	S4/00	S2/00	S2/00	S6/00
11	S3/00	S2/00	S0/10	S4/00
10	S0/11	S5/00	X/XX	X/XX

STATO SUCCES-SIVO/USCITA

OVVERO

y_1y_0 \ y_2x	00	01	11	10
00	100/00	001/00	101/00	100/00
01	100/00	011/00	011/00	111/00
11	010/00	011/00	000/10	100/00
10	000/11	101/00	XXX/XX	XXX/XX

a2,a1,a0,z1,z0

Sintesi della variable 'a' (stato successivo o ingresso dello STATUS REGISTER -- STAR):

y_1y_0 \ y_2x	00	01	11	10
00	1	0	1	1
01	1	0	0	1
11	0	0	0	1
10	0	1	X	X

a_2

a2=/y1/x+y2/x+y2/y1/y0+y1/y0x

y_1y_0 \ y_2x	00	01	11	10
00	0	0	0	0
01	0	1	1	1
11	1	1	0	0
10	0	0	X	X

a_1

a1=/y2y1y0+y2/y1y0+/y1y0x

y_1y_0 \ y_2x	00	01	11	10
00	0	1	1	0
01	0	1	1	1
11	0	1	0	0
10	0	1	X	X

a_0

a0=/y2x+/y1x+y2/y1y0

Sintesi della variable 'b' (uscita successiva o ingresso dell'OUTPUT REGISTER -- OUTR):

y_1y_0 \ y_2x	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	0	0	1	0
10	1	0	X	X

b_1

b1=y2y1x+y1/y0/x

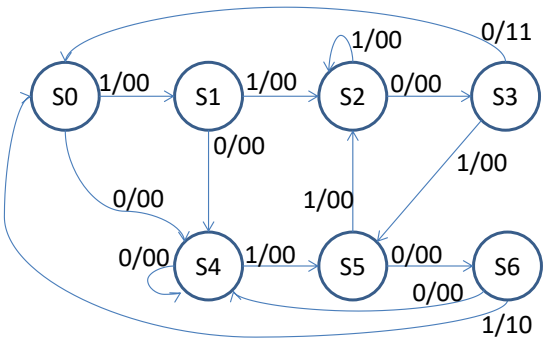
y_1y_0 \ y_2x	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	0	0	0	0
10	1	0	X	X

b_0

b0=y1/y0/x

ESERCIZIO 7

Diagramma degli stati:



Codice Verilog del modulo da realizzare (possibile soluzione con Mealy-Ritardato):

Implementazione basta sul template di Mealy-Ritardato	Implementazione basata sulle equazioni booleane

```
module XXX(x,z,clock,reset_);
input        clock,reset_,x;
output [1:0] z;
reg    [1:0] OUTR;
reg    [2:0] STAR;
parameter S0='B000,S1='B001,S2='B010,S3='B011,
   S4='B100,S5='B101,S6='B110;
always @(reset_==0) #1 begin STAR<=S0; OUTR<=0; end
assign z=OUTR;

always @(posedge clock) if(reset_==1) #3
casex(STAR)
   S0:begin STAR<=(x==0)?S4:S1; OUTR<=0; end
   S1:begin STAR<=(x==0)?S4:S2; OUTR<=0; end
   S2:begin STAR<=(x==0)?S3:S2; OUTR<=0; end
   S3:begin STAR<=(x==0)?S0:S5; OUTR<=(x==0)?'B11:'B00;
end
   S4:begin STAR<=(x==0)?S4:S5; OUTR<=0; end
   S5:begin STAR<=(x==0)?S6:S2; OUTR<=0; end
   S6:begin STAR<=(x==0)?S4:S0; OUTR<=(x==1)?'B10:'B00;
end
endcase
endmodule
```

```
module XXX(x,z,clock,reset_);
input        clock,reset_,x;
output [1:0] z;
wire   [2:0] y;
wire   [1:0] b;
wire   [2:0] a;
reg    [1:0] OUTR;
reg    [2:0] STAR;
always @(reset_==0) #1 begin STAR<=0; OUTR<=0; end
assign z=OUTR; assign y=STAR;

assign a[2]=((~y[1])&(~x))|(y[2]&(~x))|(y[2]&(~y[1])&(~y[0]))
       |(y[1]&(~y[0])&x);
assign a[1]=((~y[2])&y[1]&y[0])|(y[2]&(~y[1])&y[0])|((~y[1])&y[0]&x);
assign a[0]=((~y[2])&x)|((~y[1])&x)|(y[2]&(~y[1])&y[0]);

assign b[1]=(y[2]&y[1]&x)|(y[1]&(~y[0])&(~x));
assign b[0]=y[1]&(~y[0])&(~x);

always @(posedge clock) if(reset_==1) #3
begin OUTR<=b; STAR<=a; end

endmodule
```

Diagramma di Temporizzazione: (template)

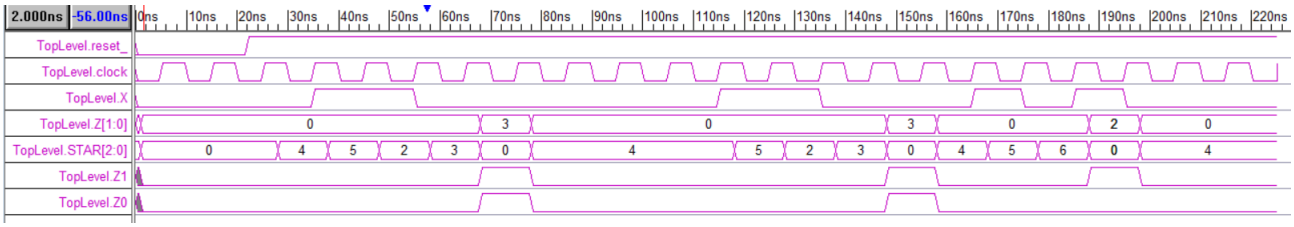
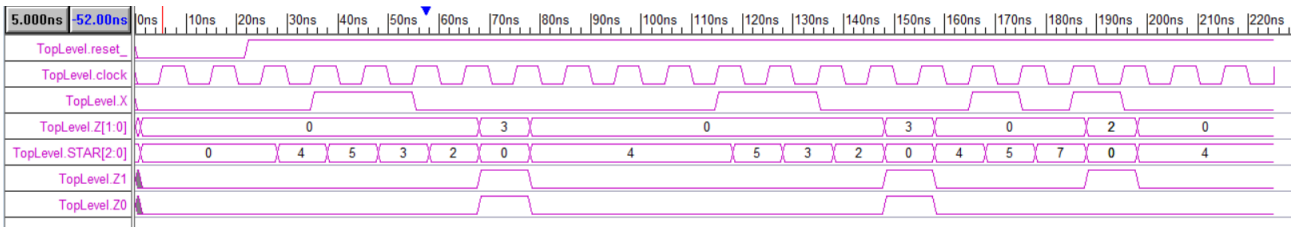
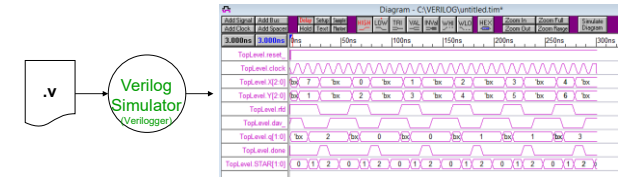


Diagramma di Temporizzazione: (equazioni booleane – nota 2→S3,3→S2,7→S6)

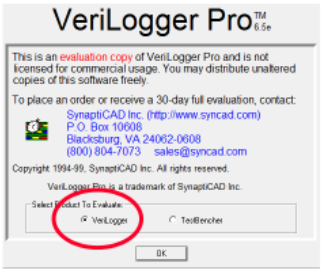


Analisi del Diagramma Temporale di una rete logica



Uso di Verilogger: un primo esempio

- Lanciare il programma Verilogger (vlogev.exe)
  - Selezionare nella prima finestra "VeriLogger"

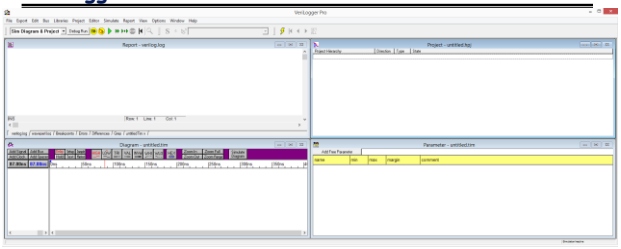


Istallazione di Verilogger

- Distribuibile gratuitamente

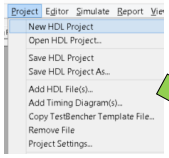
- Istruzioni per l'istallazione:
  - Sul proprio PC (Windows)
    - Scaricare: https://arcal.diism.unisi.it/tools1/verilog.exe
    - Eseguire (verrà creato il folder C:\VERILOG)
    - Crearsi un link (es. dal Desktop) al file C:\VERILOG\vlogev.exe
  - In Laboratorio
    - (gia' istallato) → individuare l'icona relativa

Verilogger: schermata iniziale

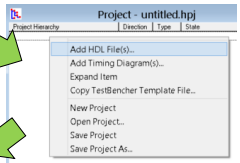


- Sono presenti 4 finestre
  - Report
  - Project
  - Parameter
  - Diagram

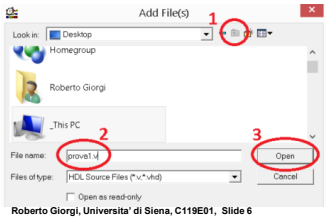
Creazione di un progetto VERILOG



- Selezionare: Project/New_HDL_Project



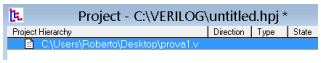
- Poi: tasto destro del mouse nella finestra "Project" e selezionare: "Add HDL File(s)"



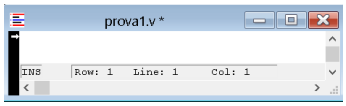
- 1: scegliere la cartella di lavoro (es. Desktop)
- 2: scrivere il nome del file Verilog (es. Prova1.v)
- 3: aprire (Open) → il file viene creato (se non esiste)

Inserimento codice VERILOG

- Confermare la creazione del file
- Fare doppio click sul riga del file HDL (verilog)



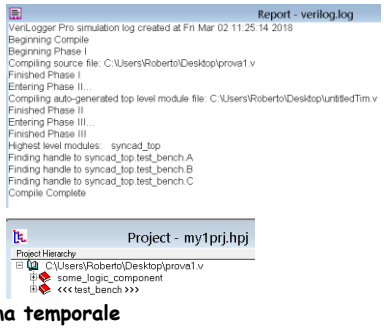
- Inserire il codice Verilog (o fare copy-paste di esempi esistenti)



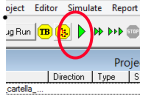
- Inserire ad esempio il codice di una porta NAND (v. codice di esempio nella slide successiva)
- Salvare ! → CTRL+S sulla finestra "prova1.v" e
  → CTRL+S sulla finestra "Project"
  (es. Assegnare al progetto il nome "my1prj.hpj")

Compilazione e generazione del diagramma temporale

- Compilare

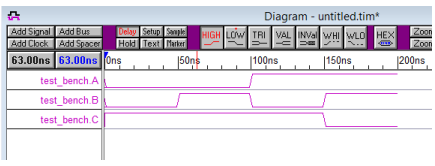


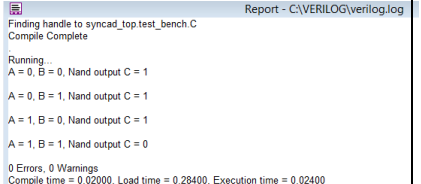
- Generare il diagramma temporale



(risultato nella slide successiva)

NAND: diagramma e report





Esercitazione 3
Testbench

Informazioni

- **Assistente: Marco Procaccini**
- **E-mail: marco.procaccini@unisi.it**
- **Ricevimento: Lunedì 16:30/19:00**
- **Online: https://unisi.webex.com/meet/marco.procaccini**
- **In persona: Laboratorio 223 – Secondo Piano San Niccolò**

Come possiamo sapere se un circuito funziona?

- **Abbiamo scritto il codice Verilog del nostro circuito…**

- **Ma … funziona correttamente? … il circuito fa ciò che deve?**

- **Dobbiamo testare il codice e abbiamo due possibilità:**

1. **Trial and Error:** carichiamo il codice direttamente su hardware (e.g., sintesi su FPGA) e testiamo che tutti i segnali siano corretti.

2. **Simulazione:** scriviamo un altro modulo in Verilog per testare il nostro circuito (il Testbench) all'interno di un simulatore (Verilogger).

Il modulo TestBench

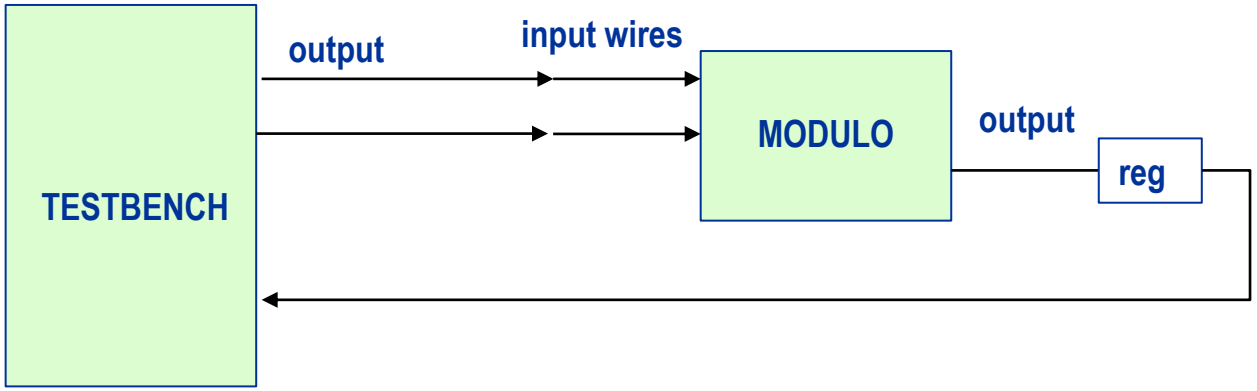
- Codice Verilog per testare un altro modulo Verilog, detto anche "Device Under Test" (DUT), oppure "Unit Under Test" (UUT).

- E' il modulo "Top-Level" nella simulazione
  - contiene tutti gli altri moduli DUT.
  - non è instanziato all'interno di altri moduli.

Testbench vs Moduli DUT

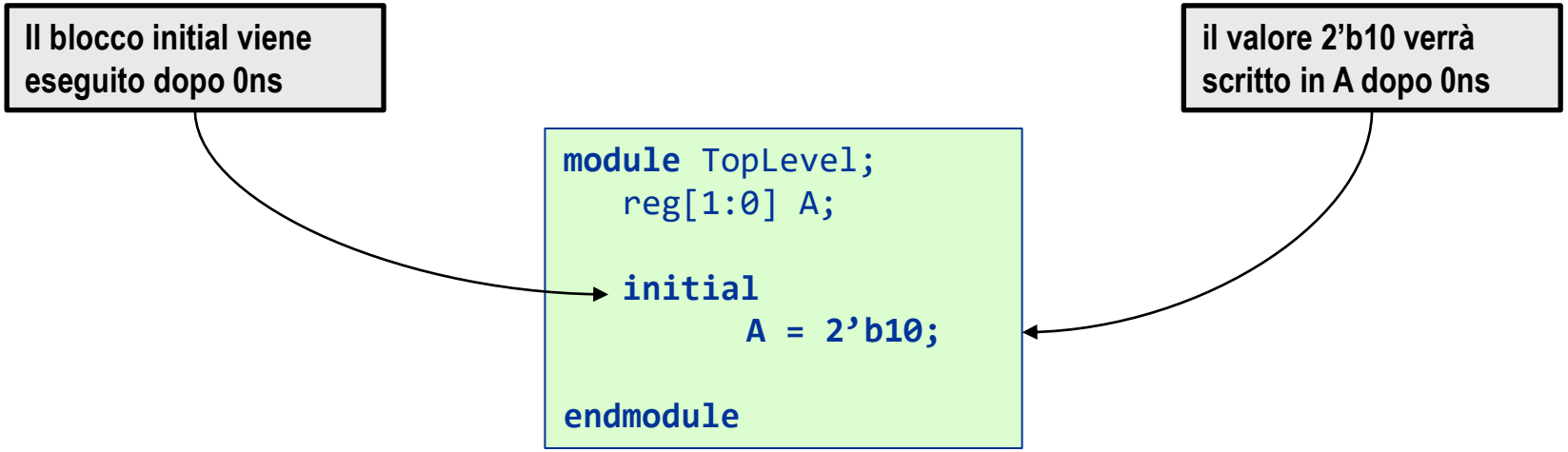
- I moduli DUT hanno come input dei wire e l'output può essere un wire oppure un registro.

- Il modulo testbench utilizza dei wire come output per inviare gli stimoli ai moduli DUT, mentre l'input è rappresentato dai dati in uscita del DUT.



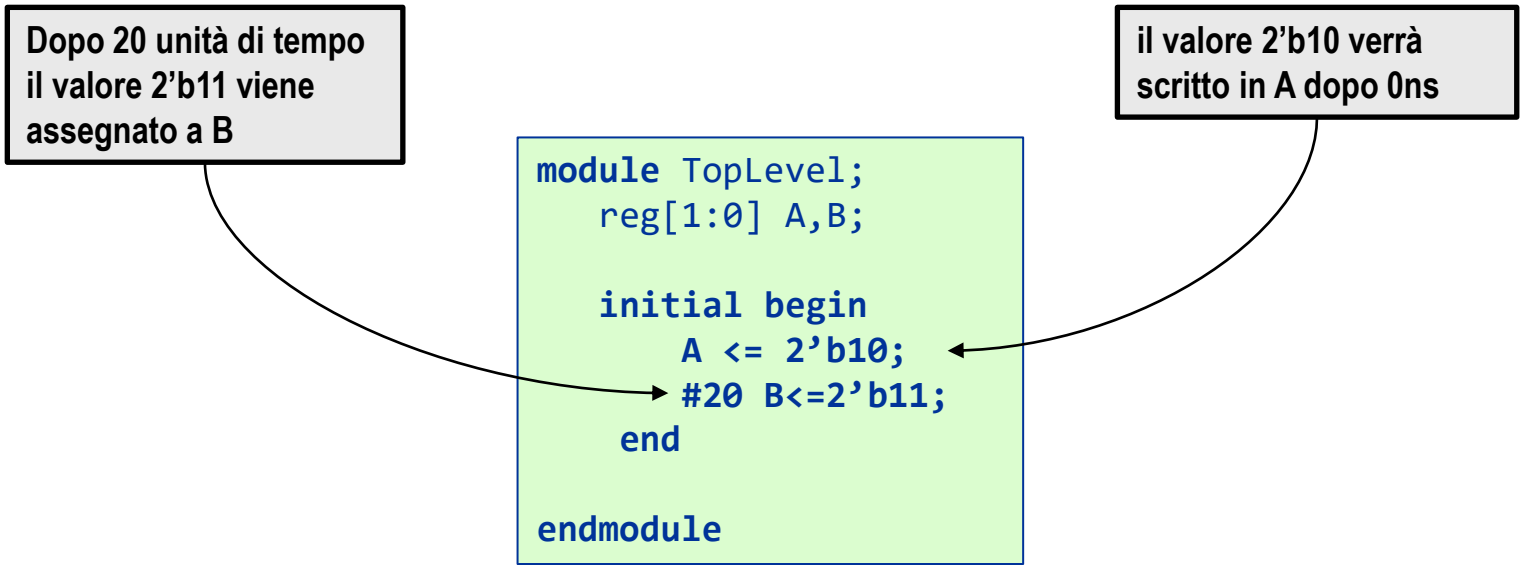
Blocco Initial

- Il blocco **initial** viene utilizzato per inizializzare registri e wire all'inizio della simulazione.
- Il blocco **initial** viene quindi eseguito una sola volta.



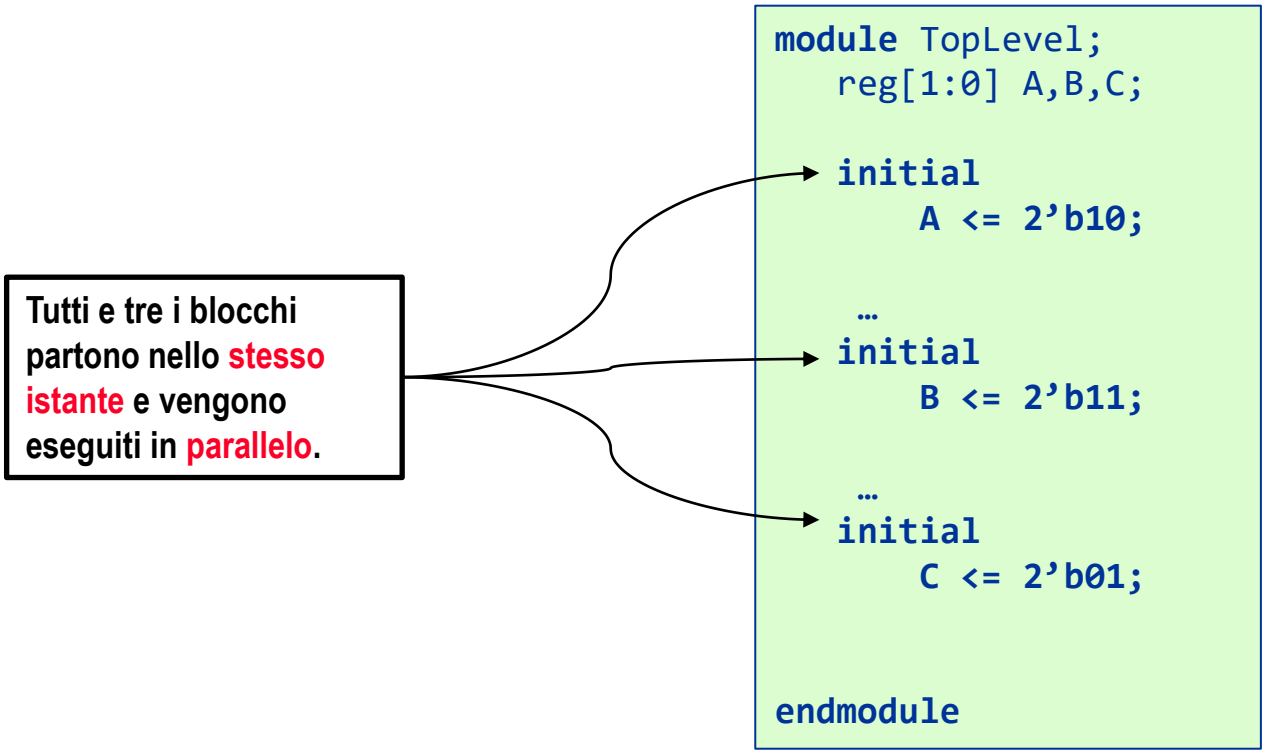
Esempio di blocco initial

- Si possono inserire dei ritardi



Blocchi Initial multipli

- Non c'è un limite al numero di blocchi initial.
- Tutti i blocchi initial partono nello stesso istante e vengono eseguiti in parallelo.



Esempio: la porta NAND

- Scriviamo il codice Verilog per implementare una porta NAND in hardware:

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

- Chiamiamo questo modulo: nand_module

```
module nand_module(c,a,b);
   outut c;
   intput a,b;
   wire d;          //filo interno per connettere and e not

   and a1(d,a,b);   //d è output, a e b sono inputs
   not n1(c,d);     // c è output, d è input
endmodule
```

Istanziare il modulo DUT nand

Modulo da testare

```
module nand_module (c,a,b);
   outut c;
   intput a,b;
   wire d;
   and a1(d,a,b);
   not n1(c,d);
endmodule
```

Sintassi

<nome_modulo> <nome_istanza> (<segnali_modulo>)

Istanza nel Testbench

```
module TopLevel;
    reg A,B;
    wire C;

    nand_module   N   (C,A,B);

endmodule
```

Testbench semplice

```
`timescale 1ns/1ps

module test_bench;
   reg A,B;
   wire C;

   nand_module   NM(C,A,B);


   //testiamo il modulo generando modificando i valori di A e B
   initial
      begin
         A<=1'b0; B<=1'b0;
         #50;
         A<=1'b0; B<=1'b1;
         #50;
         A<=1'b1; B<=1'b0;
         #50;
          A<=1'b1; B<=1'b1;
         #50;
      end
endmodule
```

definisco l'unità di tempo della simulazione

definiscono l'ordine dei segnali nel time diagram

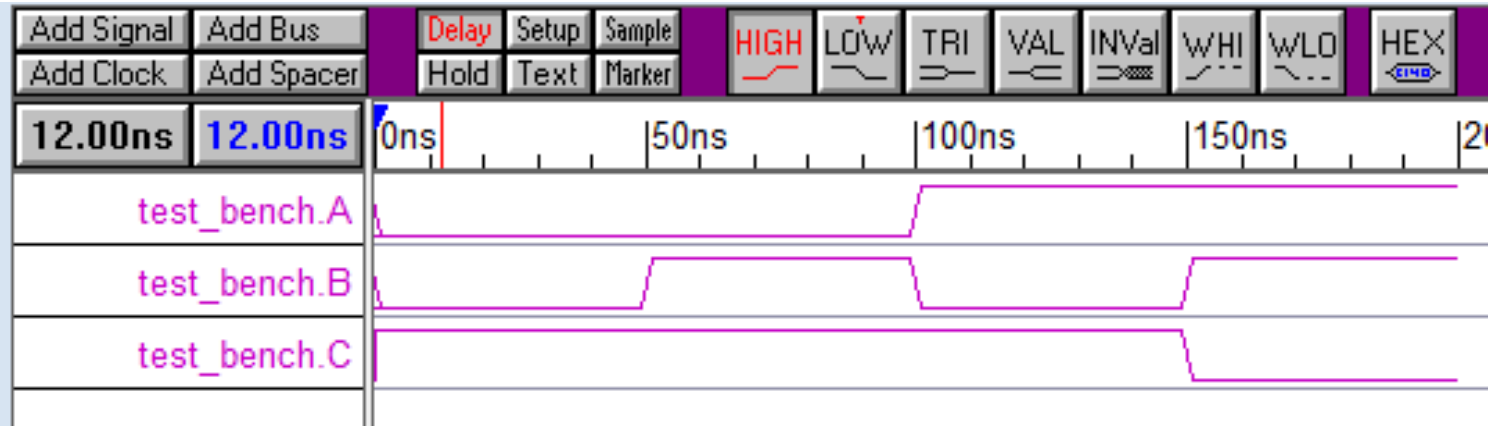
nome del modulo da testare

istanza del modulo da testare

Assegno un diverso valore
ad A e B ogni 50 nanosecondi

Testbench semplice

```
   //testiamo il modulo generando modificando i valori di A e B
   initial
      begin
         A<=1'b0; B<=1'b0;
         #50;
         A<=1'b0; B<=1'b1;
         #50;
         A<=1'b1; B<=1'b0;
         #50;
          A<=1'b1; B<=1'b1;
         #50;
      end
endmodule
```



Testbench Processi

- **$display ("<formato>", espressione1, espressione2…);**

  Stampa immediata a video del valore delle espressioni

- **$monitor ("<formato>", espressione1, espressione2...);**

  Stampa a video alla fine del time step corrente se il valore dell'espressione e' cambiato.

- **$strobe ("<formato>", espressione1, espressione2...);**

  Stampa a video del valore delle espressioni alla fine del time step:

Alcuni possibili formati	
%b	Binario
%c	Char
%d	Decimale
%s	String
%h	Esadecimale
%t	Tempo di simulazione

Testbench Processi - 2

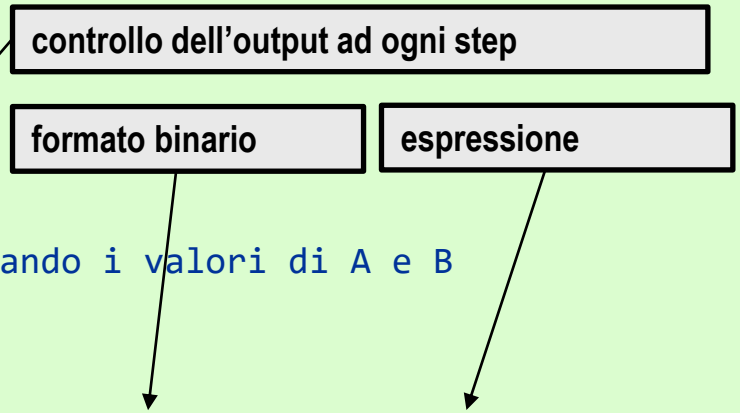
- **$stop**
  - Sospende la simulazione ed imposta il simulatore in modalità interattiva.
  - Utile per osservare la simulazione nel momento dello stop.

- **$finish**
  - Esce dalla simulazione e restituisce il controllo al sistema operativo.
  - Obbligatorio inserirlo per terminare la simulazione.

- **$readmemh("data.txt", buf)**
  - Legge il contenuto di un file in formato esadecimale (caratteri ASCII 0-9, A-F) e lo carica nel registro buf (con la corrispondente codifica binaria).
  - buf deve essere opportunamente dichiarato (ad esempio, reg[31:0] buf[0:11])

Testbench Monitoraggio delle Variabili

```
`timescale 1ns/1ps

module test_bench;
   reg A,B;
   wire C;

   nand_module   NM(C,A,B);

   //testiamo il modulo generando modificando i valori di A e B
   initial
      begin
         A<=1'b0; B<=1'b0;
         #50; if (C != 1)  $display ("ERROR! A=%b B=%b C=%0b", A,B,C);
         A<=1'b0; B<=1'b1;
         #50; if (C != 1)  $display ("ERROR! A=%b B=%b C=%0b", A,B,C);
         A<=1'b1; B=1'b0;
         #50; if (C != 1)  $display ("ERROR! A=%b B=%b C=%0b", A,B,C);
          A<=1'b1; B<=1'b1;
         #50;
          if (C != 0)  $display ("ERROR! A=%b B=%b C=%0b", A,B,C);
               else    $display ("OK C=%d",C);
      end
endmodule
```



Esempi Testbench: DET con SRMS

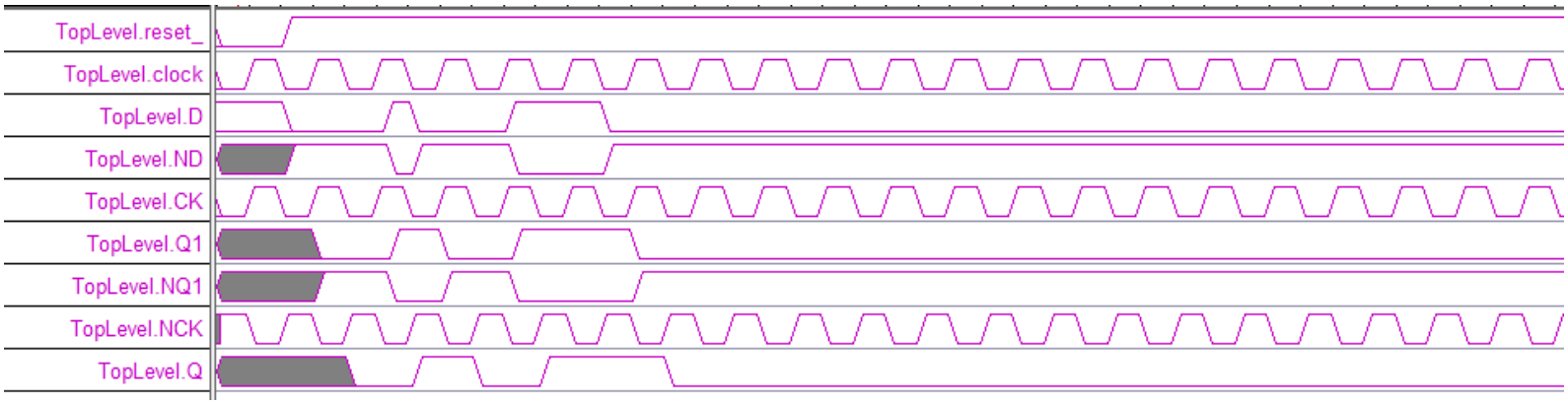
```
`timescale 1ns/1ps

module TopLevel;
   reg reset_;
   initial begin reset_=0; #22 reset_=1; #500; $stop; end

   reg clock; initial clock=0; always #10 clock<=(!clock);
   reg D;
   wire ND=dsrms.r;
   wire CK=dsrms.mysrms.ck;
   wire Q1 = dsrms.mysrms.q1;
   wire NQ1 = dsrms.mysrms.qbar1;
   wire NCK = dsrms.mysrms.nck;
   wire Q;

   initial begin
      wait(reset_==1); D<=0; #32 D<=1; #8; D<=0; #30 D<=1; #30; D<=0; #1000
      $finish;
   end

   DET_SRMS dsrms(D,clock,Q);
endmodule
```

Monitoraggio delle variabili dei sottomoduli nel time diagram

Esempi Testbench: DET con SRMS

```
wire ND=dsrms.r;
   wire CK=dsrms.mysrms.ck;
   wire Q1 = dsrms.mysrms.q1;
   wire NQ1 = dsrms.mysrms.qbar1;
   wire NCK = dsrms.mysrms.nck;
   wire Q;
```

Monitoraggio delle variabili dei sottomoduli nel time diagram



Esempi Testbench: Riconoscitore sequenza

```
`timescale 1ns/1ps
module Toplevel;

 reg reset_; initial begin reset_=0; #22 reset_=1; #300; $stop; end
 reg clock; initial clock=0; always #5 clock<=(!clock);
 reg [1:0] X;
 wire z=Xxx.z;
 wire [1:0] STAR=Xxx.STAR;
 initial begin X=0;
 wait(reset_==1);

 @(posedge clock); X<=0; @(posedge clock); X<=3; @(posedge clock); X<=1;
 @(posedge clock); X<=2; @(posedge clock); X<=3; @(posedge clock); X<=1;
 @(posedge clock); X<=3; @(posedge clock); X<=1; @(posedge clock); X<=2;
 @(posedge clock); X<=1; @(posedge clock); X<=3; @(posedge clock); X<=1;
 @(posedge clock); X<=1; @(posedge clock); X<=2; @(posedge clock); X<=3;
 @(posedge clock); X<=3; @(posedge clock); X<=3; @(posedge clock); X<=1;
 @(posedge clock); X<=2; @(posedge clock); X<=2; @(posedge clock); X<=2;
 @(posedge clock); X<=3; @(posedge clock); X<=2; @(posedge clock); X<=2;
 $finish;
 end

 ricseq110110moore Xxx(z,X,clock,reset_); //ad esempio, Moore

endmodule
```

ATTENZIONE: non è un always. L'assegnamento successivo diventa bloccante.

ESERCIZI in LABORATORIO – VERIFICA DEI MODULI PRINCIPALI DEL RISC-V (ESERCITAZIONE/LAB #04)

OBIETTIVI

1) **Capire la descrizione formale in Verilog di un semplice processore RISC-V**
2) **Capire come costruire il testbench per un modulo**

---

1) Capire la descrizione formale in Verilog di un semplice processore RISC-V

v. slide collegate https://arcal.dii.unisi.it/lezioni/lezioni124/c124es04-verilog_riscv.pdf

2) Capire come costruire il testbench per un modulo

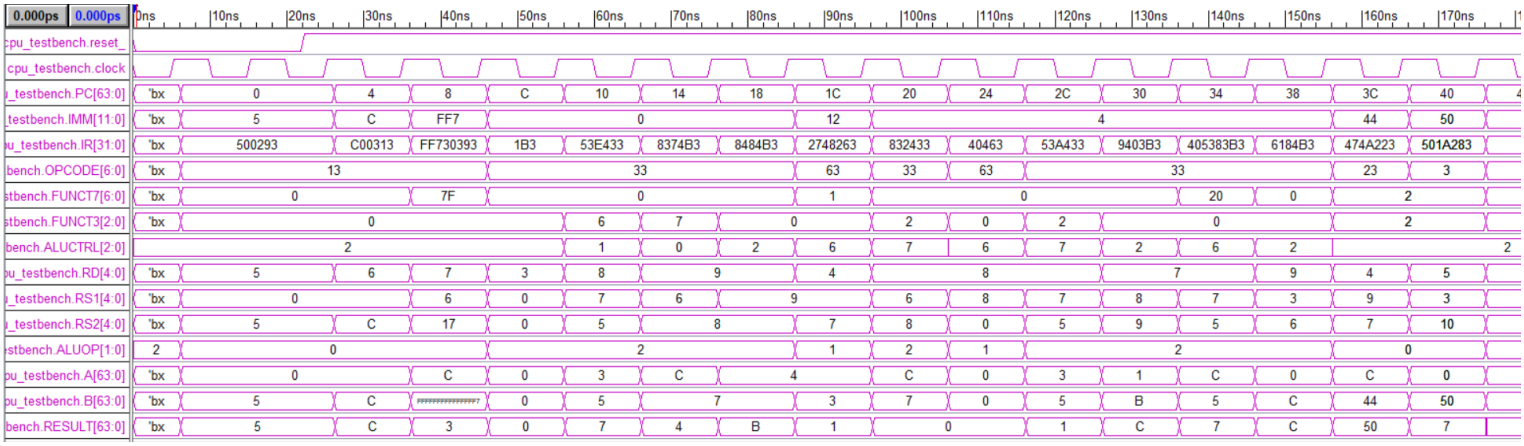
a) Scaricare i file:

https://arcal.dii.unisi.it/tools1/rv64_test1.v (descrizione in Verilog di un semplice RISC-V)

https://arcal.dii.unisi.it/tools1/memfilerv.dat (codice macchina di un semplice programma)

b) Generare un nuovo progetto Verilogger (v. esercitazione #01) e aggiungere il file Verilog rv64_test1.v

c) Provare a compilare ed eseguire la simulazione di questo codice: deve essere riprodotto il diagramma temporale https://arcal.dii.unisi.it/tools1/rv64_test1.png



d) Ad ogni gruppo verrà assegnato uno o più moduli

e) Il gruppo dovrà creare un nuovo progetto relativo solo a quel modulo e generare i segnali di ingresso (nel testbench) e verificare che i segnali di uscita corrispondano a quelli del diagramma del punto c per quel modulo. La struttura del testbench sarà del tipo

```
module testbench;
   reg reset_; initial begin reset_=0; #22 reset_=1; #400; end
   reg clock;  initial clock=0; always #5 clock<=(!clock);
   initial begin wait(reset_); #180 $finish; end
   reg …
   wire …
   initial begin
    …
   $finish
   end
   XXX  xxx (clock,reset, <segnali_ingresso>,   <segnali_uscita>)
endmodule
```

Esercitazione 4: Processore RISC-V in Verilog

- OBIETTIVO: DESCRIVERE IL PROCESSORE RISC-V IN VERILOG e IMPARARE A SCRIVERE IL TESTBENCH PER UN MODULO

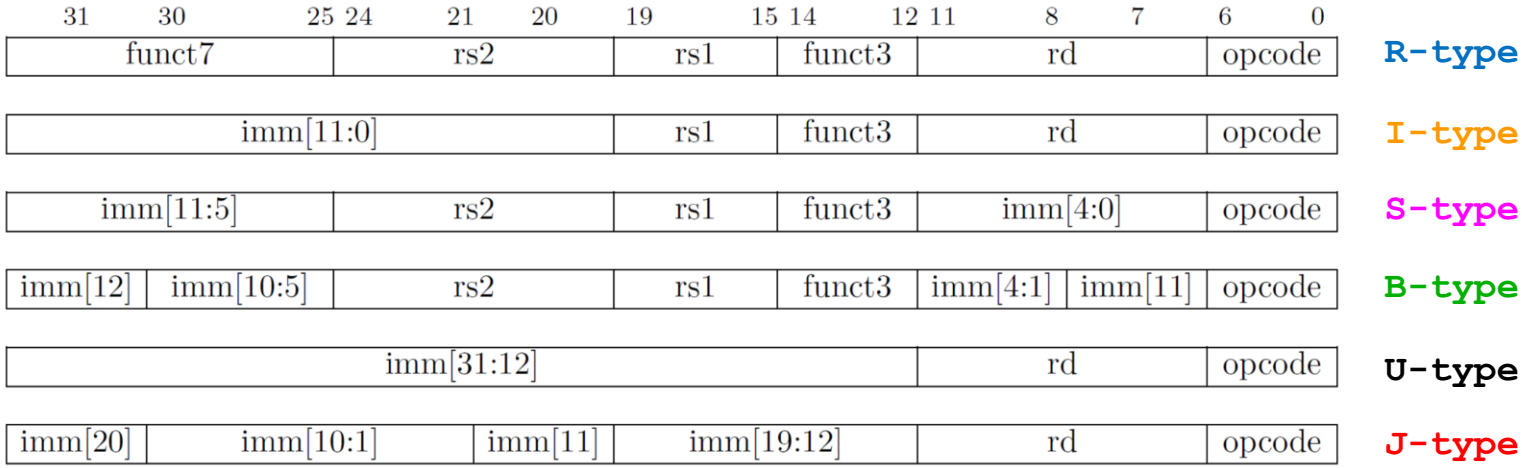
  - Verificare la struttura dei vari moduli che lo compongono in modo da descrivere i corrispondenti moduli del processore RISC-V a 64 bit

  - Ad OGNI GRUPPO è assegnato un modulo:
    il gruppo dovrà scrivere un testbench per fare il testing del modulo assegnato

  - Verificare nel Verilogger
    la correttezza del diagramma temporale di ciascun modulo

  - Possibilmente: mettiamo insieme tutti i moduli per avere una descrizione completa del processore RISC-V

Ricapitolazione dei formati e istruzioni principali



- **Istruzione** **Significato** **Variante** **Significato**

```
add x5,x6,x7    x5 = x6 + x7          addi x5,x6,10    x5 = x6 + 10
sub x5,x6,x7    x5 = x6 - x7
slt x5,x6,x7    x5 = (x6<x7) ? 1 : 0  slti x5,x6,10    x5 = (x6<10) ? 1 : 0

ld  x5,100(x9)  x5 = Memory[x9+100] (64 bit)   lw  x5,100(x9)  (32 bit)
sd  x5,100(x9)  Memory[x9+100] = x5 (64 bit)   sw  x5,100(x9)  (32 bit)

bne x6,x7,L     salta a L se x6!=x7 (L si trova a offset imm13=(&L-PC))
beq x6,x7,L     salta a L se x6==x7 (L si trova a offset imm13=(&L-PC))

lui x5,imm20    carica imm20 nei 20 bit alti

jal FUN         esegue la funzione FUN (che si trova a offset imm21=(&FUN-PC))
ret             ritorna da funzione (jalr x0,0(x1))
```

Programma riscvtest.s → derivare gli opcode = bit 6:0
func7,func3 =bit 31:25,14:12, o immediate,func3=bit 31:20,14:12, o…

	ISTRUZIONE ASSEMBLY		PC	ISTR.MACCHINA	OPCODE	R-type:func7,func3
main:	addi x5, x0, 5	# initialize x5 = 5	0	00500293	→ 13	→ 005,0
	addi x6, x0, 12	# initialize x6 = 12	4	00c00313	→ 13	→ 00c,0
	addi x7, x6, -9	# initialize x7 = 3	8	ff730393	→ 13	→ ff7,0
	add x3, x0, x0	# initialize x3 = 0	c	000001b3	→ 33	→ 00,0
	or x8, x7, x5	# x8 = (3 OR 5) = 7	10	0053e433	→ 33	→ 00,6
	and x9, x6, x8	# x9 = (12 AND 7) = 4	14	008374b3	→ 33	→ 00,7
	add x9, x9, x8	# x9 = 4 + 7 = 11	18	008484b3	→ 33	→ 00,0
	beq x9, x7, end	# shouldn't be taken	1c	02748263	→ 63	→ 012,0
	slt x8, x6, x8	# x8 = 12 < 7 = 0	20	00832433	→ 33	→ 00,2
	beq x8, x0, around	# should be taken	24	00040463	→ 63	→ 004,0
	addi x9, x0, 0	# shouldn't happen	28	00000493	→ 13	→ 000,0
around:	slt x8, x7, x5	# x8 = 3 < 5 = 1	2c	0053a433	→ 33	→ 00,2
	add x7, x8, x9	# x7 = 1 + 11 = 12	30	009403b3	→ 33	→ 00,0
	sub x7, x7, x5	# x7 = 12 - 5 = 7	34	400538b3	→ 33	→ 20,0
	add x9, x3, x6	# x9 = 10008000 + 12	38	006184b3	→ 33	→ 00,0
	sw x7, 68(x9)	# [10008050] = 7	3c	0474a223	→ 23	→ 047,2
end:	lw x5, 80(x3)	# x5 = [10008050] = 7	40	0501a283	→ 03	→ 050,2

I-type:imm, func3

B-type:bit31:25,11:5→imm[12],imm[10:5],imm[4:1],imm[11], func3

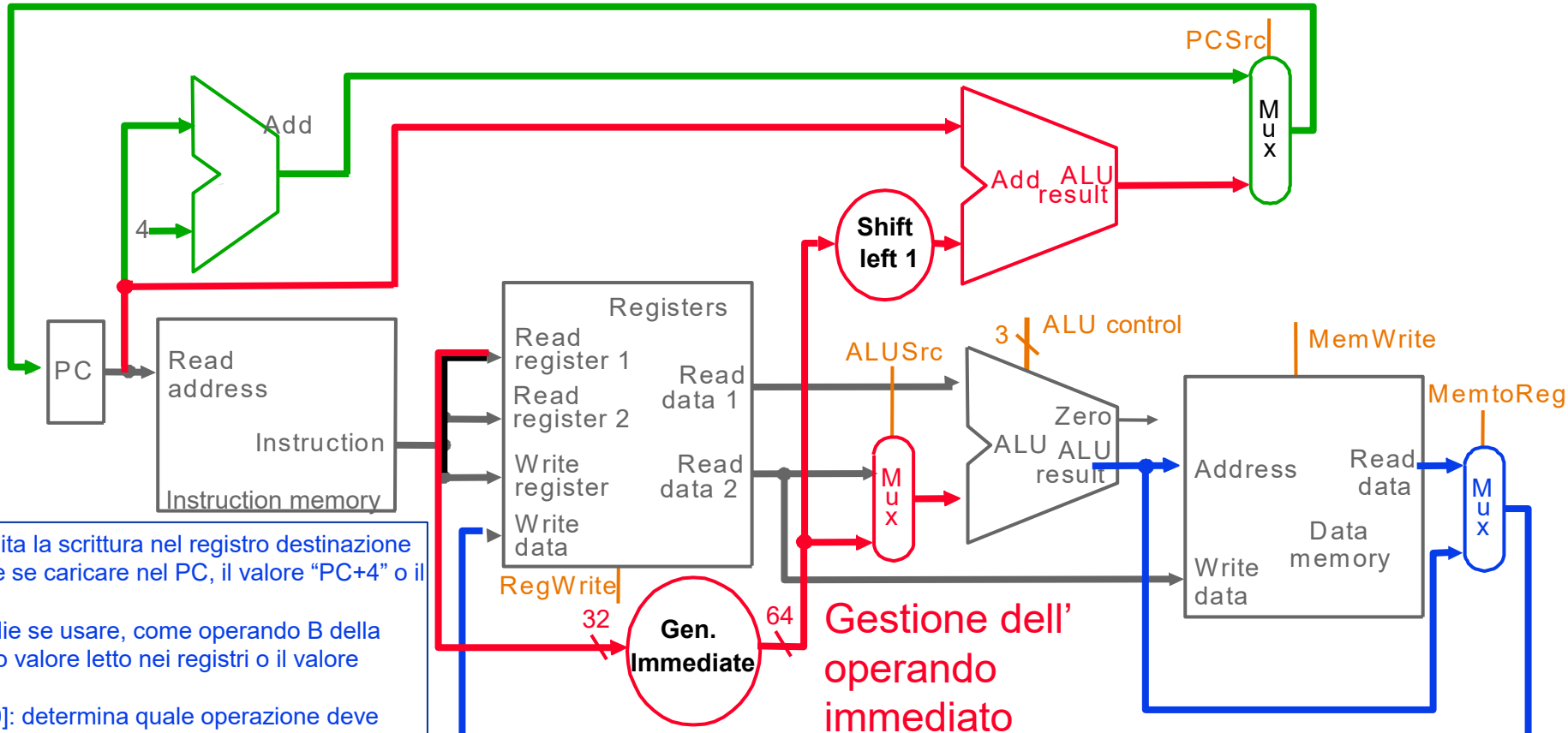
S-type:bit31:25,11:5→imm[11:5],imm[4:0], func3

Costruzione del Datapath del processore RISC-V

- c.f. PHRV1 figura 4.11

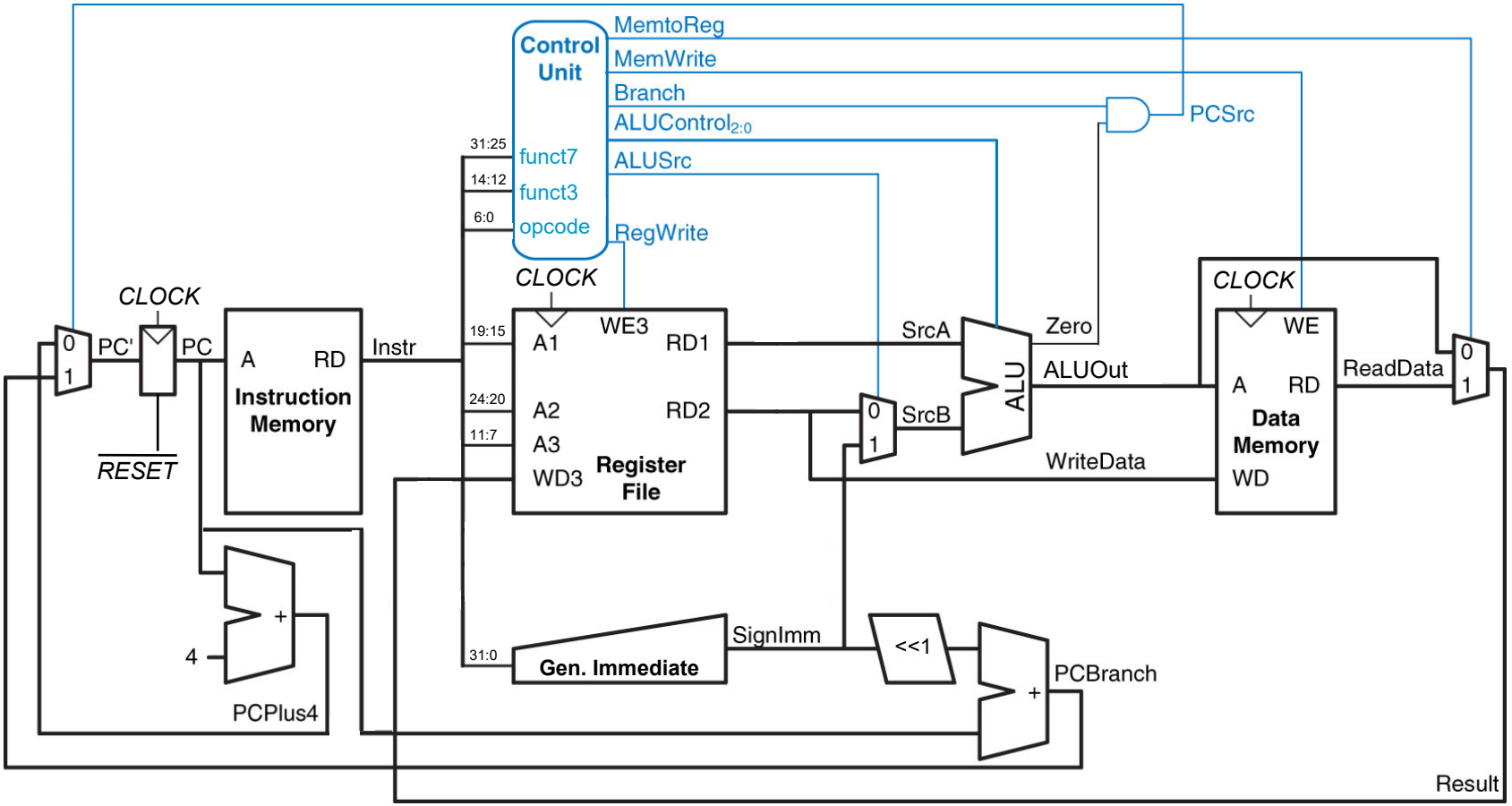
Gestione indirizzo istruzione successiva

I segnali di controllo vengono pilotati in base a “op-code” e “funct” dell’istruzione:



- **RegWrite**: abilita la scrittura nel registro destinazione
- **PCSrc**: sceglie se caricare nel PC, il valore “PC+4” o il target del salto
- **ALUSrc**: sceglie se usare, come operando B della ALU, il secondo valore letto nei registri o il valore immediato
- **AluControl**[2:0]: determina quale operazione deve essere eseguita dalla ALU
- **MemWrite**: abilita la scrittura nella memoria
- **MemtoReg**: sceglie se scrivere nel registro destinazione il valore letto dalla memoria o il risultato prodotto dalla ALU

Datapath del RISC-V (più preciso)



RegWrite
PCSrc
ALUSrc
AluControl[2:0]
MemWrite
MemtoReg

MemRead — Non c'e' rispetto al Patterson

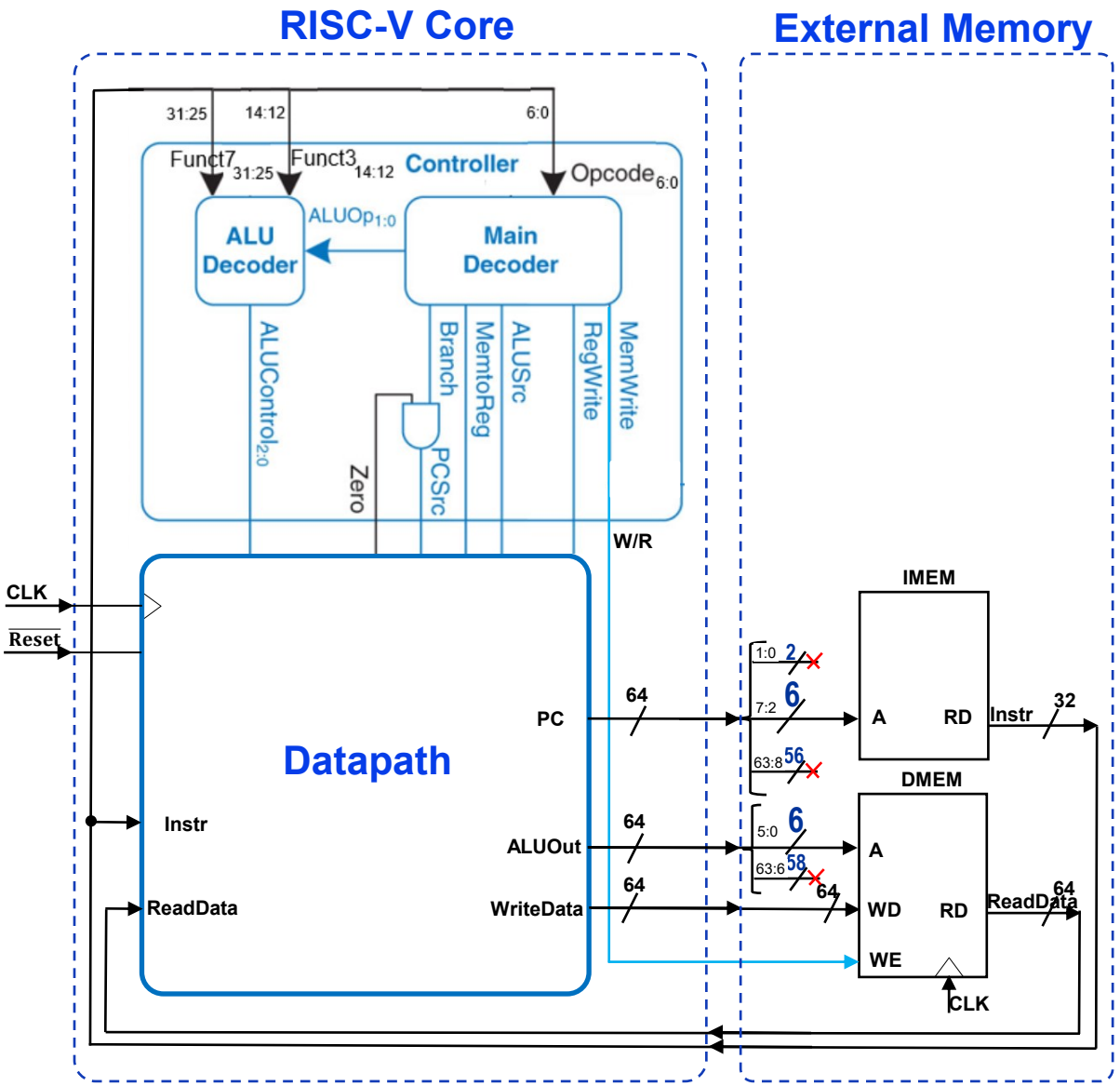
Branch — Nuovo

ALUcontrol — 3 fili anziche' 4

TESTBENCH:

```
`timescale 1ns/1ps
module cpu_testbench;
   reg reset_; initial begin reset_=0; #22 reset_=1; #400; end
   reg clock;  initial clock=0; always #5 clock<=(!clock);
   initial begin wait(reset_); #180 $finish; end
   riscvmem cpu(clock,reset_);
endmodule
```

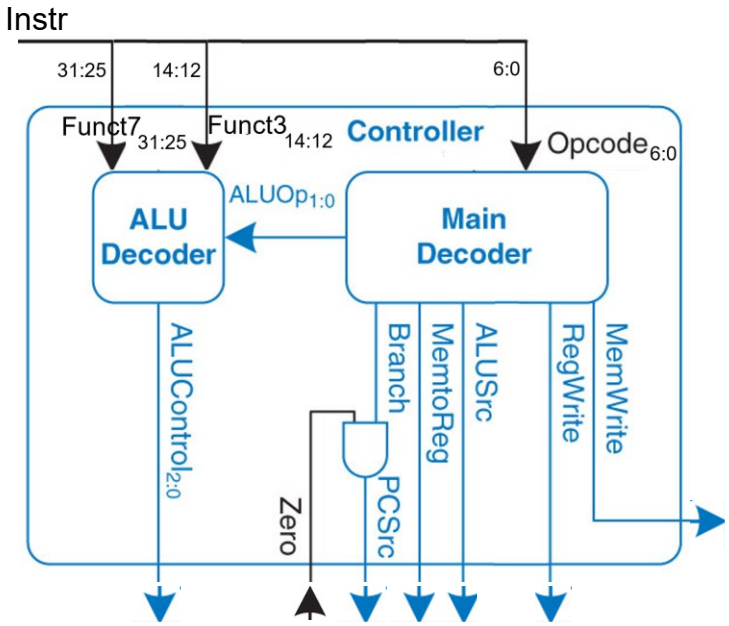
RISC-V processor




```
module riscvmem(clk,reset_);
   input clk,reset_; wire memwrite;
   wire [63:0] readdata,pc,aluout,writedata;
   wire [31:0] instr;
   // instantiate processor and memories
   rvcore mycore(clk,reset_,readdata,instr,
                 pc,aluout,writedata,memwrite);
   imem imem(pc[7:2],  instr);
   dmem dmem(clk,aluout[5:0],writedata,memwrite,
             readdata);
endmodule
```

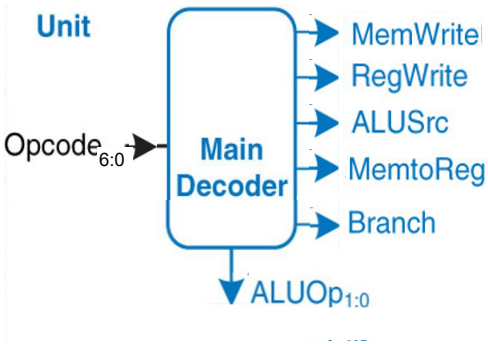
```
module rvcore(clk,reset_,readdata,instr,
              pc,aluout,writedata,memwrite);
   input        clk, reset_;
   input[63:0]  readdata;
   input[31:0]  instr;
   output[63:0] pc,aluout,writedata;
   output       memwrite;
   wire         regwrite,alusrc,memtoreg,pcsrc,
                zero;
   wire[2:0]    alucontrol;

   controller c(instr[31:25],instr[14:12],
        instr[6:0],zero,  memwrite,regwrite,
        alusrc,memtoreg,pcsrc,alucontrol);

   datapath dp(clk,reset_,instr,readdata,
        regwrite,alusrc,memtoreg,pcsrc,
        alucontrol,  zero,pc,aluout,writedata);
endmodule
```

RISC-V Controller

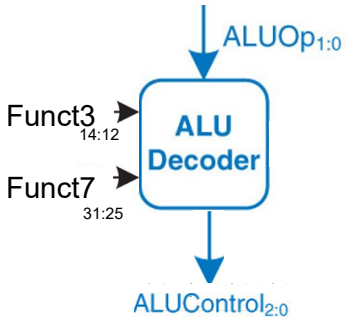



```
module controller(funct7,funct3,opcode,zero,
                  memwrite,regwrite,alusrc,memtoreg,
                  pcsrc,alucontrol);
   input[2:0] funct3;
   input[6:0] opcode,funct7;
   input zero;
   output memwrite,regwrite,alusrc,memtoreg,pcsrc;
   output [2:0] alucontrol;
   wire [1:0] aluop;
   wire branch;

   maindec md(opcode,  memwrite,regwrite,alusrc,
              memtoreg,branch,aluop);
   aludec ad(funct7,funct3,aluop,  alucontrol);
   assign pcsrc = branch & zero;
endmodule
```

RISC-V Main Decoder e RISC-V ALU Decoder



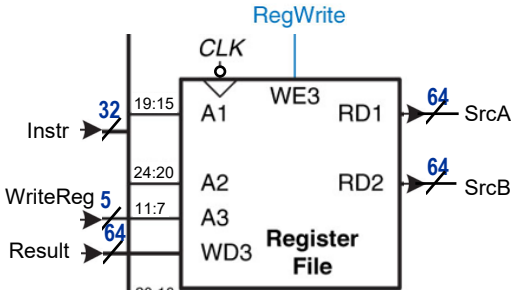

```
module maindec(opcode,  memwrite,regwrite,alusrc,memtoreg,branch,aluop);
   input[6:0] opcode;
   output memwrite,regwrite,alusrc,memtoreg,branch; output[1:0] aluop;reg[7:0]controls;
   assign {regwrite,alusrc,branch,memwrite,memtoreg,aluop} = controls;
   always @(opcode) casex(opcode)          //       (opcode->controls)
      7'b0110011: controls <= 7'b1000010; // R-TYPE  (0x33->0x42)
      7'b0000011: controls <= 7'b1100100; // LW      (0x03->0x64)
      7'b0100011: controls <= 7'b0101000; // SW      (0x23->0x28)
      7'b1100011: controls <= 7'b0010001; // BEQ     (0x63->0x11)
      7'b0010011: controls <= 7'b1100000; // ADDI    (0x13->0x60)
      default:    controls <= 7'bxxxxxxx; // illegal op
   endcase
endmodule
```



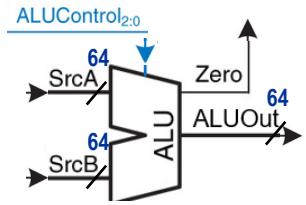
ALUcontrol	Function
0 00 =0	AND
0 01 =1	OR
0 10 =2	ADD
1 10 =6	SUB
1 11 =7	SLT
1 10 =6	EQ

```
module aludec(funct7,funct3,aluop,   alucontrol);
   input[6:0] funct7; input[2:0] funct3; input[1:0] aluop;
   output [2:0] alucontrol; reg[2:0] alucontrol;
   always @(aluop or funct3 or funct7) casex(aluop)        //      (aluop->alucontrol)
      2'b00: alucontrol <= 3'b010;                 //lw/sw/addi          0->2 alu=add
      2'b01: alucontrol <= 3'b110;                 //beq                 1->6 alu=eq
      2'b10: casex(funct3)                         //R-TYPE instructions 2->R-type
            3'b000: casex(funct7)                    //add or sub  (funct3->alucontrol)
               7'b0000000: alucontrol <= 3'b010;   //add  0(funct7=0x00)->2 alu=add
               7'b0100000: alucontrol <= 3'b110;   //sub  0(funct7=0x20)->6 alu=sub
               endcase
            3'b111:  alucontrol <= 3'b000;         //and                 7->0 alu=and
            3'b110:  alucontrol <= 3'b001;         //or                  6->1 alu=or
            3'b010:  alucontrol <= 3'b111;         //slt                 2->7 alu=slt
            default: alucontrol <= 3'bxxx;         //???                 Unknown funct3
         endcase
      default: alucontrol<=3'bxxx;                 //???                 Unknown aluop
   endcase
endmodule
```

RISC-V Registerfile, Adder, ALU

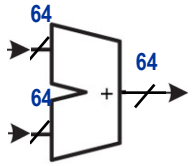



```
module regfile(clk,we3,a1,a2,a3,wd3,   rd1,rd2);
   input clk,we3; input[4:0] a1,a2,a3; input[63:0] wd3;
   output[63:0] rd1,rd2; reg[63:0] rd1,rd2;
   reg[63:0] rf[0:31];
   // three ported register file: read two ports combinationally;
   // write third port on falling edge of clk; register 0 hardwired to 0
   always @(negedge clk) if (we3) rf[a3] <= wd3;
   always @(a1) rd1 <= (a1 != 0) ? rf[a1] : 0;
   always @(a2) rd2 <= (a2 != 0) ? rf[a2] : 0;
endmodule
```



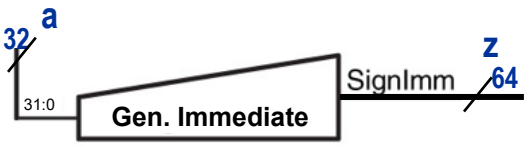
ALUcontrol	Function
0 00	AND
0 01	OR
0 10	ADD
1 10	SUB
1 11	SLT
1 10	EQ

```
module alu(a,b,aluctrl,    aluout,zero);
   input[63:0] a,b; input[2:0] aluctrl;
   output[63:0] aluout; reg[63:0] aluout; output zero;
   assign zero = (aluout==0);
   always @(aluctrl or a or b) #0.1 //delay to avoid infinite speed loops
      casex (aluctrl)
         0: aluout <= a & b;
         1: aluout <= a | b;
         2: aluout <= a + b;
         6: aluout <= a - b;
         7: aluout <= a<b ? 1:0;
         default: aluout<=0; // should not happen!
      endcase
endmodule
```

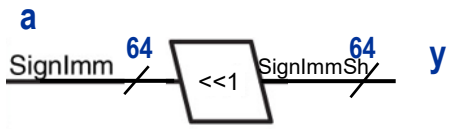



```
module adder(a,b,  aluout);
   input[63:0] a,b;
   output[63:0] aluout;
   assign aluout=a+b;
endmodule
```

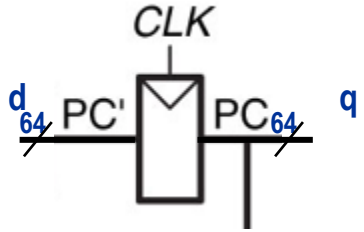
Shift-Left-1, Gen-Immediate, Resettable Flip-flop, MUX2



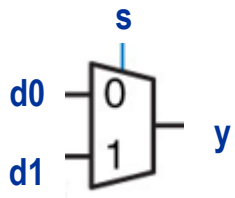

```
module genimm(a,   z);
   input[31:0] a; output[63:0] z; reg[11:0] y; wire[6:0] opc;
   assign opc=a[6:0];
   always @(a or opc) casex(opc)
      7'b00x0011: y <= a[31:20];
      7'b0100011: y <={a[31:25],a[11:7]};
      7'b1100011: y <={a[31],a[7],a[30:25],a[11:8]};
   endcase
   assign z = {{52{y[11]}}, y};
endmodule
```




```
module sl1(a, y); // shift left by 1
  input[63:0] a; output[63:0] y;
  assign y = {a[62:0], 1'b0};
endmodule
```

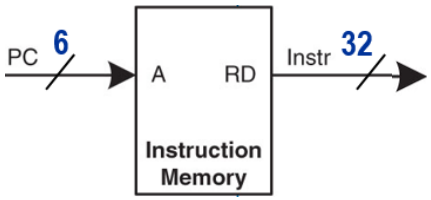



```
module flopr (clk,reset_,d,   q);
   parameter WIDTH = 8;
   input  [WIDTH-1:0] d; input clk, reset_;
   output [WIDTH-1:0] q; reg [WIDTH-1:0] q;
   always @(posedge clk) // reset on posedge
      #1.1 if (!reset_) q <= 0; //delay to avoid infinite speed loops
      else q <= d;
endmodule
```

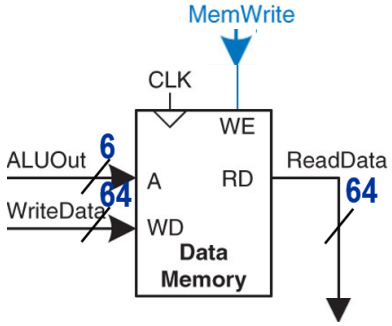



```
module mux2 (d0,d1,s,   y);
   parameter WIDTH = 64;
   input[WIDTH-1:0] d0, d1; input s; output[WIDTH-1:0] y;
   assign y = s ? d1 : d0;
endmodule
```

Instruction Memory and Data Memory

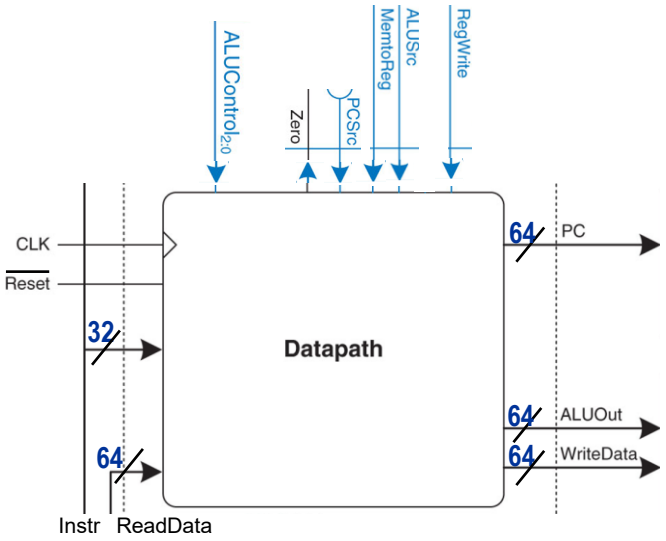


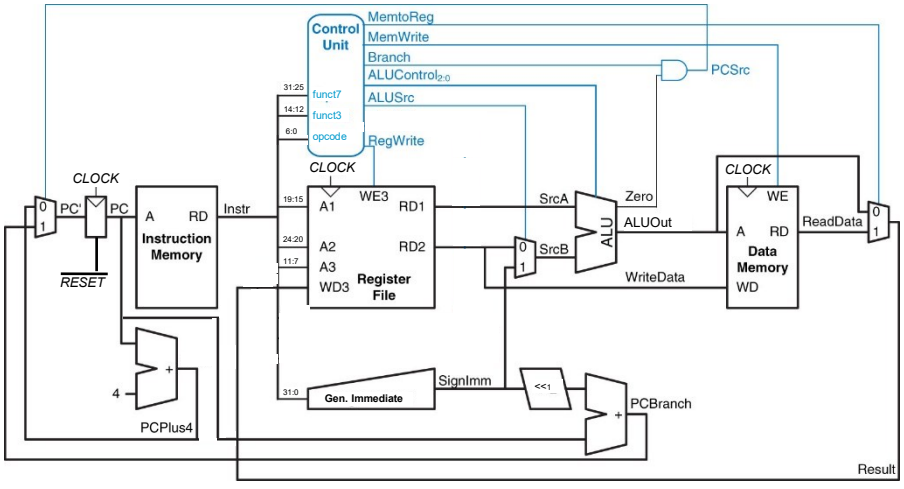

```
module imem(a,  rd);
   input[5:0] a; output[31:0] rd;
   reg[31:0] ROM[0:63];
   initial $readmemh("memfilerv.dat", ROM);
   assign rd = ROM[a]; // word aligned
endmodule
```




```
module dmem(clk,a,wd,we,  rd);
   input clk, we;  input[5:0] a; input[63:0] wd; output[63:0] rd;
   reg[63:0] RAM[0:7];  // 64 bytes (i.e., 8 dwords)
   assign rd = RAM[a[5:3]]; // dword aligned
   always @(posedge clk)
   if (we) RAM[a[5:3]] <= wd;
endmodule
```

RISC-V Datapath






```
module datapath(clk,reset_,instr,readdata,regwrite,
                alusrc,memtoreg,pcsrc,alucontrol,
                zero,pc,aluout,writedata);
   input         clk,reset_;
   input[31:0]   instr;
   input[63:0]   readdata;
   input         regwrite,alusrc,pcsrc,memtoreg;
   input[2:0]    alucontrol;
   output        zero;
   output[63:0]  pc,aluout,writedata;
   wire[63:0]    pcnext,pcplus4,pcbranch,signimmsh;
   wire[63:0]    signimm,srca,srcb,result;

   // next PC logic
   flopr #(64)   pcreg(clk,reset_,pcnext,       pc);
   adder         pcadd1(pc,64'b100,             pcplus4);
   sl1           immsh(signimm[63:0],           signimmsh);
   adder         pcadd2(pc,signimmsh,           pcbranch);
   mux2 #(64)    brmux(pcplus4,pcbranch,pcsrc,  pcnext);

   // register file logic
   regfile       rf(clk,regwrite,instr[19:15],instr[24:20],
                    instr[11:7],result,   srca,writedata);
   mux2 #(64)    resmux(aluout,readdata,memtoreg,  result);
   genimm        gimm(instr,   signimm);

   // ALU logic
   mux2 #(64)    srcbmux(writedata,signimm,alusrc,  srcb);
   alu           alu(srca,srcb,alucontrol,   aluout,zero);
endmodule
```

ESERCIZIO (a gruppi – ogni fila un gruppo)

- Costruire il TESTBENCH **SEPARATO** per i seguenti moduli:
- FILA1: CONTROLLER
- FILA2: MAIN-DEC
- FILA3: ALU-DEC
- FILA4: REGFILE
- FILA5: ALU
- FILA6: GEN-IMM + SL1
- FILA7: FLOPR + MUX2
- FILA8: IMEM + DMEM

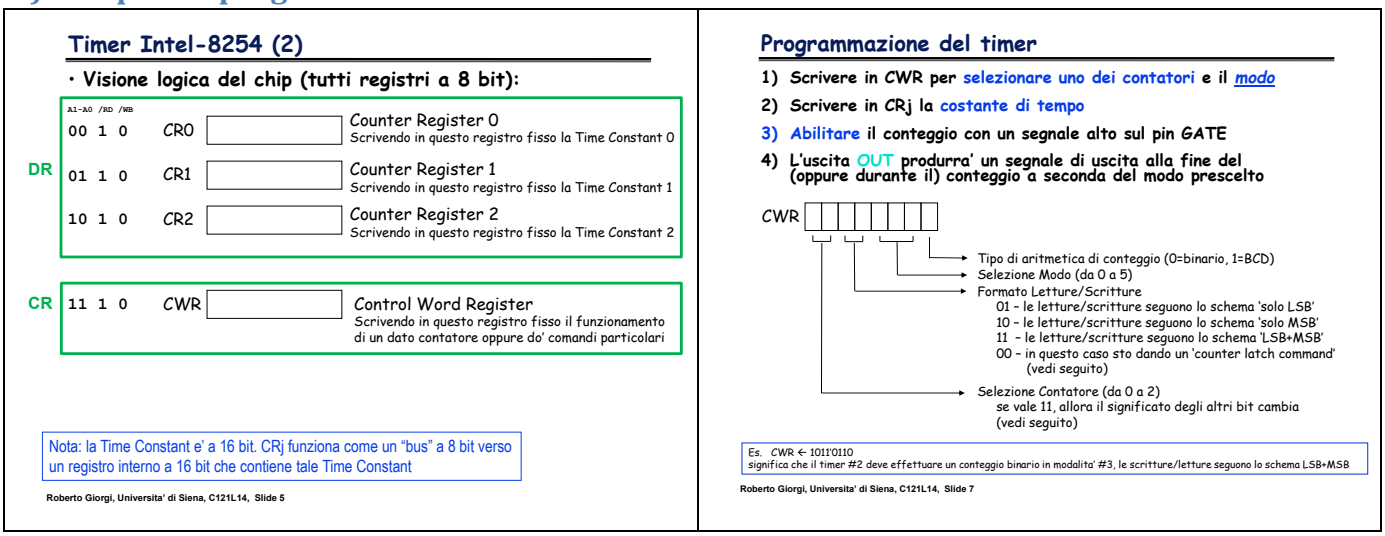
- TESTBENCH (versione estesa per verificare più segnali):

```
module cpu_testbench;
   reg reset_; initial begin reset_=0; #22 reset_=1; #400; end
   reg clock;  initial clock=0; always #5 clock<=(!clock);
   initial begin wait(reset_); #180 $finish; end
   wire[63:0] PC; wire[11:0] IMM; wire[31:0] IR; wire[6:0]OPCODE, FUNCT7;
   wire[2:0] FUNCT3,ALUCTRL; wire[4:0] RD, RS1, RS2; wire[1:0] ALUOP;
   assign PC=cpu.mycore.dp.pc,IR=cpu.mycore.dp.instr, ALUCTRL=cpu.mycore.dp.alu.aluctrl;
   assign ALUOP=cpu.mycore.c.ad.aluop;
   assign OPCODE=cpu.mycore.c.opcode,FUNCT7=cpu.mycore.c.funct7;
   assign FUNCT3=cpu.mycore.c.funct3,RD=cpu.mycore.dp.rf.a3;
   assign RS1=cpu.mycore.dp.rf.a1,RS2=cpu.mycore.dp.rf.a2;
   wire[63:0] A,B,RESULT; assign A=cpu.mycore.dp.srca,B=cpu.mycore.dp.srcb;
   assign RESULT=cpu.mycore.dp.result, IMM=cpu.mycore.dp.gimm.y;
   initial begin wait(reset_); #180
      $display("X5=%h  (should contain '00000007')",cpu.mycore.dp.rf.rf[5]);
      $finish;
   end
   riscvmem cpu(clock,reset_);
endmodule
```

OBIETTIVI

1) **Capire la programmazione del Timer 8254**
2) **Capire la programmazione della Porta Seriale 16550A)**

1) Capire la programmazione del Timer 8254



DOMANDA: Generare una interruzione dopo Δt=50ms utilizzando il timer 8254.

RISPOSTA: Nel timer 8254 la frequenza di riferimento vale Fc=1.19318 MHz (Tc=1/1.19318·10^6 ≈ 838ns). Da cui si ricava che la costante da scrivere per programmare il timer è N=Fc·Δt=1.19318·10^6·50·10^{-3} ovvero

N ≈ 59659

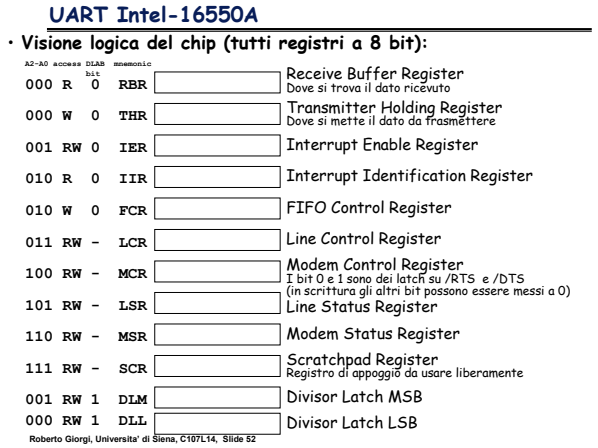
Tale valore va scritto nel registro CR0 (quindi con indirizzo 0x0040).
Inoltre per programmare il contatore 0 con costante di tempo a 16bit LSB+MSB, modo operativo 0 e conteggio binario la costante di tempo da scrivere nel registro CWR (quindi con indirizzo 0x0043) vale 0011'0000 ovvero in esadecimale: 0x30. Il codice equivalente in C risulta quindi:

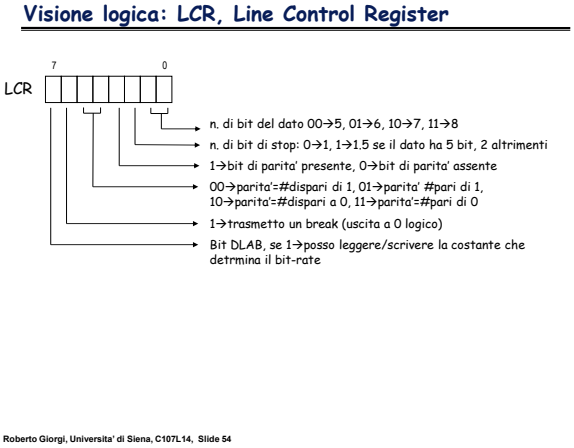
```
int init_8254(void)
{
     int cr_l, cr_h;
    outp(0x0043, 0x30);  /* contatore 0, costante di tempo a 16 bit,
                                                modo 0, conteggio binario */
     cr_l = 59659 & 0x000000FF;
     cr_h = (59659 >> 8) & 0x000000FF;
     outp(0x0040, cr_l);
     outp(0x0040, cr_h);
}
```

2) Capire la programmazione della Porta Seriale 16550A)

Es. n.1 del 4/12/2006: https://arcal.diism.unisi.it/esercizi1/c1061204-sol.pdf

Si supponga di dover programmare la porta seriale 16550A in modo da generare una trama di 5 bit dati, 1.5 bit di stop, parità dispari (numero di "uni" dispari) a un baud rate pari a 9600. Tale chip sia mappato a partire dall' indirizzo 0x 8000 03F8 dello spazio di memoria di un processore RISC-V. Si scriva il codice assembly necessario per programmare tale chip supponendo che la frequenza esterna di clock sia pari a 1.8432 MHz.





SVOLGIMENTO DELL'ESERCIZIO 1

Dalla figura a destra si ricava che il valore da scrivere nel registro LCR per programmare la porta seriale in modo da generare una trama di 5 bit dati, 1.5 bit di stop, parità dispari (numero di "uni" dispari) e ricordando che occorreva predisporre l'accesso al registro DLM/DLL ponendo ad 1 il bit DLAB (bit7 di LCR) e':

$$V_{LCR}=(1000'1100)_2=(140)_{10}$$

Inoltre, dato che la trasmissione avviene ad un baud rate BR=9600 si ricava che il valore della costante di tempo da scrivere nel Divisor Latch e' (dato che la frequenza esterna di clock del chip 16550A e' pari a F_{CK}=1.8432 MHz):

$$V_{DL}=F_{CK}/BR/16=1843200/9600/16=12$$

Dalla figura a sinistra si ricava che gli offset (rispetto all'indirizzo base 0x8000'03F8 a cui è mappato il chip) delle porte LCR, DLL, DLM sono rispettivamente:

$$OFF_{LCR}=3, \quad OFF_{DLL}=0, \quad OFF_{DLM}=1$$

Per programmare la porta come richiesto una possibile versione del codice assembly RISC-V e':

```
addi   t1, x0, 1            # >
slli   t1, t1, 31           # >
ori    t1, t1, 0x03F8       # carica indirizzo base 16550° pari a 0x8000'03F8
addi   t2, x0, 140          # carica V_LCR
addi   t3, x0, 12           # carica V_DLL
sb     t2, 3(t1)            # scrive in LCR (OFF_LCR=3)
sb     t3, 0(t1)            # scrive in DLL (OFF_DLL=0)
sb     x0, 1(t1)            # scrive in DLM (OFF_DLM=1) (gli 8 bit più significativi
                            # di V_DLM sono zero)
```

ESERCIZI in LABORATORIO – RETI LOGICHE SEQUENZIALI (ESERCITAZIONE/LAB #06)

OBIETTIVI

1) **Analizzare gli esercizi di esame sulle reti logiche sequenziali in Verilog nel simulatore Verilogger e verificare la correttezza dei relativi diagrammi temporali:**
   - **I. Ripple Counter a 4-bit**
   - **II. Serial Carry Counter a 4-bit**
   - **III. Ring Counter a 4-bit**
   - **IV. Carry Look Ahead Adder a 4-bit**
   - **V. Look Ahead Carry Counter a 4-bit**
   - **VI. Parallel Carry Counter a 4-bit**
   - **VII. Ripple Carry Adder a 4-bit**

---

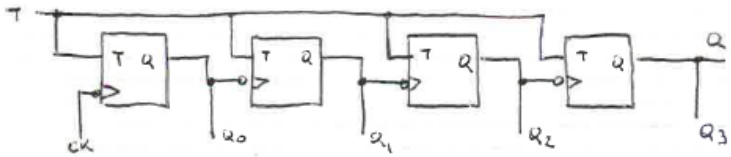
1. Testare le implementazioni in Verilog nel simulatore Verilogger delle seguenti reti logiche sequenziali e verificare la correttezza dei relativi diagrammi temporale

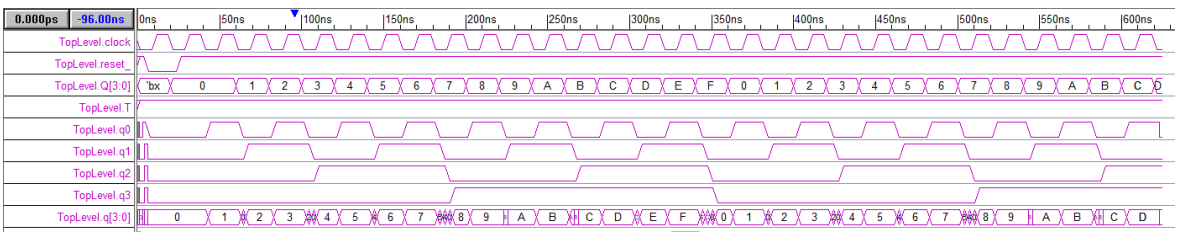
1. Aprire simulatore Verilogger in modalità Verilogger
2. Creare un nuovo progetto HDL (Project-> new HDL Project)
3. Creare un nuovo file HDL nel progetto (Project-> Add HDL file)
4. Salvare il file con il nome della rete logica (e.g., ripple_counter.v)
5. Aprire il file HDL nell'editor di Verilogger con un doppio click
6. Copiare l'implementazione in Verilog e il relativo testbench nel file HDL
7. Salvare il progetto (Project-> Save HDL Project)
8. Compilare cliccando sul pulsante 
9. Verificare nella finestra Report che la compilazione sia avvenuta con successo
10. Eseguire la simulazione cliccando sul pulsante 
11. Verificare la correttezza del diagramma temporale

Le tracce dei compiti con le relative soluzioni sono disponibili al seguente link:
https://arcal.diism.unisi.it/compitini.htm

I. Ripple Counter a 4-bit– Compito del 17-01-2018

Un ripple counter a 4-bit effettua il conteggio attraverso il collegamento di 4 Flip-Flop di tipo T (FFt).
I FFt ricevono tutti in comune il segnale di Toggle (T) che significa che, quando T=1, ad ogni fronte del clock i FFt devono commutare l'uscita: questo segnale viene messo quindi a 1 "fisso". Il clock fa quindi commutare il primo FFt ad ogni ciclo: questo genera il bit meno significativo del conteggio (Q0); il secondo FFt commuta sulla base dell'uscita Q0 (quindi ogni 2 cicli di clock) generando Q1 e così via. Con 4 FFt si generano quindi 4 cifre {Q3,Q2,Q1,Q0} di un numero intero su 4 bit. Il conteggio inizia con valore 0, 1, 2, … arriva a 15 e poi continua di nuovo con 0,1, 2, … . Il difetto di questo contatore è che le cifre Q1, Q2, Q3 non commutano precisamente sui fronti del clock ma hanno un lieve ritardo (crescente per i bit successivi).

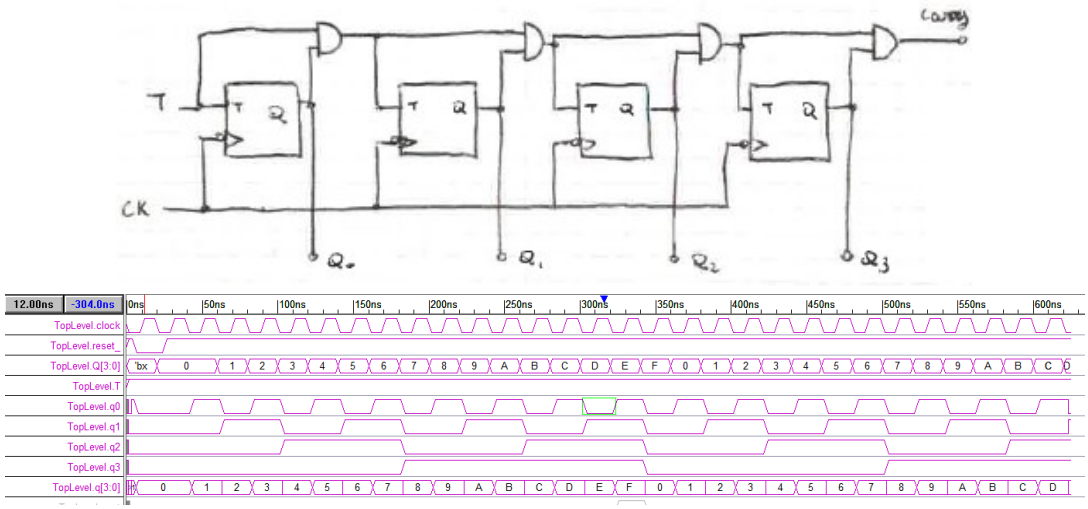




Realizzare in Verilog sia un ripple counter a 4-bit che il relativo testbench: il clock ha un periodo di 10ns; il segnale _reset e' attivo basso: resta alto per 5ns, basso per 20ns, e poi ritorna alto per 600ns. Il contatore inizia il conteggio producendo sull'uscita Q il valore binario 0000 quando il segnale di /reset e'attivato, appena disattivato il /reset il conteggio prosegue. Tracciare il diagramma di temporizzazione come verifica della correttezza dell'unità riportando i segnali clock, /reset, uscita Q per la durata complessiva (625ns). Nota: si può svolgere l'esercizio su carta oppure con ausilio del simulatore salvando una copia dell'output (diagramma temporale) e del programma Verilog su USB-drive del docente. (Per studenti 2014 e anni precedenti descrivere il comportamento di questa rete e disegnare líntero diagramma di temporizzazione, come sopra specificato).

II. Serial Carry Counter a 4-bit – Compito del 14-02-2018

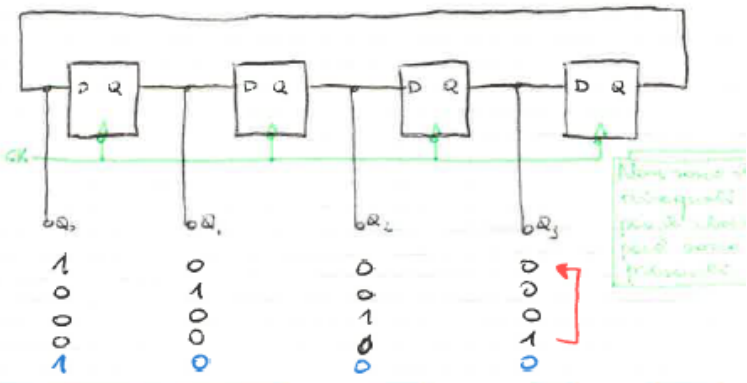
Un Serial Carry Counter a 4-bit effettua il conteggio attraverso il collegamento di 4 Flip-Flop di tipo T (FFt). Gli FFt ricevono tutti in comune il segnale di clock che attiva i FFt ad ogni ciclio di clock, andando a mitigare il ritardo generato dal Ripple Counter. Per poter abilitare il conteggio, il segnale di Toggle (T) viene impostato ad un "1" fisso. Quindi il primo FFt commuta e produce il bit meno significativo sull'uscita Q0. Il secondo FFt commuta quando il segnale Toggle (T) e l'uscita Q0 sono entrambi "1" (il check avviene attraverso una porta AND), producendo l'uscita Q1 e cosi' via. Il Carry del conteggio verra' generato solo quando il quarto FFt avra' prodotto l'output Q4. Con 4 FFt si generano quindi 4 cifre {Q3,Q2,Q1,Q0} di un numero intero su 4 bit ed il segnale Carry "cout". Il conteggio inizia con valore 0, 1, 2, … arriva a 15 e poi continua di nuovo con 0,1, 2, … . Il difetto di questo contatore è che le porte AND generano un lieve ritardo nel produrre il segnale. Questo ritardo si propaga fino alla generazione del "cout" (critical path).

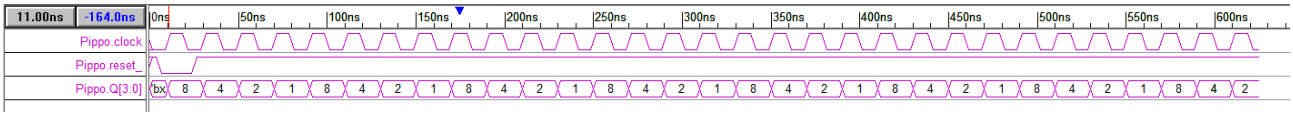


Realizzare in Verilog sia un serial carry counter a 4-bit che il relativo testbench: il clock ha un periodo di 10ns; il segnale _reset e' attivo basso: resta alto per 5ns, basso per 20ns, e poi ritorna alto per 600ns. Il contatore inizia il conteggio producendo sull'uscita Q il valore binario 0000 quando il segnale di /reset e'attivato, appena disattivato il /reset il conteggio prosegue. Tracciare il diagramma di temporizzazione come verifica della correttezza dell'unità riportando i segnali clock, /reset, uscita Q per la durata complessiva (625ns). Nota: si può svolgere l'esercizio su carta oppure con ausilio del simulatore salvando una copia dell'output (diagramma temporale) e del programma Verilog su USB-drive del docente. (Per studenti 2014 e anni precedenti descrivere il comportamento di questa rete e disegnare líntero diagramma di temporizzazione, come sopra specificato).

III. Ring Counter – Compito del 14-11-2017

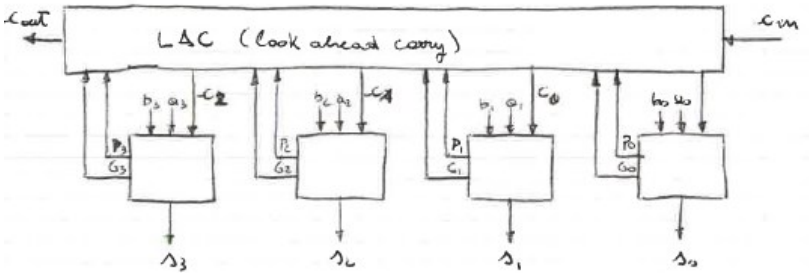
Il Ring Counter si basa su un Flip Flop di tipo D (FFd) sensibile al fronte si salita del clock che effettua il conteggio decrescente su registro Q a 4 bit da 8 verso 1. Inizialmente il valore del registro Q e' impostato a 0000. Al primo fronte di salita del clock, il conteggio parte scrivendo "1" nel bit piu' significativo (1000). Nel secondo fronte di salita del clock, il valore "1" verra' spostato nella seconda posizione del registro Q (0100) e cosi' via. Quando il valore "1" viene impostato nel bit meno significativo (0001), lo spostamento successivo fa tornare il valore "1" nel bit piu' significativo (1000) andando a formare un anello.

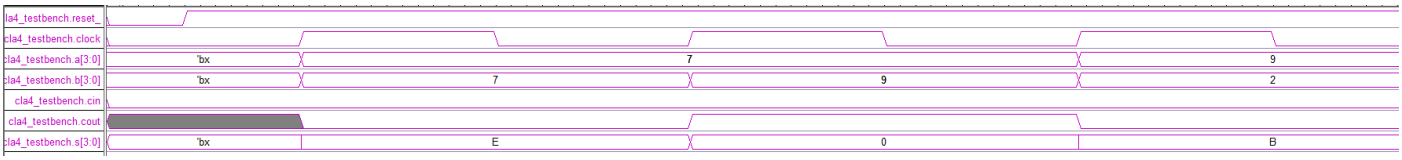




IV. Carry Look Ahead Adder a 4-bit – Compito 08-02-2019 (vedi Lezione 06)

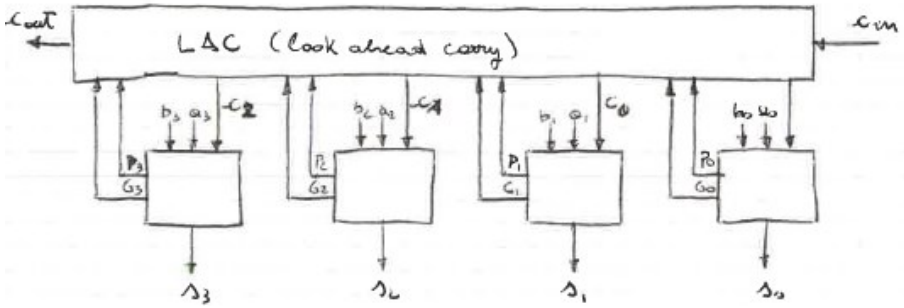
Realizzare in Verilog un sommatore a 4-bit di tipo carry-look-ahead. Il testbench e' dato. Tracciare il diagramma di temporizzazione come verifica della correttezza dell'unità riportando i segnali clock, /reset, uscita S e Cout (carry in uscita al sommatore) per la durata complessiva (45ns). Nota: si può svolgere l'esercizio su carta oppure con ausilio del simulatore salvando una copia dell'output (diagramma temporale) e del programma Verilog su USB-drive del docente

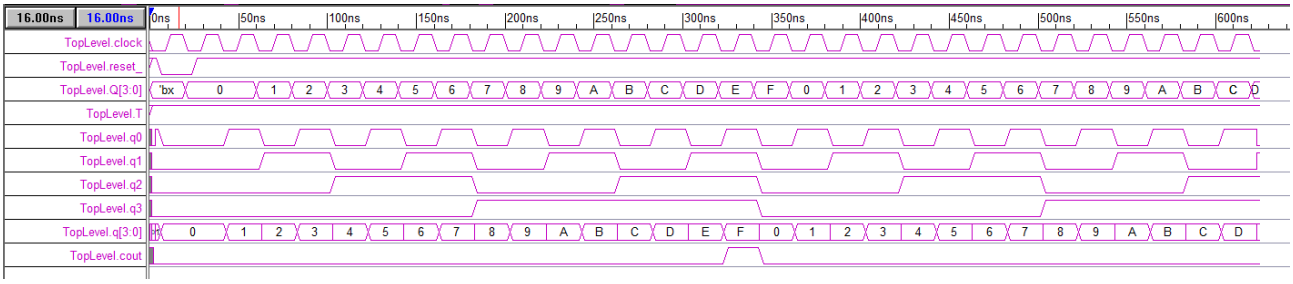




V. Look Ahead Carry Counter a 4-bit– Compito del 11-04-2018

Il Look Ahead Carry Counter effettua il conteggio utilizzando la stessa implementazione del carry look ahead adder a 4-bit. La differenza consiste negli stimoli prodotti dal testbench.

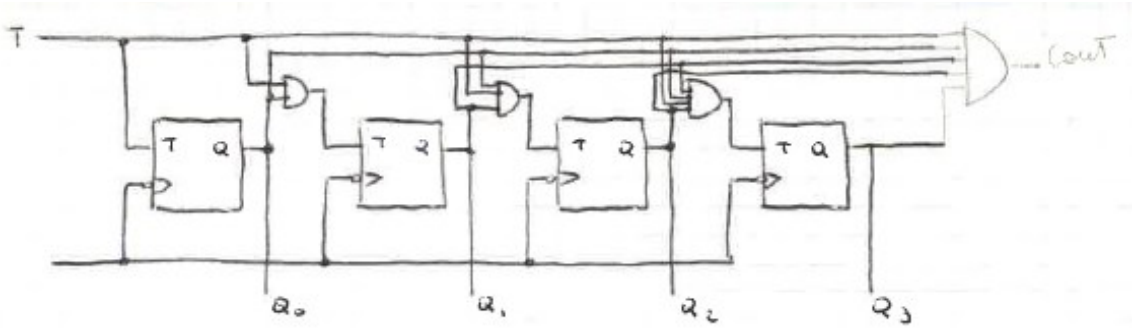


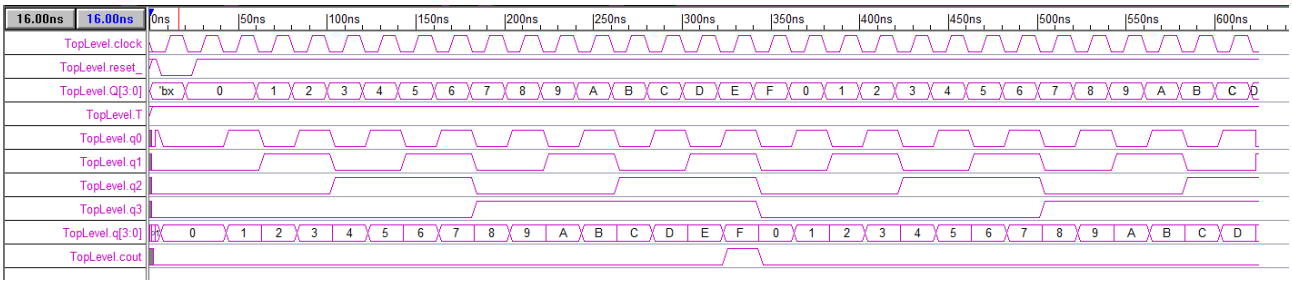


Realizzare in Verilog un contatore look-ahead-carry a 4-bit ottenuto collegando l'uscita S (opportunamente memorizzata su flip-flop-D) all'ingresso B di un sommatore look-ahead-carry mentre l'ingresso A e' fissato a 4'b0001. Realizzare la descrizione del circuito in Verilog e il relativo testbench: il clock ha un periodo di 10ns; il segnale _reset e' attivo basso: resta alto per 5ns, basso per 20ns, e poi ritorna alto per 600ns. Il contatore inizia il conteggio producendo sull'uscita Q il valore binario 0000 quando il segnale di /reset e'attivato, appena disattivato il /reset il conteggio prosegue. Tracciare il diagramma di temporizzazione come verifica della correttezza dell'unità riportando i segnali clock, /reset, uscita Q e Cout (carry in uscita al contotare) per la durata complessiva (625ns). Nota: si può svolgere l'esercizio su carta oppure con ausilio del simulatore salvando una copia dell'output (diagramma temporale) e del programma Verilog su USB-drive del docente. (Per studenti 2014 e anni precedenti descrivere il comportamento di questa rete e disegnare líntero diagramma di temporizzazione, come sopra specificato).

VI. Parallel Carry Counter a 4-bit– Compito del 01-03-2018

Il Parallel Carry Counter a 4-bit effettua il conteggio attraverso il collegamento di 4 Flip-Flop di tipo T (FFt).
Gli FFt ricevono tutti in comune il segnale di clock che attiva i FFt ad ogni ciclio di clock, andando a mitigare il ritardo generato dal Ripple Counter. Inoltre, il Parallel Carry va a mitigare il ritardo prodotto dalle porte AND del Serial Carry Counter andando ad espandere le porte AND come somma di prodotti (teorema di Shannon). Per poter abilitare il conteggio, il segnale di Toggle (T) viene impostato ad un "1" fisso. Quindi il primo FFt commuta e produce il bit meno significativo sull'uscita Q0. Il secondo FFt commuta quando il segnale Toggle (T) e l'uscita Q0 sono entrambi "1" (il check avviene attraverso una porta AND), producendo l'uscita Q1. Il terzo FFt commuta quando il segnale di Toggle e gli output precedenti Q0 e Q1 sono uguali a "1". Il quarto FFt commuta quando il segnale di Toggle e gli output precedenti Q0,Q1,Q2 sono uguali a "1". Il Carry in uscita verra' prodotto quando mettendo in AND il segnale di Toggle e tutti gli output dei FFt Q0,Q1,Q2,Q3. Si generano quindi 4 cifre {Q3,Q2,Q1,Q0} di un numero intero su 4 bit ed il segnale di Carry cout. Il conteggio inizia con valore 0, 1, 2, … arriva a 15 e poi continua di nuovo con 0,1, 2, …





Realizzare in Verilog sia un parallel carry counter a 4-bit che il relativo testbench: il clock ha un periodo di 10ns; il segnale _reset e' attivo basso: resta alto per 5ns, basso per 20ns, e poi ritorna alto per 600ns. Il contatore inizia il conteggio producendo sull'uscita Q il valore binario 0000 quando il segnale di /reset e'attivato, appena disattivato il /reset il conteggio prosegue. Assumere inoltre che le porte AND abbiano un ritardo di 1ns. Tracciare il diagramma di temporizzazione come verifica della correttezza dell'unità riportando i segnali clock, /reset, uscita Q e Cout (carry in uscita al contotare) per la durata complessiva (625ns). Nota: si può svolgere l'esercizio su carta oppure con ausilio del simulatore salvando una copia dell'output (diagramma temporale) e del programma Verilog su USB-drive del docente. (Per studenti 2014 e anni precedenti descrivere il comportamento di questa rete e disegnare líntero diagramma di temporizzazione, come sopra specificato). Opzionalmente: discutere che differenza c'e' su Cout rispetto allo stesso contatore ma con serial-carry.

VII. Ripple Carry Adder a 4-bit – vedi Lezione 06

```
module full_adder(input a,input b,input c_in,output
s,output c);

        assign s = a ^ (b ^ c_in);
        assign c = (a & b) | (c_in & (a | b));

endmodule

module rc_adder(input wire[N-1:0]a,input wire[N-1:0]b,
input wire cin,

    output wire [N-1:0]s,output wire cout);
    parameter N = 32;  // bits
    wire [N-1:0] _c;
    assign cout = _c[N-1];

    full_adder add0(.a(a[0]),.b(b[0]),.c_in(cin),
                                    .s(s[0]),.c(_c[0]));
    genvar i;
    generate
        for (i = 1; i < N; i = i + 1) begin : gen_adder
            full_adder addN(.a(a[i]),.b(b[i]),.c_in(_c[i-1]),
                                        .s(s[i]),.c(_c[i]));
        end
    endgenerate

endmodule
```

OBIETTIVI

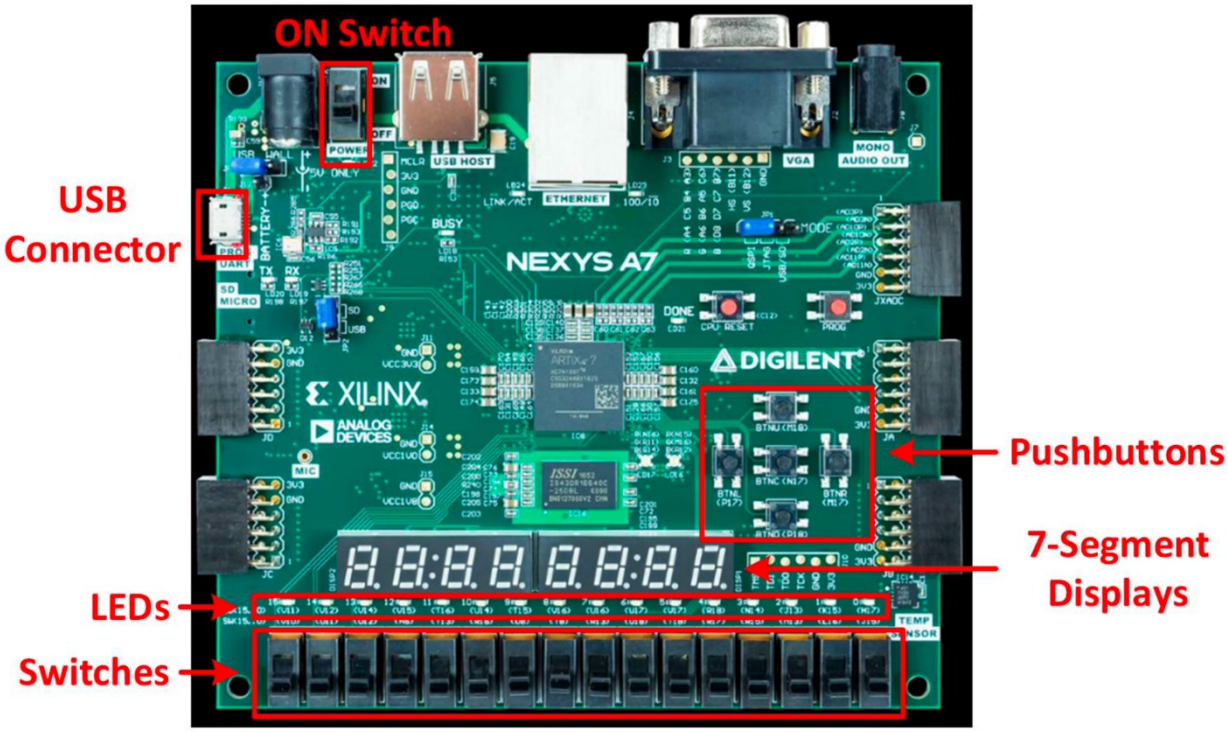
1) Introduzione e guida agli strumenti hardware e software necessari all'esercitazione
2) Analizzare ed eseguire programmi interagendo con una board FPGA reale

   I. eseguire unprogrammain RISCV assembly
   II. Analizzare la traccia del programma RISCV in GTKWave
   III. eseguire un programma in RISCV assembly che faccia lampeggiare un led della board
   IV. eseguire un programma in RISCV assembly che abiliti un interruttore ad accendere un led della board
   V. Eseguire un programma in RISCV assembly che stampi il numero 51 sul display della board

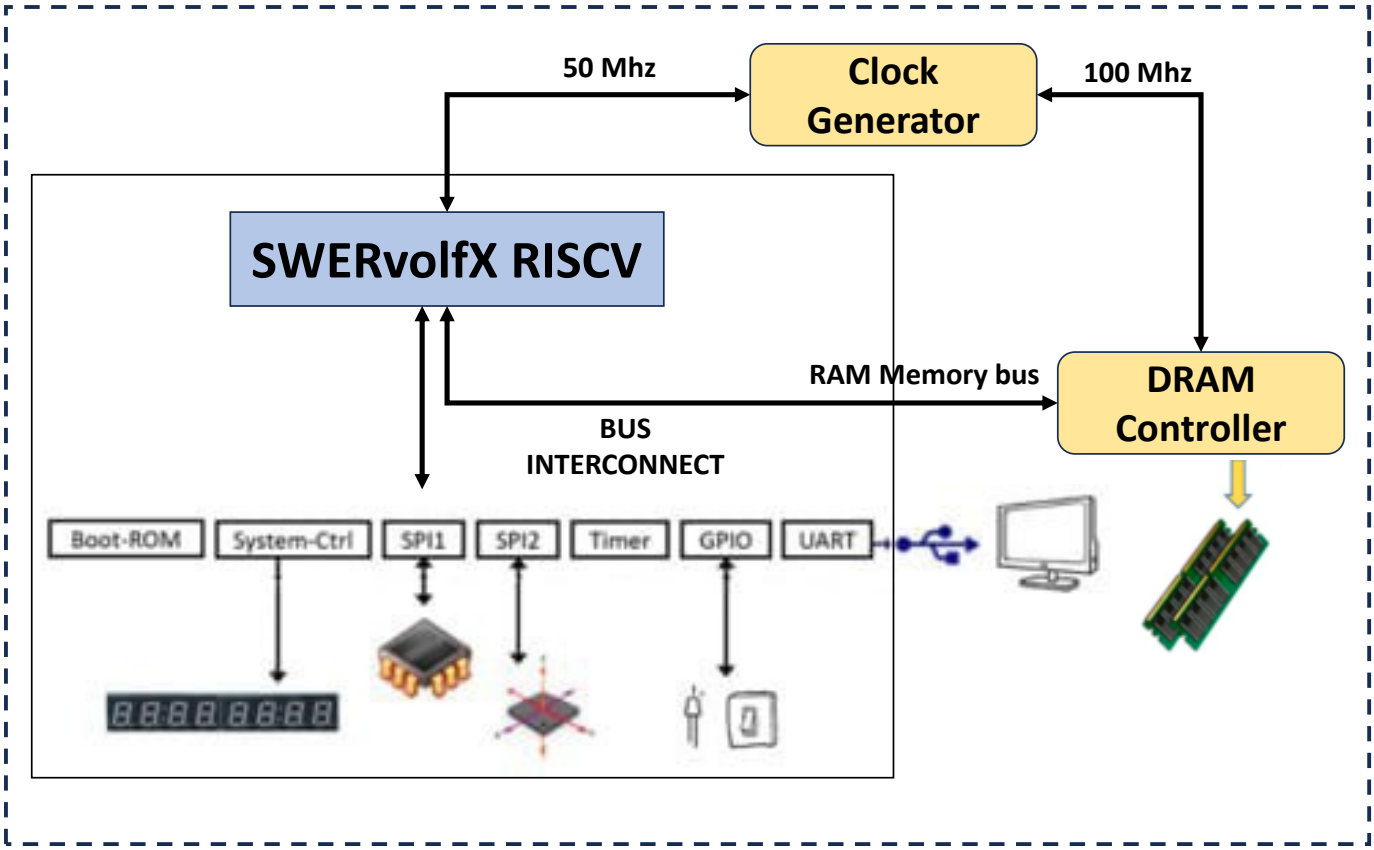
1. Introduzione e guida agli strumenti hardware e software necessari all'esercitazione

- **Esercitazione basata sul corso prodotto da Imagination Technologies "RVFPGA Undestanding Computer Architecture".**

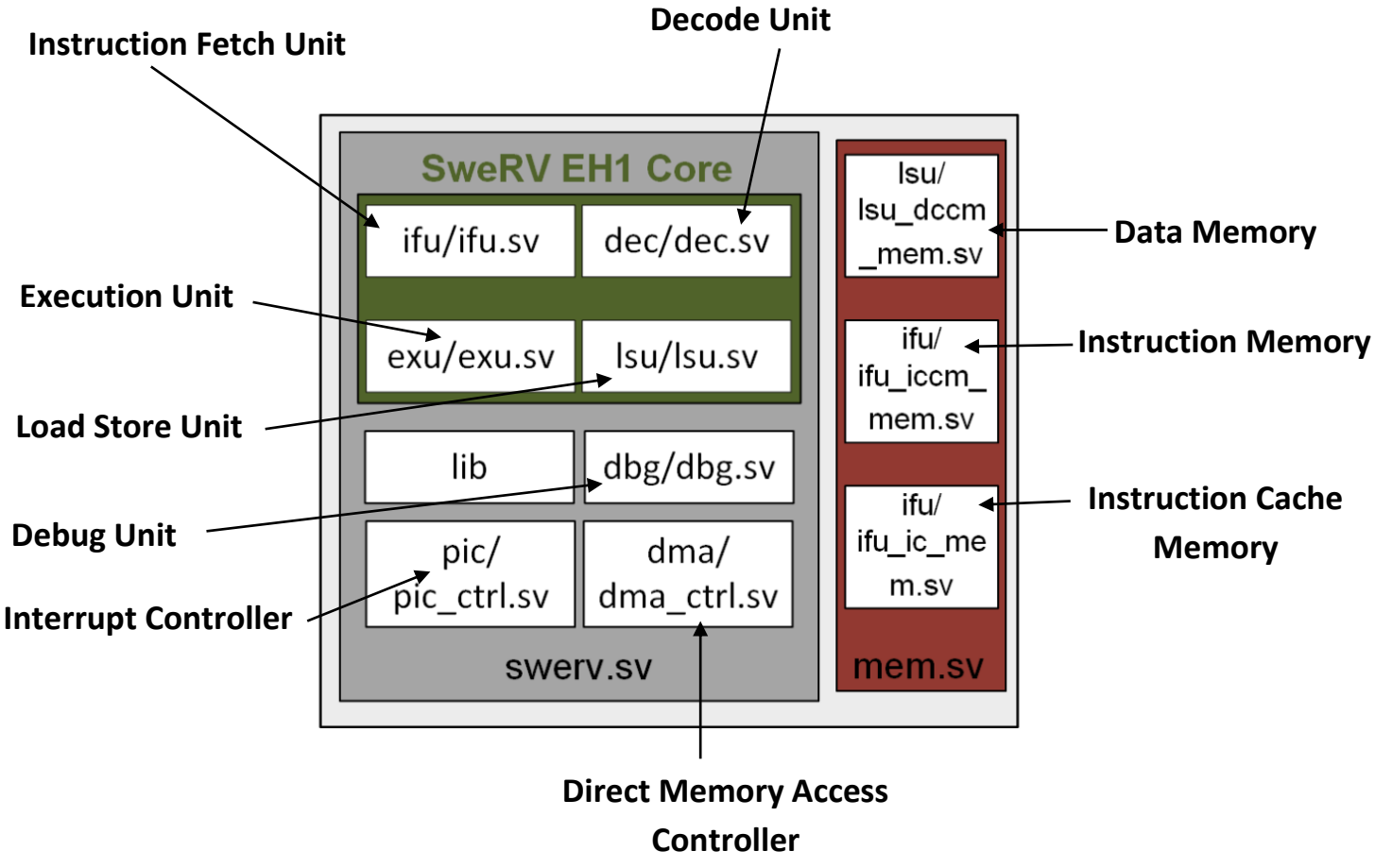
- **Gli esercizi verranno svolti utilizzando una versione del core RISCV SweRVolf adattato per essere eseguito su una board FPGA reale: la Digilent Nexys 4 DDR/Nexis A7.**



- **Di seguito possiamo analizzare il design del progetto, che ci permette di accedere alle periferiche della board (per esempio, led, interruttori ecc)**



- **Di seguito la struttura del progetto verilog del RISCV core SweRV EH1 Core. Il progetto e' organizzato in due blocchi: un wrapper (box grigio) che contiene il core e alcuni blocchi di debugging ed interrupt, e il blocco Dati/Istruzioni (box rosso).**



- **Le periferiche della board sono mappate ad indirizzi specifici**

SweRVolfX Memory Map

System	Address
Boot ROM	0x80000000 - 0x80000FFF
System Controller	0x80001000 - 0x8000103F
SPI1	0x80001040 - 0x8000107F
SPI2	0x80001100 - 0x8000113F
Timer	0x80001200 - 0x8000123F
GPIO	0x80001400 - 0x8000143F
UART	0x80002000 - 0x80002FFF

- **Altri tool software necessari per questa esercitazione**

1. **Visual Studio Code**
2. **PlatformIO extention di Visual Studio Code**
3. **VerilatorSIM**
4. **GTK-Wave**

- **Viene messa a disposizione una macchina virtuale Virtualbox (richiesta versione 7.0.8) gia' configurata per l'esercitazione che e' possibile avviare cliccando sul link** 

"Win10RVFPGA" presente sul desktop del pc del laboratorio e poi sul tasto

STEP1) Avviare VirtualBox

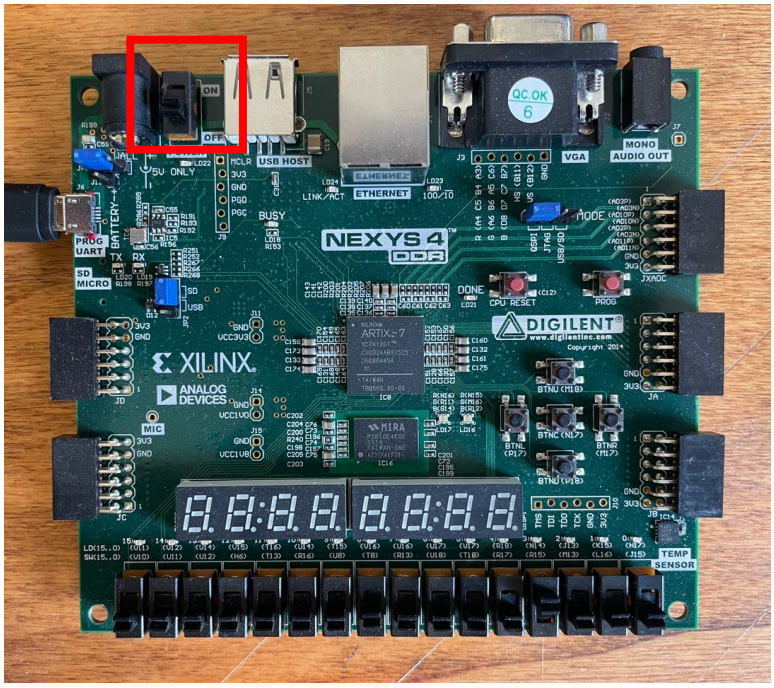
STEP2) Cliccare sul file Win10RVFPGA.vbox

STEP3) Cliccare sul pulsante e attendere l'avvio del sistema

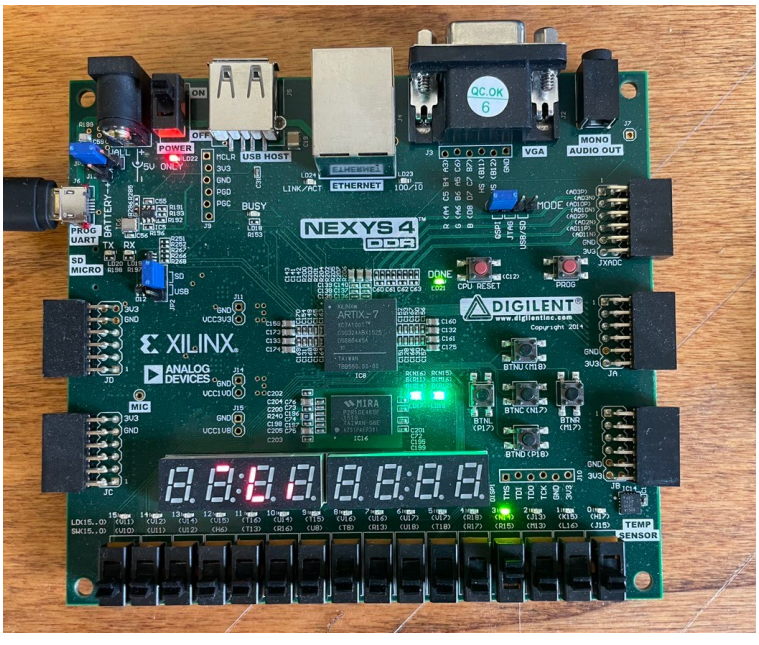
STEP4) Inserire la password: rvfpga

- **Accensione della board FPGA (Saltare al punto 2 della guida se non si ha a disposizione una board FPGA)**

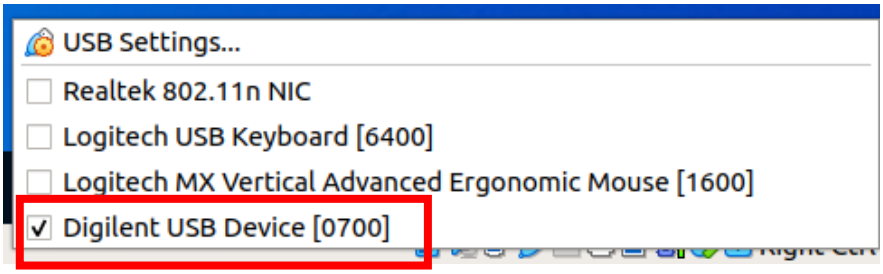
STEP1) Accendere la board FPGA spostando su ON l'interruttore che si trova nella parte in alto a sinistra della boad (adiacente al connettore dell'alimentazione).



STEP2) I LED presenti sulla board inizieranno a lampeggiare e quando rimarranno fissi, la board e' pronta per essere programmata (vedi foto seguente)

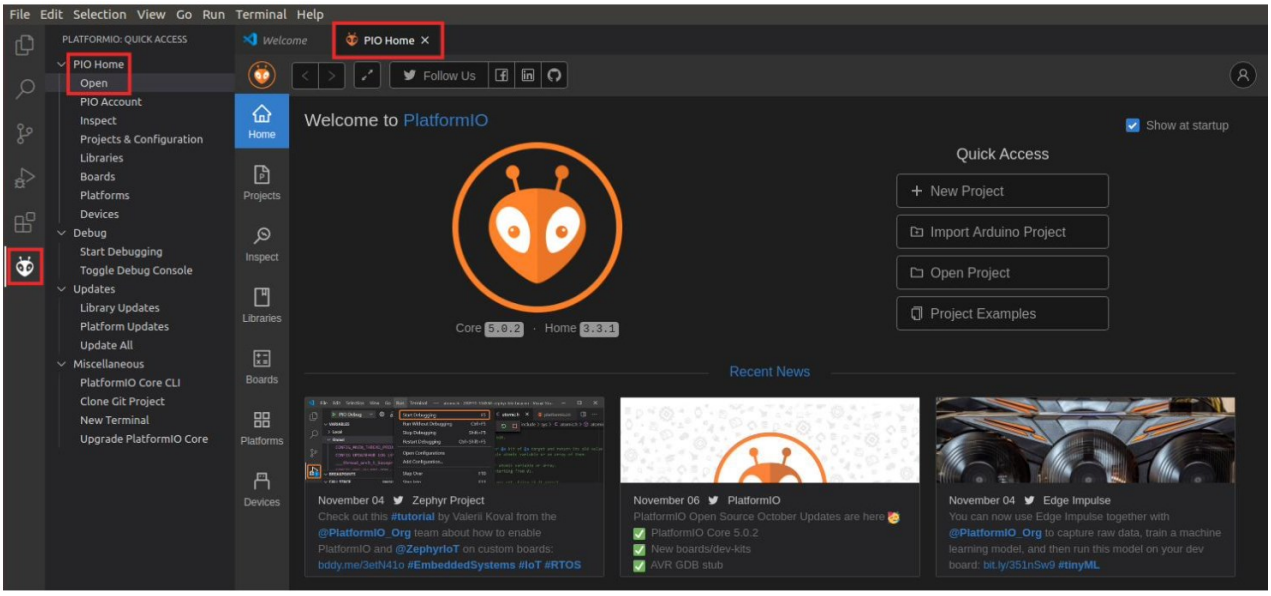


STEP3) Una volta avviata la board e la macchina virtuale, assicurarsi che la board sia connessa cliccando con il tasto destro del mouse sull'icona che si trova nella toolbar in basso destra. Nel menu successivo, spuntare l'opzione "Digilent USB Device"



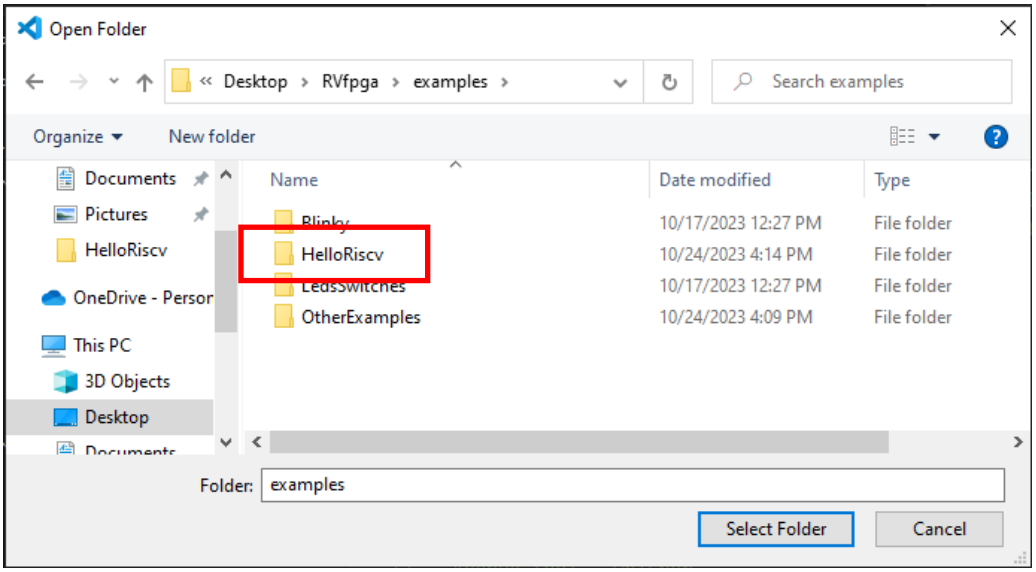
2. Analizzare ed eseguire un programma assembly RISCV

Aprire Visual Studio Code cliccando sull'icona presente sul desktop e spostarsi sulla estensione PlatformIO cliccando sul pulsante nella barra verticale a sinistra.

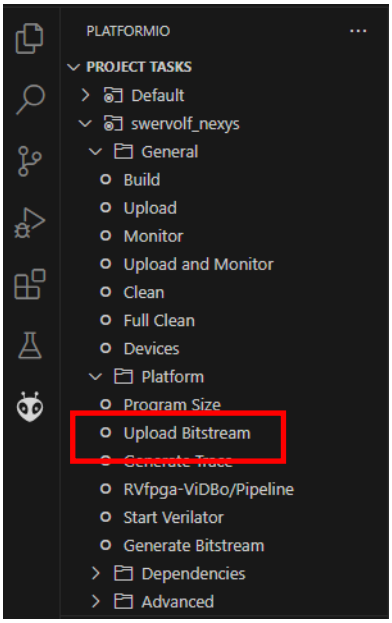


I. Eseguire un programma in RISCV assembly

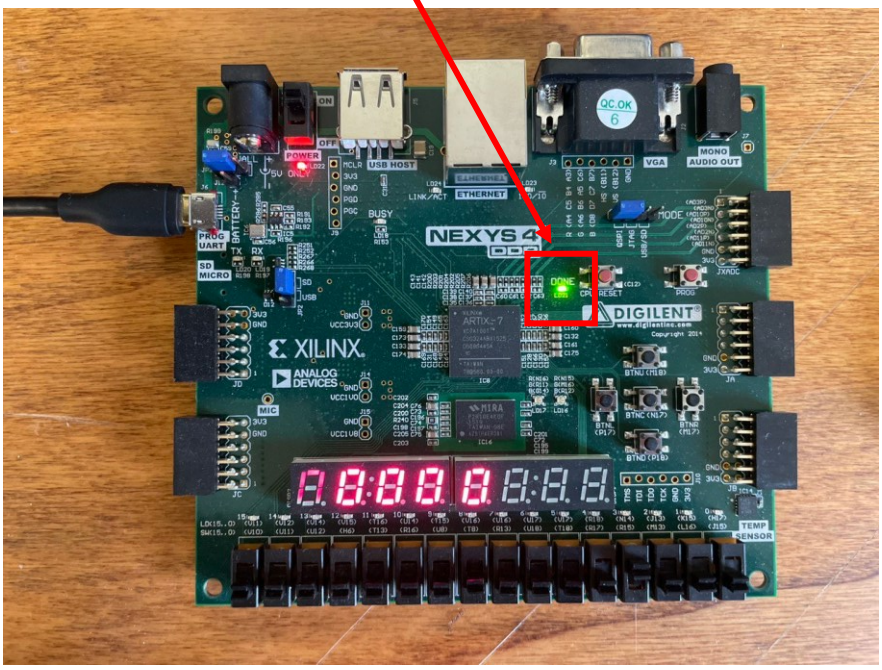
STEP1) Aprire il progetto HelloRiscv cliccando su File->OpenFolder nel menu' in alto a sinistra di Visual Studio Code, selezionare la cartella "HelloRiscv e cliccare sul pulsante "Select Folder".



STEP2) Prima di eseguire il programma dobbiamo programmare la board FPGA caricando il "Bitstream". Spostarsi nell'estensione PlatformIO cliccando sul pulsante nel menu a sinistra di Visual Studio Code e cliccare su "Upload Bitstream"

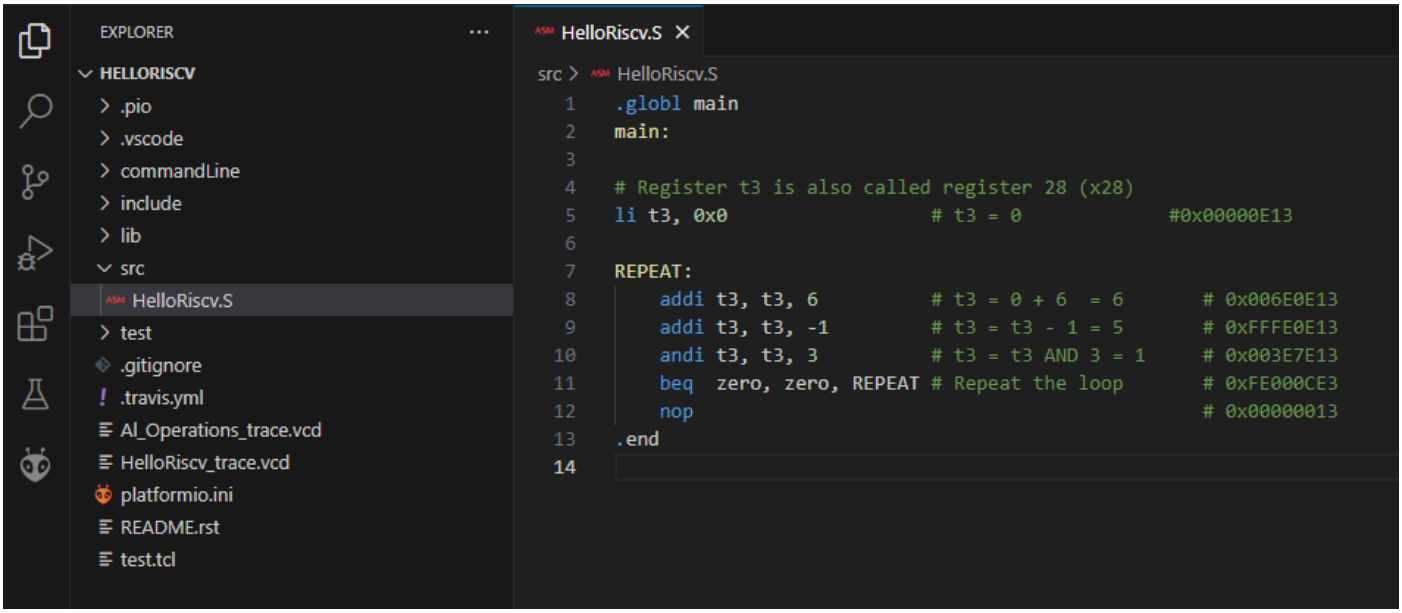


STEP3) Se l'upload e' avvenuto con successo, il terminale stampera' il messaggio `[SUCCESS]` **e la board avra' il LED "DONE" acceso e di colore verde.**



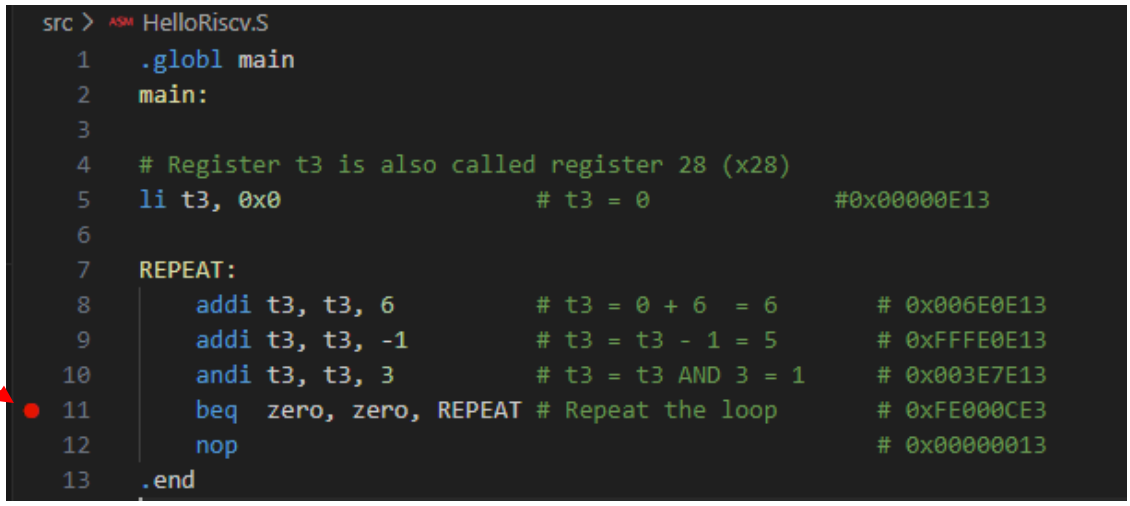
STEP3.1) Nel caso l'upload fallisse mostrando il messaggio di errore `[FAILED]` **nel terminale, installare i driver seguendo le istruzioni nell'appendice A.**

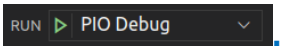
STEP4) Eseguire il programma sulla board FPGA. Cliccare sul pulsante  **nel menu a sinistra di Visual Studio Code e poi cliccare sul file HelloRiscv.S**



STEP5) Il programma che stiamo eseguento e' un loop continuo che va ad aggiornare il registro "t3". Inseriamo un breakpoint sull'istruzione "beq" per stoppare l'esecuzione ed effettuare del debugging. Per inserire un breakpoint sull'istruzione "beq" clicchiamo alla sinistra del numero "11" nella colonna che numera le istruzioni ed apparira' un pallino rosso 

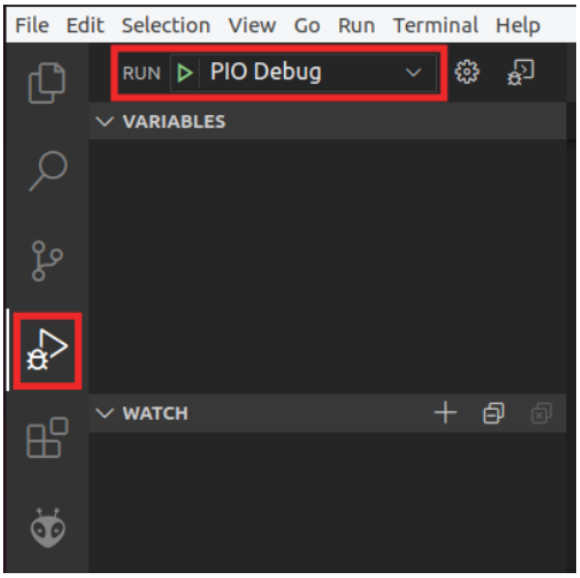
BREAKPOINT



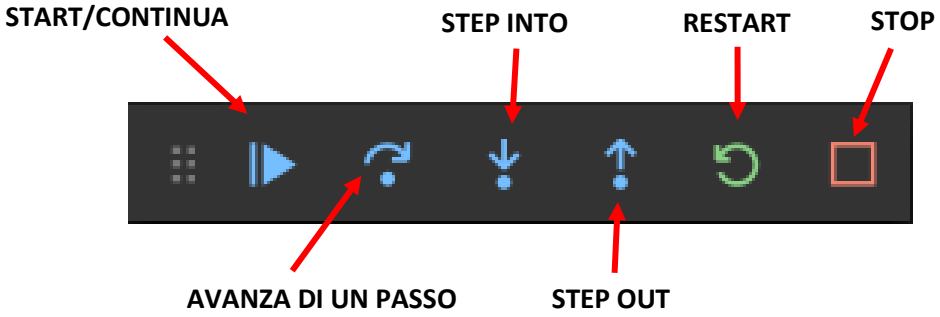
STEP6) Compilare il programma cliccando sul pulsante "Run and Debug"  **nella toolbar a sinistra. La compilazione sarà terminata quando apparirà il messaggio "success" nella finestra Terminal. Una volta terminata la compilazione apparirà in alto a sinistra il pulsante** 

.

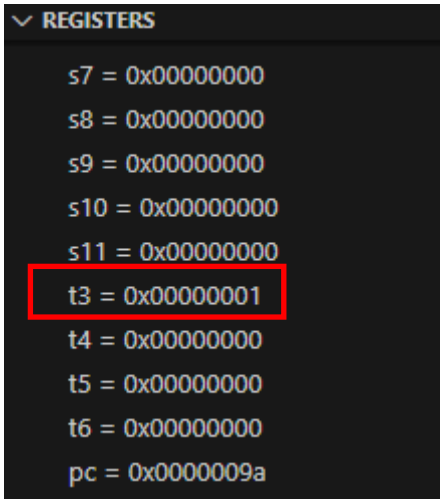
STEP7) E' possibile eseguire il programa cliccando sul pulsante a forma di freccia verde



STEP8) Controllare l'esecuzione attraverso la debugging toolbar. Cliccando una volta sul tasto , il debugging partira' e si fermera' alla prima istruzione (breakpoint automatico). Cliccando nuovamente sul tasto , partira' l'esecuzione delle istruzioni.



STEP9) Possiamo controllare lo stato dei registri utilizzati dal programma nel menu "Registers" a sinistra.



STEP10)Stoppiamo l'esecuzione cliccando sul pulsante della debugging toolbar.

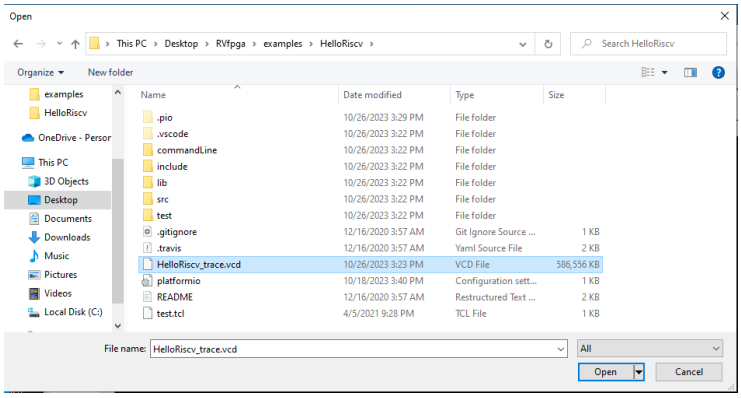
II. Analizzare la traccia del programma RISCV in GTKWave

• Possiamo analizzare e debuggare le forme d'onda dei segnali del nostro programma attraverso il tool GTKWave. La traccia del programma viene fornita nella macchina virtuale virtualbox, ma puo' essere anche generata attraverso l'utilizzo di un simulatore verilator (attualmente non compatibile con windows).

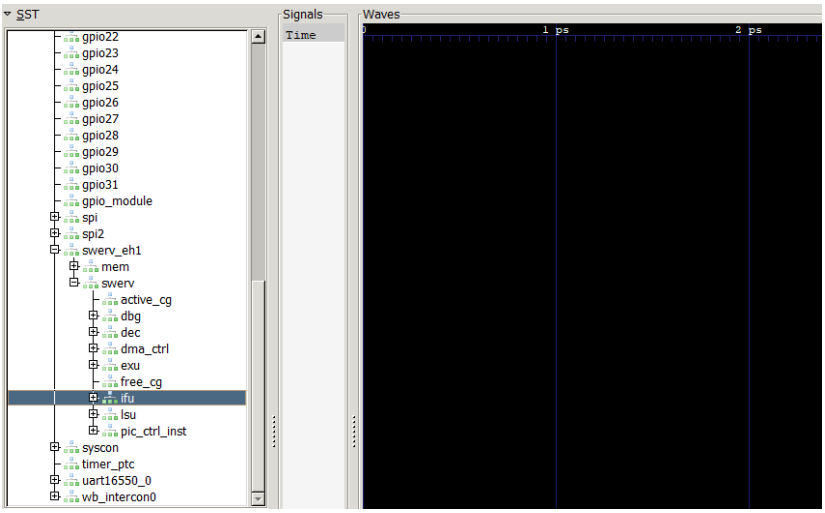
STEP1) Lanciamo il programma gtkwave facendo doppio click sull'icona 

presente sul desktop della macchina virtuale.

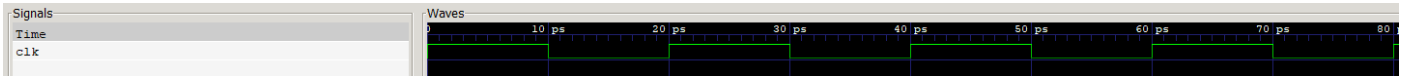
STEP2) Carichiamo la traccia del programma cliccando sul "File" -> "Open New Tab". Selezioniamo il file "HelloRiscv_trace.vcd" presente nella cartella "Desktop/Rvfpga/examples/HelloRiscv" e clicchiamo su "Open"



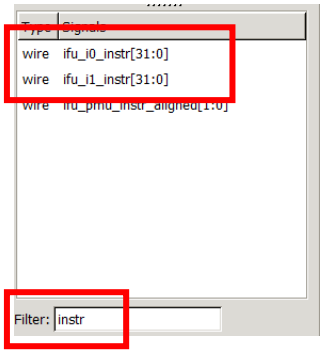
STEP3) Adesso dobbiamo aggiungere i segnali che vogliamo analizzare nella colonna "Signals". Per fare cio', espandiamo la gerarchia presente nel pannello in alto a sinistra: "rvfpgasim->swervolf->swerv_eh1->swerv" e clicchiamo sul modulo "ifu" (istruction fetch unit).



STEP4) Visualizziamo l'onda del segnale di clock. Cliccare sul segnale di clock clk presente nel tab in basso a sinistra e trascinarlo nel tab "Signals". Cliccare sul pulsante  per ridurre lo zoom e visualizzare l'onda del segnale di clock.

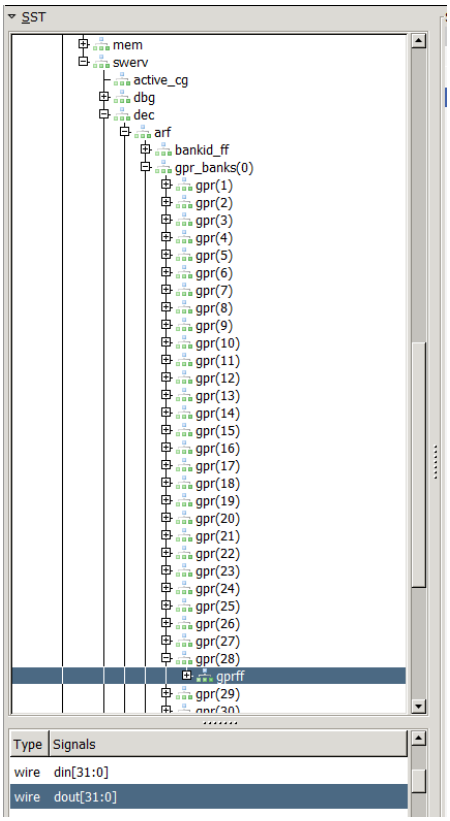


STEP5) Aggiungiamo al tab "Signals" i segnali che contengono le istruzioni della Instruction Fetch Unit. Trascinare i segnali "ifu_i1_instr[31:0]" e "ifu_i0_instr[31:0]" nel tab "Signals". Potete cercare i segnali utilizzando il campo "Filter"



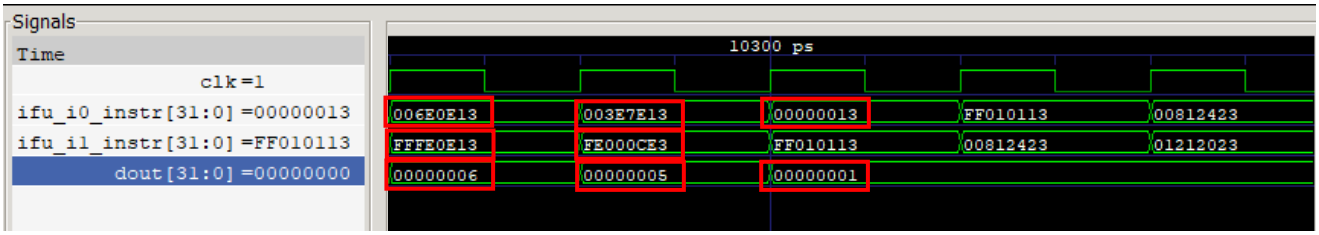
STEP6) Aggiungiamo il segnale che contiene i valori del registro "t3" (x28) utilizzato nel programma. Espandiamo la gerarchia nel tab in alto a sinistra:

swerv->dec->arf->gpr_banks(0)->gpr(28) e clicchiamo sul modulo "gprff" e trasciniamo il segnale "dout[31:0] nel tab "Signals"



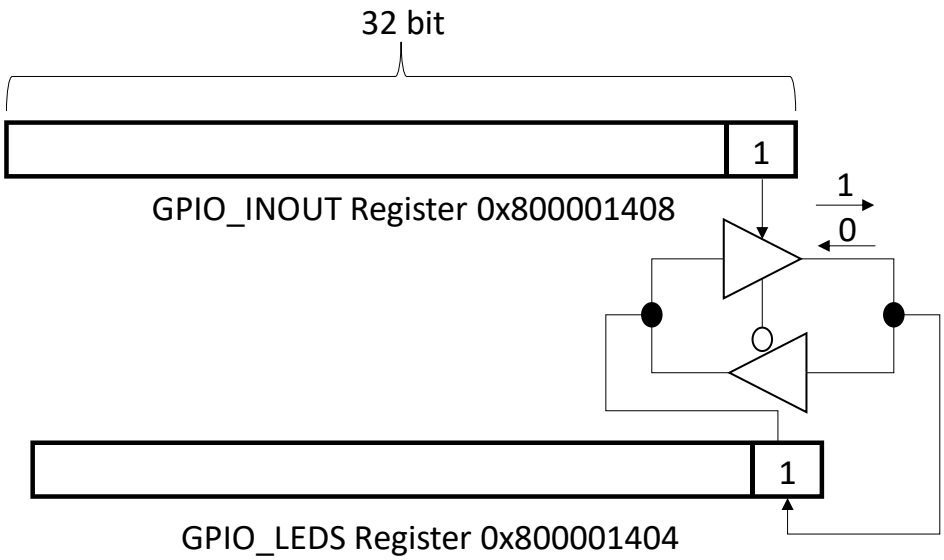
STEP7) Spostiamoci nella tab “Wave” intorno ai 10100 ps, dove possiamo notare come le istruzioni del nostro programma vengono caricare nei segnali ifu_i0_instr[31:0] e ifu_i1_instr[31:0]

```
            li   t3, 0x0              # t3 = 0              # 0x00000E13
REPEAT:     addi t3, t3, 6            # t3 = 0 + 6 = 6      # 0x006E0E13
            addi t3, t3, -1           # t3 = 5              # 0xFFFE0E13
            andi t3, t3, 3            # t3 = 5 & 3 = 1      # 0x003E7E13
            beq  zero, zero, REPEAT   # Repeat the loop     # 0xFE000CE3
            nop                       # nop                 # 0x00000013
REPEAT:     addi t3, t3, 6            # t3 = 1 + 6 = 7      # 0x006E0E13
            addi t3, t3, -1           # t3 = 7 - 1 = 6      # 0xFFFE0E13
            andi t3, t3, 3            # t3 = 6 & 3 = 2      # 0x003E7E13
            beq  zero, zero, REPEAT   # Repeat the loop     # 0xFE000CE3
```

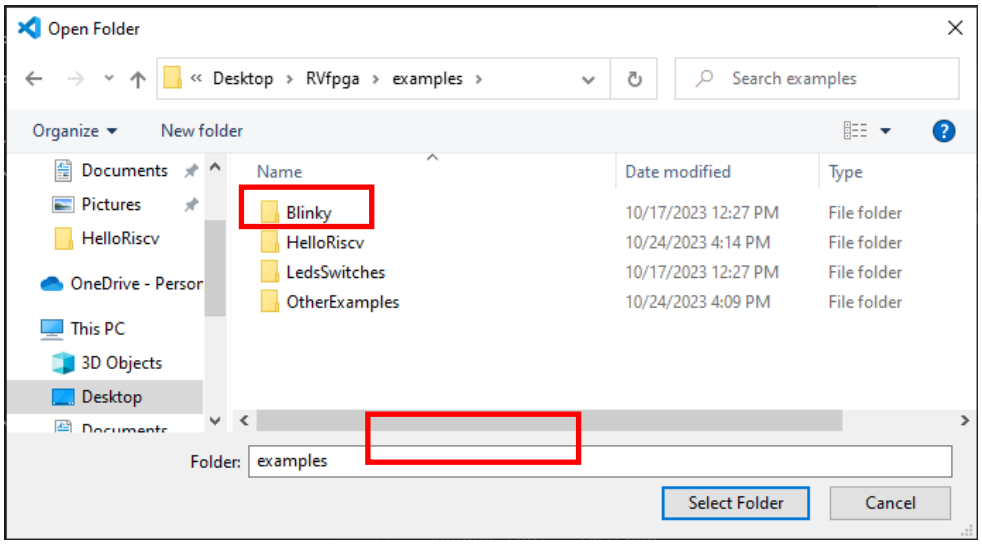


III. Eseguire un programma in RISCV assembly che faccia lampeggiare un led della board

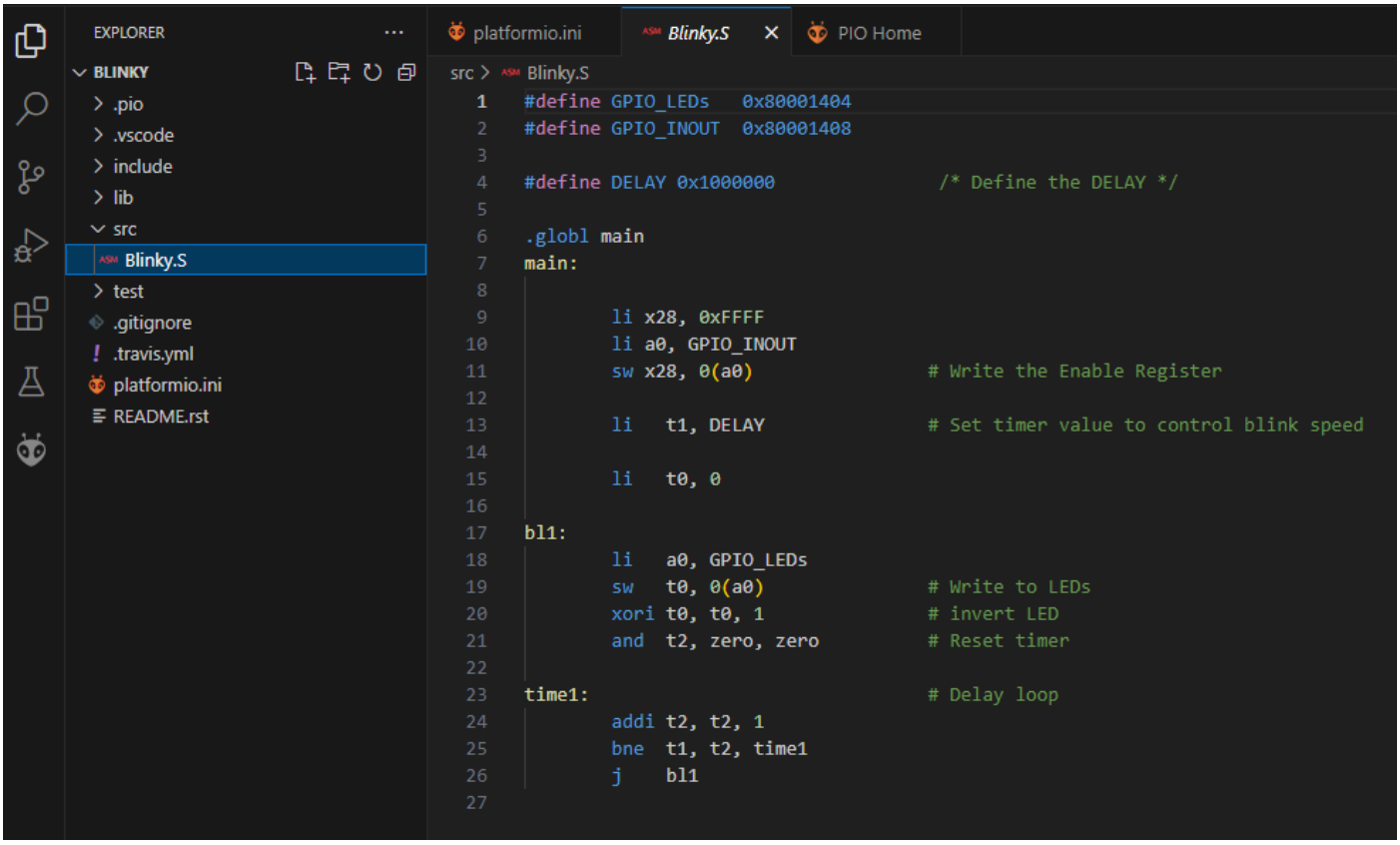
L'obiettivo e' andare ad abilitare l'attivazione dei LEDs sulla board e creare un programma che faccia lampeggiare un LED. Inizialmente dobbiamo abilitare il LED che vogliamo lampeggi, abilitando il relativo three state buffer attraverso il registro GPIO_INOUT. Successivamente, possiamo far lampeggiare il LED abilitato cambiando lo stato del LED nel registro GPIO_LEDS. Lo schema dell'abilitazione e cambiamento di stato del LED numero zero e' il seguente:



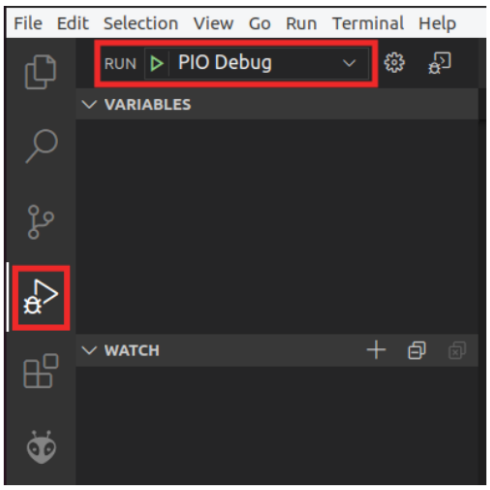
STEP1) Aprire il progetto "Blinky" cliccando su File->OpenFolder nel menu' in alto a sinistra di Visual Studio Codeselezionare la cartella "Blinky" e cliccare sul pulsante "Select Folder".



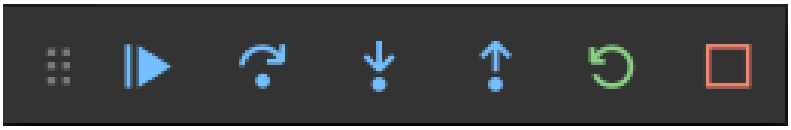
STEP2) Cliccare sul pulsante Explorer eclliccare sul file “Blinky.S”



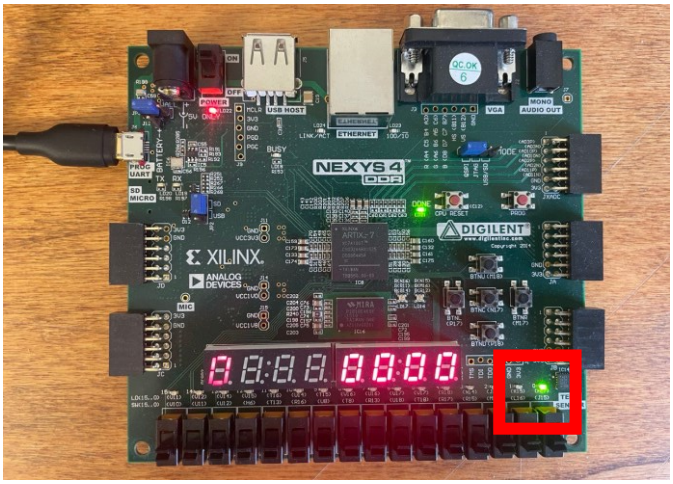
STEP3) Compilare il programma cliccando sul pulsante “Run and Debug” . Una volta terminata la compilazione, e’ possibile eseguire il debugging cliccando sul pulsante RUN PIO Debug .



STEP4) Controllare l’esecuzione attraverso la debugging toolbar. Cliccando una volta sul tasto , il debugging partira’ e si fermera’ alla prima istruzione (breakpoint automatico). Cliccando nuovamente sul tasto , partira’ l’esecuzione delle istruzioni.



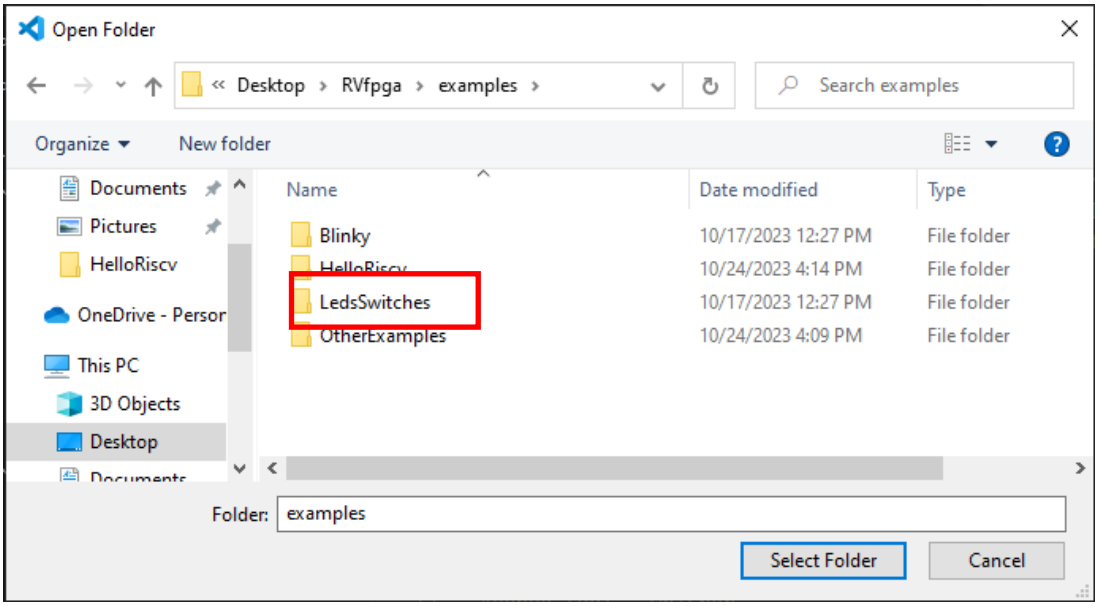
STEP5) Controllare che il led numero 0 stia lamepeggiando sulla board.



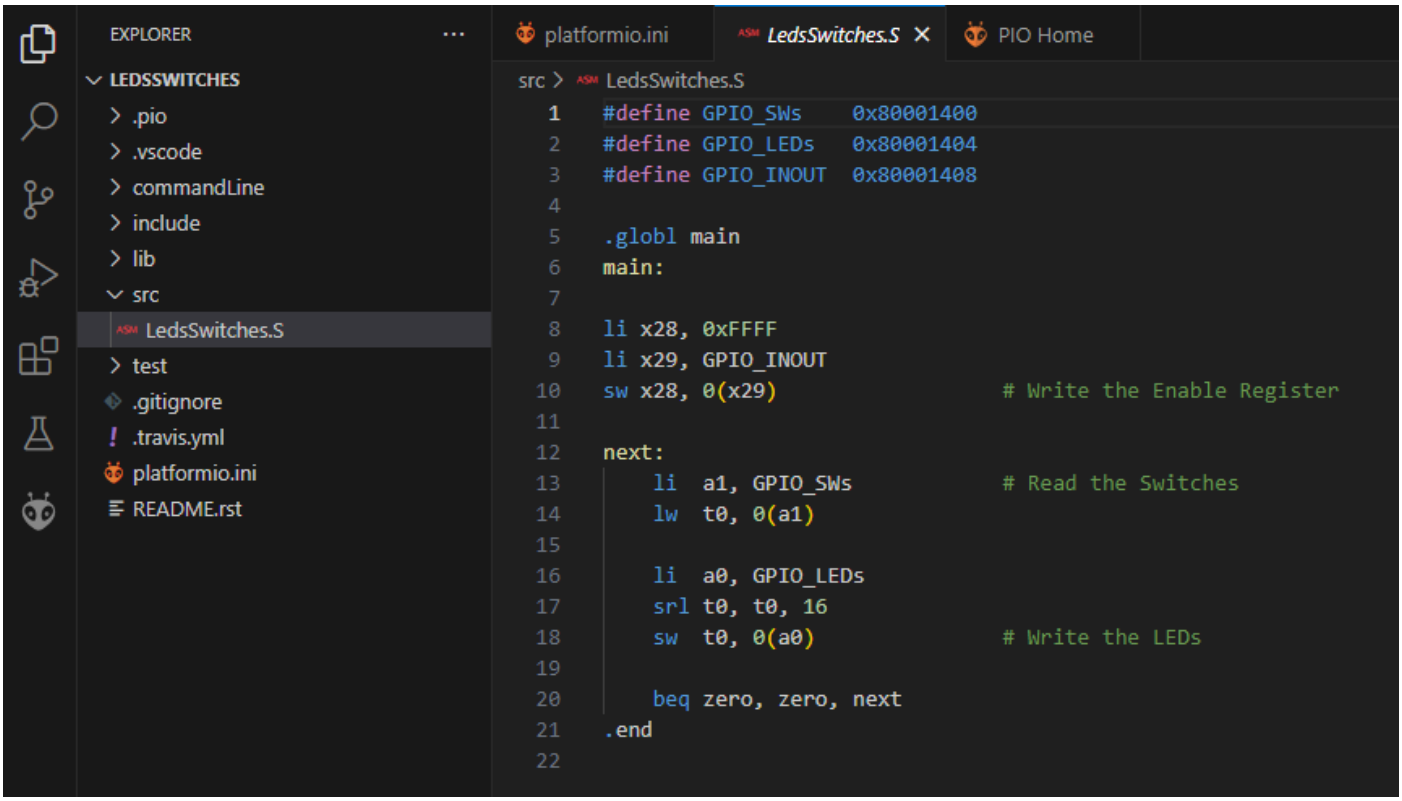
ESERCIZIO) Modificare il codice assembly del programma per in modo che venga acceso il LED numero 2

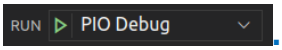
IV. Eseguire un programma in RISCV assembly che abiliti un interruttore ad accendere un led della board

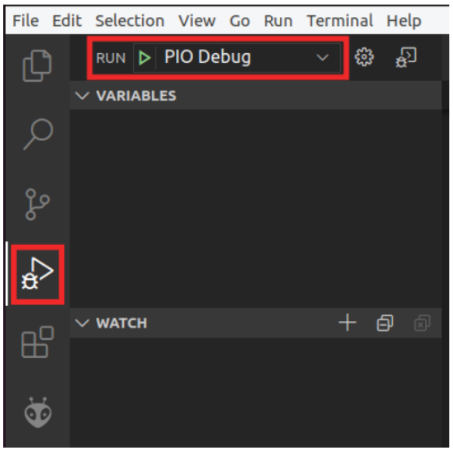
STEP1) Aprire il progetto “LedSwitches” cliccando su File->OpenFolder nel menu’ in alto a sinistra di Visual Studio Code selezionare la cartella “LedSwitches” e cliccare sul pulsante “Select Folder”.



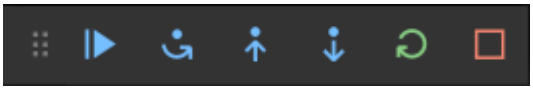
STEP2) Cliccare sul pulsante Explorer e cliccare sul file “LedsSwitches.S”



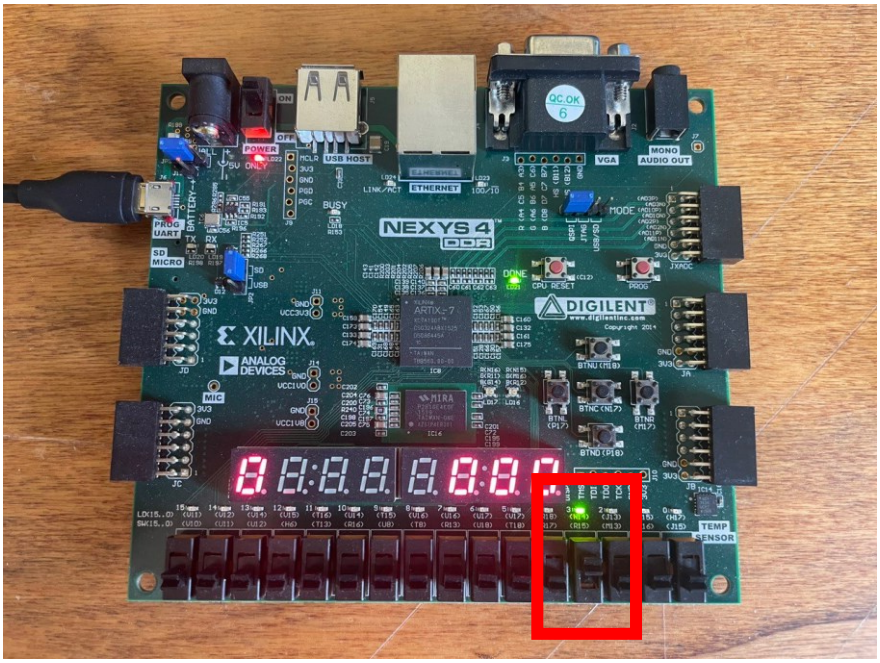
STEP3) Compilare il programma cliccando sul pulsante "Run and Debug"  . Una volta terminata la compilazione, e' possibileeseguireildebuggingcliccandosul pulsante  .



STEP4) Controllare l'esecuzione attraverso la debugging toolbar. Cliccando una volta sul tasto  , il debugging partira' e si fermera' alla prima istruzione (breakpoint automatico). Cliccando nuovamente sul tasto  , partira' l'esecuzione delle istruzioni.

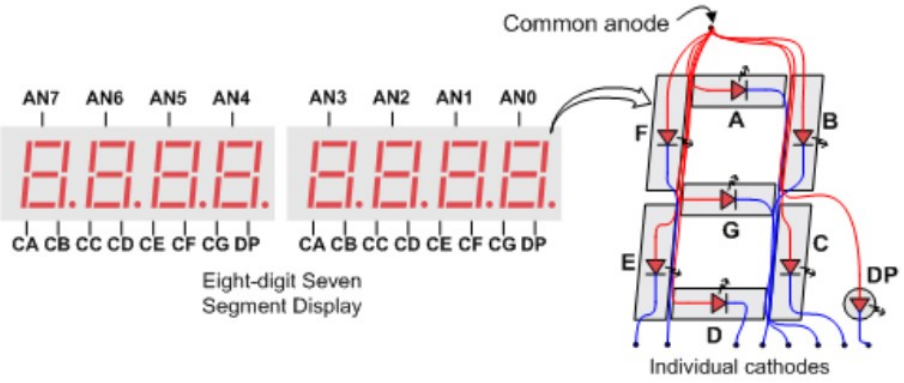


STEP5) Aziando gli interruttori  presenti sulla board, si accendera' il led corrispondente.



V. Eseguire un programma in RISCV assembly che stampi il numero 51 sul display della board

L'obiettivo e' quello di utilizzare il display ad 8 cifre presente sulla board. Ognuna delle 8 cifre sulla board e' composta da 7 segmenti, con un LED per ogni segmento. Ogni segmento e' definito con una label A-G. Ognuno dei 7 segmenti e' raggruppato in un unico circuito "anode". Essendoci 8 cifre a disposizione, abbiamo 8 anode marcati come AN0-AN7. Tutti i 7 segmenti A-G di tutti gli 8 anode (8 cifre) sono connessi su 7 segnali (CA-CG). Tutti questi segnali sono attivi bassi. Quindi, per esempio, per illimunare il segmento D della cifra 2 dobbiamo attivare (basso) l'anode AN2 e il segnale CD.



- **Struttura dell'implementazione Verilog del SegDisplay**

La connessione tra l'input/output del core SweRV e la board viene definita nel file "Rvfpga/src/rvfpganexys.xdc". Ognuno dei device I/O presente sulla board e' connesso ad uno specifico pin del chip FPGA. I segnali che connettono gli 8 anode per le 8 cifre sono definiti come AN[0-7], mentre i 7 segnali dei segmenti sono definiti come CA,CB,CC,CD,CE,CF,CG (vedi foto sotto).

```
60  ##7 segment display
61  set_property -dict { PACKAGE_PIN T10   IOSTANDARD LVCMOS33 } [get_ports { CA }]; #IO_L24N_T3_A00_D16_14 Sch=ca
62  set_property -dict { PACKAGE_PIN R10   IOSTANDARD LVCMOS33 } [get_ports { CB }]; #IO_25_14 Sch=cb
63  set_property -dict { PACKAGE_PIN K16   IOSTANDARD LVCMOS33 } [get_ports { CC }]; #IO_25_15 Sch=cc
64  set_property -dict { PACKAGE_PIN K13   IOSTANDARD LVCMOS33 } [get_ports { CD }]; #IO_L17P_T2_A26_15 Sch=cd
65  set_property -dict { PACKAGE_PIN P15   IOSTANDARD LVCMOS33 } [get_ports { CE }]; #IO_L13P_T2_MRCC_14 Sch=ce
66  set_property -dict { PACKAGE_PIN T11   IOSTANDARD LVCMOS33 } [get_ports { CF }]; #IO_L19P_T3_A10_D26_14 Sch=cf
67  set_property -dict { PACKAGE_PIN L18   IOSTANDARD LVCMOS33 } [get_ports { CG }]; #IO_L4P_T0_D04_14 Sch=cg
68  #set_property -dict { PACKAGE_PIN H15   IOSTANDARD LVCMOS33 } [get_ports { DP }]; #IO_L19N_T3_A21_VREF_15 Sch=dp
69  set_property -dict { PACKAGE_PIN J17   IOSTANDARD LVCMOS33 } [get_ports { AN[0] }]; #IO_L23P_T3_FOE_B_15 Sch=an[0]
70  set_property -dict { PACKAGE_PIN J18   IOSTANDARD LVCMOS33 } [get_ports { AN[1] }]; #IO_L23N_T3_FWE_B_15 Sch=an[1]
71  set_property -dict { PACKAGE_PIN T9    IOSTANDARD LVCMOS33 } [get_ports { AN[2] }]; #IO_L24P_T3_A01_D17_14 Sch=an[2]
72  set_property -dict { PACKAGE_PIN J14   IOSTANDARD LVCMOS33 } [get_ports { AN[3] }]; #IO_L19P_T3_A22_15 Sch=an[3]
73  set_property -dict { PACKAGE_PIN P14   IOSTANDARD LVCMOS33 } [get_ports { AN[4] }]; #IO_L8N_T1_D12_14 Sch=an[4]
74  set_property -dict { PACKAGE_PIN T14   IOSTANDARD LVCMOS33 } [get_ports { AN[5] }]; #IO_L14P_T2_SRCC_14 Sch=an[5]
75  set_property -dict { PACKAGE_PIN K2    IOSTANDARD LVCMOS33 } [get_ports { AN[6] }]; #IO_L23P_T3_35 Sch=an[6]
76  set_property -dict { PACKAGE_PIN U13   IOSTANDARD LVCMOS33 } [get_ports { AN[7] }]; #IO_L23N_T3_A02_D18_14 Sch=an[7]
```

I segnali delle 8 cifre e dei 7 segmenti sono poi specificati nel top-module file "Rvfpga/src/rvfpganexys.sv" per la connessione con lo swervolf_core (vedi foto sotto).

```
25  module rvfpganexys
26    #(parameter bootrom_file  = "boot_main.mem")
27     (input wire         clk,
28      input wire         rstn,
29      output wire [12:0] ddram_a,
30      output wire [2:0]  ddram_ba,
31      output wire        ddram_ras_n,
32      output wire        ddram_cas_n,
33      output wire        ddram_we_n,
34      output wire        ddram_cs_n,
35      output wire [1:0]  ddram_dm,
36      inout wire [15:0]  ddram_dq,
37      inout wire [1:0]  ddram_dqs_p,
38      inout wire [1:0]  ddram_dqs_n,
39      output wire        ddram_clk_p,
40      output wire        ddram_clk_n,
41      output wire        ddram_cke,
42      output wire        ddram_odt,
43      output wire        o_flash_cs_n,
44      output wire        o_flash_mosi,
45      input wire         i_flash_miso,
46      input wire         i_uart_rx,
47      output wire        o_uart_tx,
48      inout wire [15:0]  i_sw,
49      output reg [15:0]  o_led,
50      output reg [7:0]   AN,
51      output reg         CA, CB, CC, CD, CE, CF, CG,
```

```
256       .i_ram_init_error (litedram_init_error),
257       .io_data          ({i_sw[15:0],gpio_out[15:0]}),
258       .AN (AN),
259       .Digits_Bits ({CA,CB,CC,CD,CE,CF,CG}),
260       .o_accel_sclk     (accel_sclk),
```

Infine, i due segnali AN e Digits_Bits vengono inseriti nel System Controller ("swervolf_syscon") per la connessione con lo swervolf_core all'interno del file swervolf_core.v

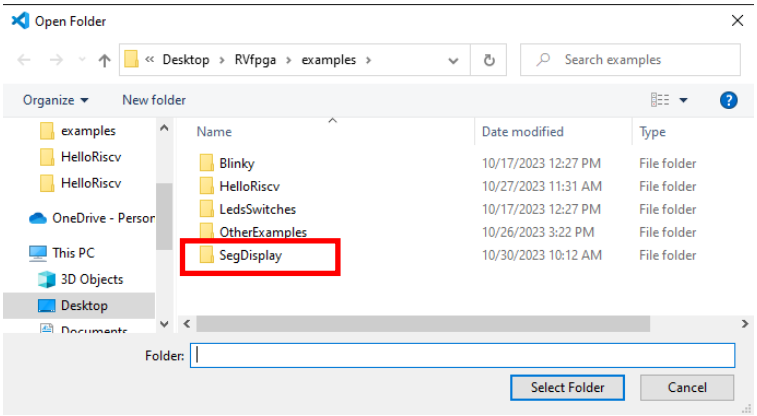
```
215     swervolf_syscon
216       #(.clk_freq_hz (clk_freq_hz))
217     syscon
218       (.i_clk            (clk),
219        .i_rst            (wb_rst),
220        .gpio_irq         (gpio_irq),
221        .ptc_irq          (ptc_irq),
222        .o_timer_irq      (timer_irq),
223        .o_sw_irq3        (sw_irq3),
224        .o_sw_irq4        (sw_irq4),
225        .i_ram_init_done  (i_ram_init_done),
226        .i_ram_init_error (i_ram_init_error),
227        .o_nmi_vec        (nmi_vec),
228        .o_nmi_int        (nmi_int),
229        .i_wb_adr         (wb_m2s_sys_adr[5:0]),
230        .i_wb_dat         (wb_m2s_sys_dat),
231        .i_wb_sel         (wb_m2s_sys_sel),
232        .i_wb_we          (wb_m2s_sys_we),
233        .i_wb_cyc         (wb_m2s_sys_cyc),
234        .i_wb_stb         (wb_m2s_sys_stb),
235        .o_wb_rdt         (wb_s2m_sys_dat),
236        .o_wb_ack         (wb_s2m_sys_ack),
237        .AN (AN),
238        .Digits_Bits (Digits_Bits));
239
```

Andiamo ad analizzare in dettaglio il modulo System Controller situato in Rvfpga/src/SweRVolfSoC/Peripherals/SystemController/swervolf_syscon.v. Il modulo "SevSegDiplays_Controller" viene instanziato alle linee 276-288

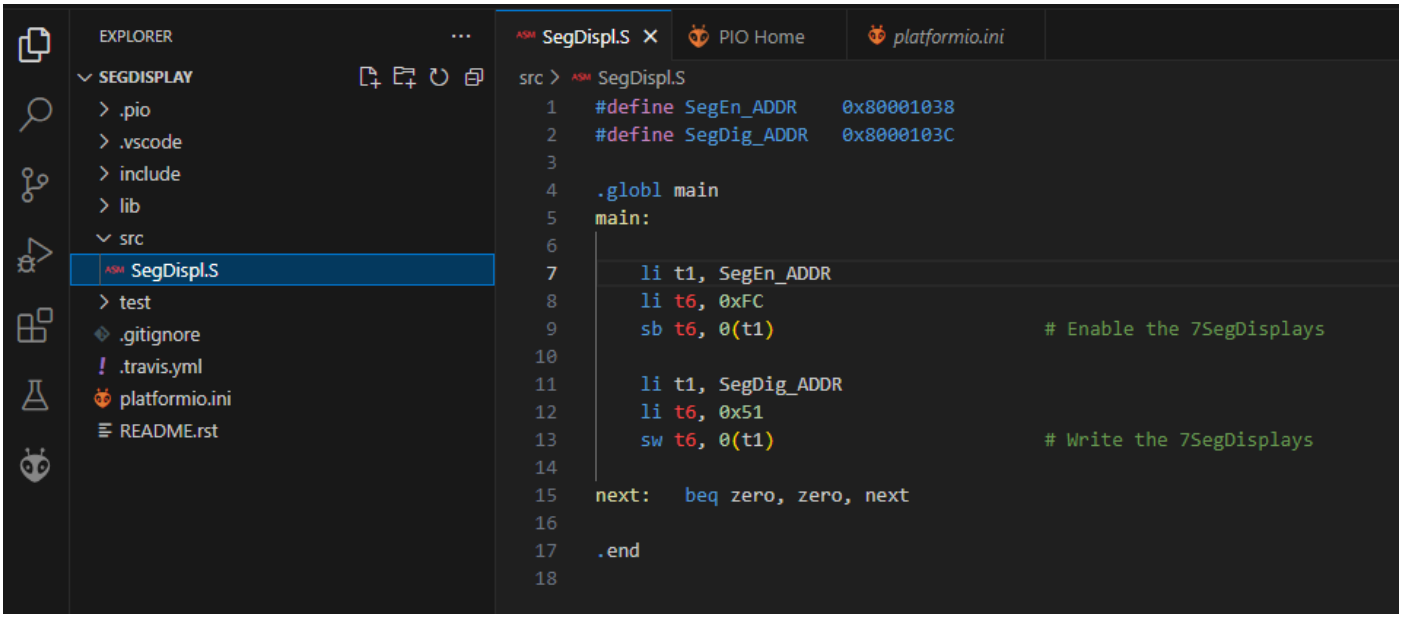
```
276    // Eight-Digit 7 Segment Displays
277
278      reg  [ 7:0]  Enables_Reg;
279      reg  [31:0]  Digits_Reg;
280
281      SevSegDisplays_Controller SegDispl_Ctr(
282        .clk                (i_clk),
283        .rst_n              (i_rst),
284        .Enables_Reg        (Enables_Reg),
285        .Digits_Reg         (Digits_Reg),
286        .AN                 (AN),
287        .Digits_Bits        (Digits_Bits)
288      );
289
290  endmodule
```

Il modulo SevSegDisplays_Controller riceve, oltre ai segnali di clock e reset, due segnali di input "Enables_Reg" e "Digits_Reg", i quali sono due memory-mapped control registers. Mentre i segnali di output AN e Digits_Bits sono connessi con il display della board.

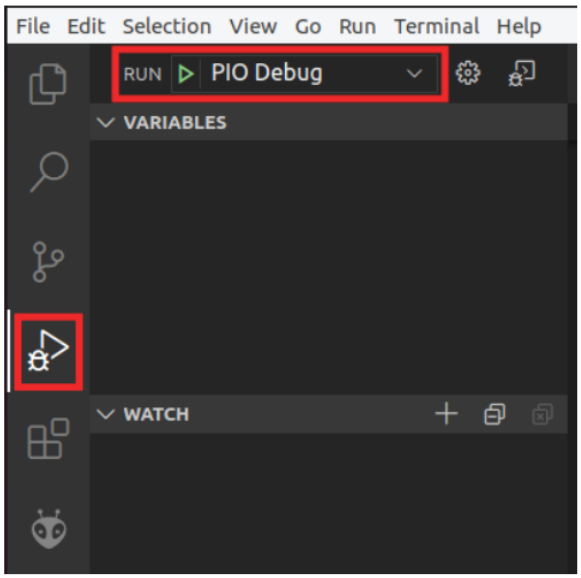
STEP1) Aprire il progetto "SegDisplay" cliccando su File->OpenFolder nel menu' in alto a sinistra di Visual Studio Code selezionare la cartella "SegDisplay" e cliccare sul pulsante "Select Folder".



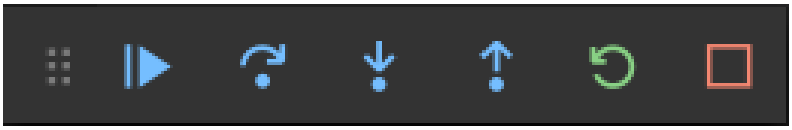
STEP2) Cliccare sul pulsante Explorer e cliccare sul file "SegDiplay.S"



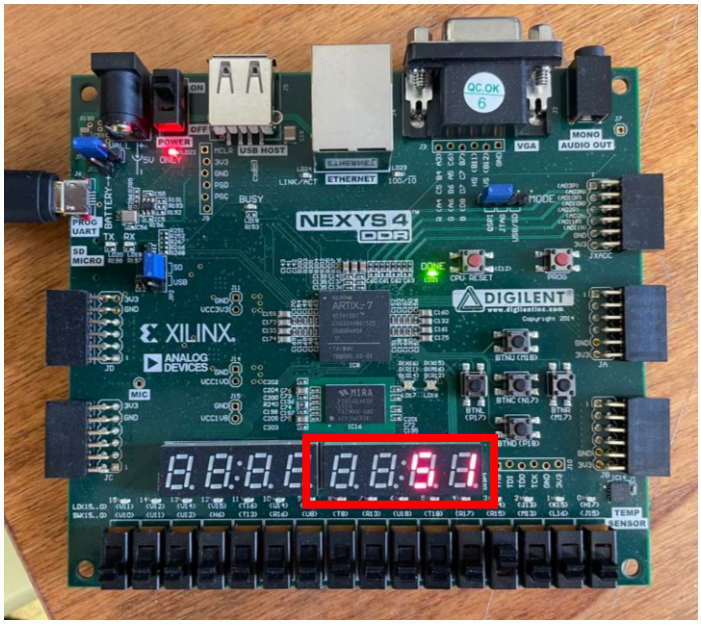
STEP3) Compilare il programma cliccando sul pulsante "Run and Debug" . Una volta terminata la compilazione, e' possibileeseguireildebuggingcliccandosul pulsante RUN PIO Debug .



STEP4) Controllare l'esecuzione attraverso la debugging toolbar. Cliccando una volta sul tasto , il debugging partira' e si fermera' alla prima istruzione (breakpoint automatico). Cliccando nuovamente sul tasto , partira' l'esecuzione delle istruzioni.



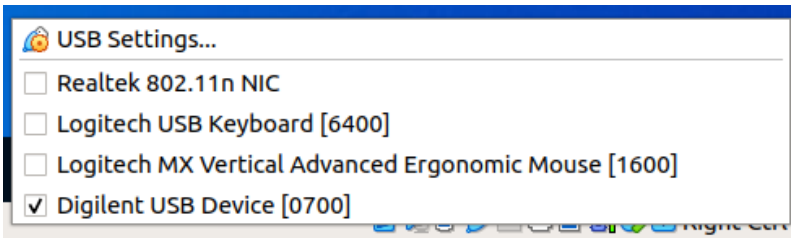
STEP5) Controllare che sul display della board appaia il numero 51



ESERCIZIO1) Modificare il codice assembly del programma in modo che venga stampato sul display il numero 1234. Suggerimento: bisogna abilitare le prime 4 cifre sul display per poter stampre l'intera cifra.

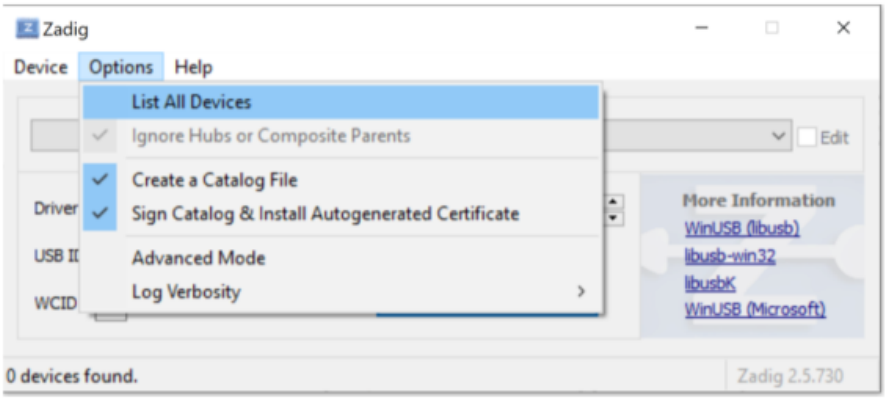
APPENDICE A: installare i driver in Windows per utilizzare il framework PlatformIO.

STEP1) Una volta avviata la board e la macchina virtuale, assicurarsi che la board sia connessa cliccando con il tasto destro del mouse sull'icona che si trova nella toolbar in basso destra. Nel menu successivo, spuntare l'opzione "Digilent USB Device"

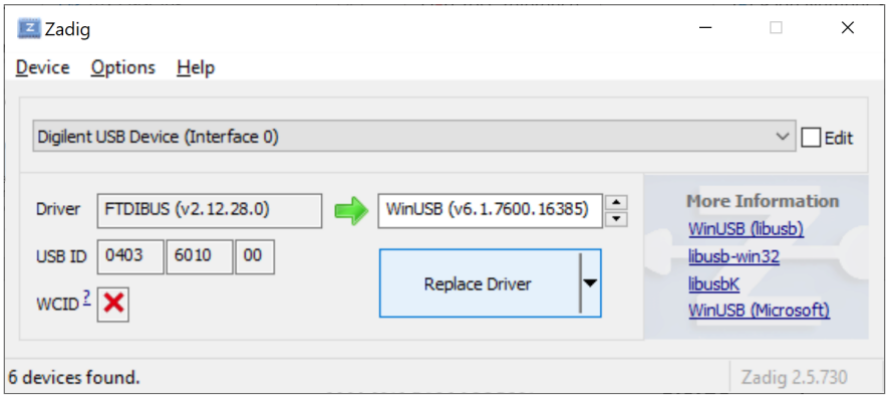


STEP2) Avviare il programma Zadig cliccando sull'icona presente sul Desktop della macchina virtuale.

STEP3) Visualizzare tutti i device cliccando su "Options->List All Devices"



STEP4) Cliccando sul menu a tendina, selezionare il device Digilent USB Device (Interface 0) e cliccare sul pulsante "Replace Driver"



STEP5) Attendere il completamento dell'installazione e poi tornare in Visual Studio Code.

ESERCIZI in LABORATORIO – MODULI SINCRONI MUTUAMENTE ASINCRONI (ESERCITAZIONE/LAB #09)

OBIETTIVI

1) **Capire il modello produttore-consumatore e la sincronizzazione fra moduli con clock diverso**
2) **Analizzare un semplice esempio di moduli produttore-consumatore**

1) Capire il modello produttore-consumatore e la sincronizzazione fra moduli con clock diverso

Siano date due moduli sincroni (ognuna con un proprio segnale di clock con periodo in generale diverso) come rappresentato in Figura 1, una denominata Produttore e l'altra Consumatore con l'idea che la prima passi dei dati alla seconda tramite il bus "data".

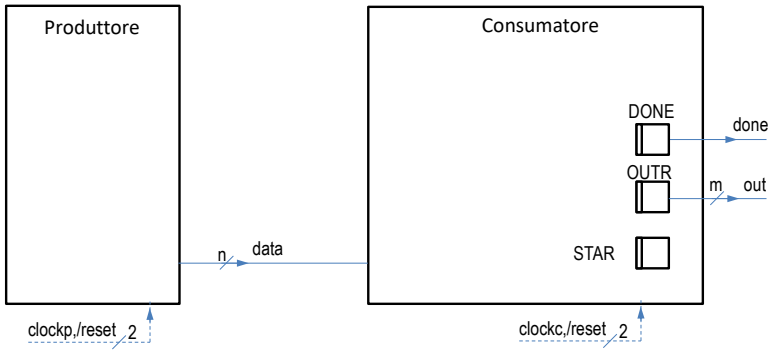
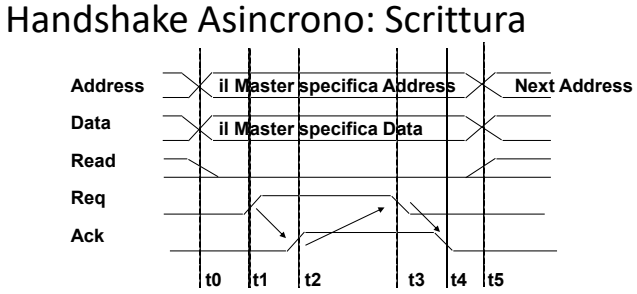


FIGURA 1 – PRODUTTORE E CONSUMATORE LOCALMENTE SINCRONIZZATI MA GLOBALMENTE ASINCRONI.

Essendo pero' i due moduli senza un clock comune, si rende necessario stabilire una comunicazione asincrona: ciò può essere fatto ricorrendo ad un protocollo di handshake del tutto simile a quello analizzato a proposito dei bus e richiamato in Figura 2 nel caso di scrittura da master a slave. Il ruolo dei segnali Req e Ack è qui giocato dai segnali rfd e /dav.



- t0 : Il Master ha ottenuto il controllo e specifica l'indirizzo, la direzione e dati; attende poi un certo intervallo di tempo affinche' lo Slave capisca di essere coinvolto in quel trasferimento...
- t1: Il Master attiva la linea "Request"
- t2: Lo Slave attiva la linea "Ack", per indicare di aver ricevuto il dato → ha finito
- t3: Il Master rilascia "Req" → ha capito che lo Slave ha finito
- t4: Lo Slave rilascia "Ack" → ha capito che il Master ha capito

FIGURA 2 - HANDSHAKE ASINCRONO IN CASO DI SCRITTURA FRA MASTER E SLAVE.

Se il consumatore indica una richiesta di dati tramite il filo Request-For-Data (ovvero rfd==1), il produttore mette a disposizione tali dati sul bus "data" e ne indica la disponibilita' tramite il filo DAta-Valid (ovvero /dav==0). Lo schema viene così ad essere quello indicato in Figura 3.

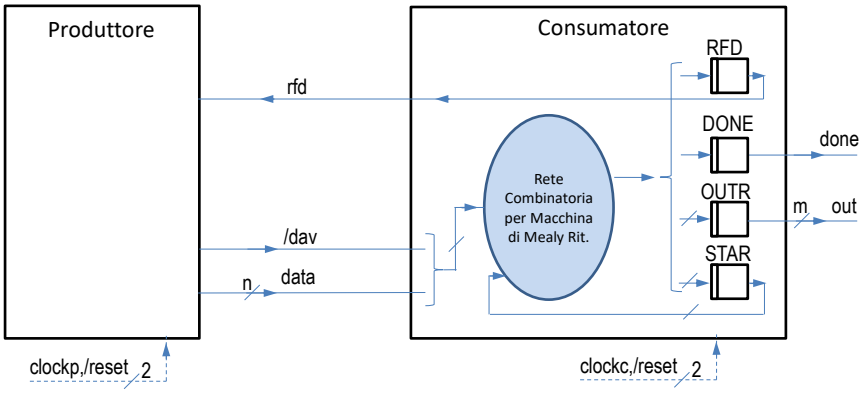


FIGURA 3 – DETTAGLIO DEL MODULO "CONSUMATORE".

Sull'uscita "out" il Consumatore fornirà un ulteriore dato da esso elaborato e la cui validità è indicata dal filo "done", che varrà 1 se l'uscita "out" è pronta mentre varrà 0 altrimenti. Il consumatore può essere implementato come una macchina sequenziale sincronizzata, ad es. ricorrendo al modello di Mealy-Ritardato il cui stato interno sarà mantenuto nel registro di stato "STAR". Le uscite sono mantenute nei registri DONE, OUTR ed inoltre il registro RFD manterrà il segnale di handshake con il produttore.

Per governare l'handshake dal lato Produttore assumiamo per il momento che il Produttore si comporti nel seguente modo:

- Prima di tutto il produttore si attiva soltanto se c'è una richiesta di un dato (rfd==1)
- Se rfd==1, il Produttore elabora e genera qualcosa di significativo sul bus "data" ed infine attiva /dav (/dav==0) al solito rispettando i tempi tsetup e thold rispettivamente prima e dopo l'istante di validità dell'informzione su "data".
- Quando il segnale rfd si disattiva (rfd==0), anche il segnale /dav viene disattivato (/dav==1): ciò può avvenire quasi istantaneamente se clockp>>clockc ovvero, se il Produttore è molto più veloce del Consumatore, anche dopo qualche ciclo del Consumatore (il Produttore è molto più lento del Consumatore (clockp<<clockc)).

Il Consumatore si comporterà invece nel seguente modo:

- Provvede ad attivare rfd per prima cosa (rfd==1) ed attende l'attivazione di /dav (/dav==0) prima di iniziare il proprio lavoro.
- Se /dav==0, il Consumatore elabora e genera qualcosa di significativo sull'uscita "out" ed infine attiva "done" (done==1) e disattiva rfd (rfd==0) per indicare al Produttore che il dato e' stato acquisito.
- Attende che il Produttore abbia capito che il dato e' stato acquisito tramite la disattivazione di /dav (/dav==1) e ritorna quindi allo stato iniziale.

Per quanto detto un diagramma a stati che descrive il comportamento del Consumatore può essere quello di Figura 4.

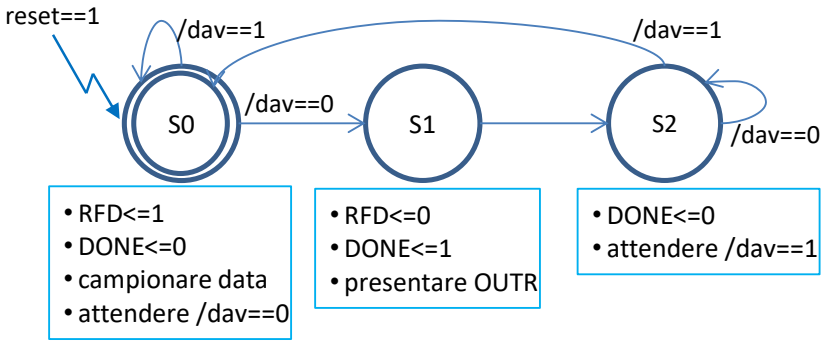


FIGURA 4 – DIAGRAMMA A STATI DEL MODULO "CONSUMATORE".

Possiamo formalizzare questo diagramma usando il linguaggio VERILOG:

```
module Consumatore(dav_,data, rfd,out,done, clock,reset_);
 input        clock, reset_;
 output [1:0] out;
 output       rfd,done;
 input        dav_;
 input[n-1:0] data;
 reg [1:0]  OUTR;      assign    out=OUTR;
 reg        RFD,DONE;  assign    rfd=RFD, done=DONE;
 reg [1:0]  STAR;      parameter S0=0, S1=1, S2=2;

 function [m-1:0] elab;
   input [n-1:0] data;
   elab = .........;
 endfunction

 always @(reset_==0) begin DONE<=0; RFD<=1; STAR<=S0; end

 always @(posedge clock) if (reset_==1)  #3
  casex (STAR)
   S0:     begin RFD<=1;                              STAR<=(dav_==1)?S0:S1; end
   S1:     begin RFD<=0; OUTR<=elab(data);DONE<=1; STAR<=S2; end
   S2:     begin DONE<=0;                             STAR<=(dav_==0)?S2:S0; end
  endcase
endmodule
```

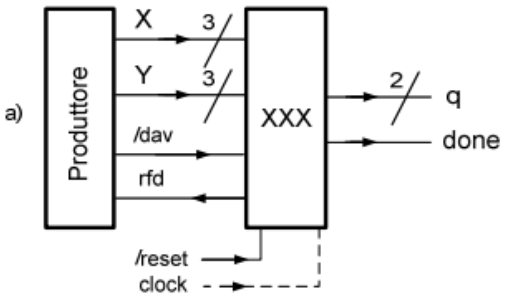
Nel precedente codice VERILOG è stata evidenziata la parte relativa alla elaborazione dei dati letti (data) e la produzione del relativo risultato. Le restanti variabili codificano l'handshake tramite il modello asincrono "/dav-rfd" (anziché "req-ack") e l'implementazione della macchina a stati finiti secondo il modello di Mealy-Ritardato.

2) Analizzare un semplice esempio di modulo produttore-consumatore

Es. n.4 (adattato) del 21/12/2015: https://arcal.diism.unisi.it/esercizi1/c1151221-sol.pdf

Il modulo XXX (Fig. a) preleva dal Produttore i due numeri naturali X e Y e li utilizza come le rappresentazioni (in complemento a due) di due numeri interi x e y. Interpreta x e y come le coordinate di un punto nel piano cartesiano ed emette tramite l'uscita q il numero d'ordine (0, 1, 2, 3) del quadrante a cui appartiene il punto (Fig. b) tenendo done ad 1 per un ciclo di clock.

a)



b)

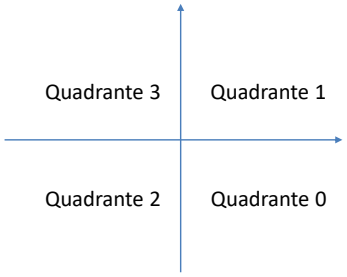
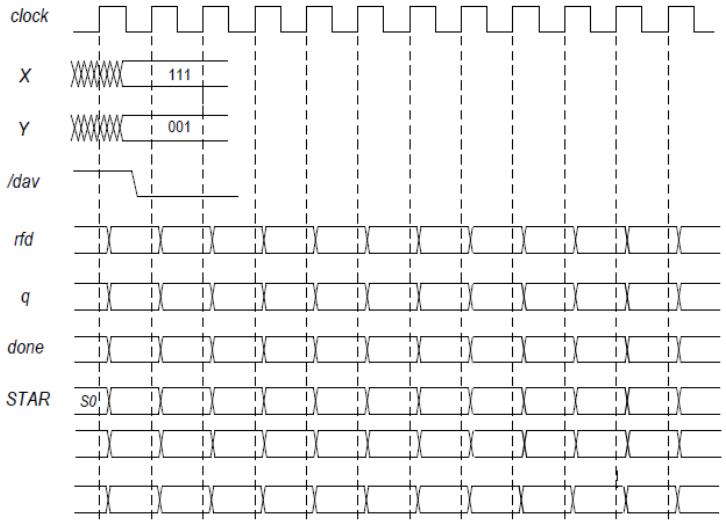


TABELLA c)

x	y	X	Y	q
+1	+1	001	001	1
+1	-1			
-1	-1			
-1	+1			
+0	+0	000	000	1
+0	+1			
+0	-1			
+1	+0			
-1	+0			

Si specifichino i numeri d'ordine dei quadranti in modo da ottimizzare la rete combinatoria che produce *q* e si riportino tali numeri d'ordine nella Fig. b. Sempre in questa ottica, si considerino i semiassi come opportunamente appartenenti ai quadranti. Per chiarirsi le idee è utile completare preliminarmente la seguente tabella c.

Si descriva e si sintetizzi il modulo XXX e se ne tracci l'evoluzione nell'ipotesi che il Produttore fornisca le rappresentazioni di *(x,y)*= (-1,+1), (0,+2) indicando chiaramente nel grafico tali rappresentazioni.



Per prima cosa conviene completare le righe più ovvie:

x	y	X	Y	q
+1	+1	001	001	1
+1	-1	001	111	0
-1	-1	111	111	2
-1	+1	111	001	3
+0	+0	000	000	1
+0	+1			
+0	-1			
+1	+0			
-1	+0			

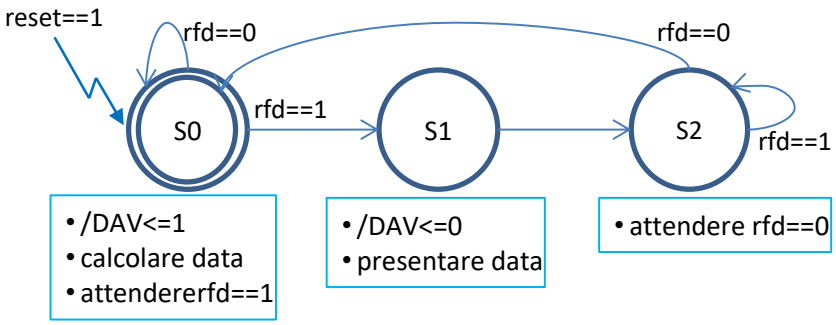
Si puo' osservare a questo punto che si potrebbe scegliere una funzione molto semplice che leghi X e Y con q, infatti usando i bit piu'significativi X[2] e Y[2] ottengo sempre un numero diverso da 0 a 3, quindi potrei usare la seguente mappa

X[2]	Y[2]	q={q1,q0}
0	0	1 =01
0	1	0 =00
1	1	2 =10
1	0	3 =11

Si puo' quindi facilmente osservare che q1 coincide con X[2], mente q0 corrisponde a NOT(Y[2]). Conviene pertanto completare la codifica dei punti sugli assi facendo un modo che le rispettive codifiche abbiano ancora il bit piu' significativo tale da mappare i punti sui quandranti che non cambino la precedente scelta {X[2],Y[2]}→{q1,q0}

x	y	X	Y	q
+1	+1	001	001	1
+1	-1	001	111	0
-1	-1	111	111	2
-1	+1	111	001	3
+0	+0	000	000	1
+0	+1	000	001	1
+0	-1	000	111	0
+1	+0	001	000	1
-1	+0	111	000	3

Per esercizio si può sintetizzare il codice VERILOG anche del **produttore** (è del tutto analogo al consumatore coi dovuti cambiamenti di nome e di ruoli delle variabili).



Per il testbench viene dato il seguente codice:

```
module testbench;
 reg reset_; initial begin reset_=0;  #1 reset_=1; #350; $stop; end
 reg clock1 ; initial clock1 =0; always #5 clock1 <=(!clock1);
 reg clock2 ; initial clock2 =0; always #2 clock2 <=(!clock2);
 wire[2:0] X,Y;
 wire rfd, dav_;
 wire[1:0]out; wire done;
 wire[1:0] STAR=Xxx.STAR;
 Consumatore Xxx(dav_,X,Y,  rfd,out,done,  clock1,reset_);
 Produttore  PRO(rfd,       dav_,X,Y,      clock2,reset_);
endmodule
```

Quindi un possibile codice VERILOG per il Produttore è dato da:

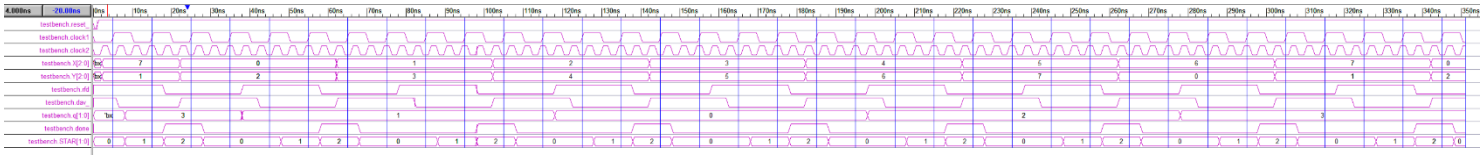
```
module Produttore(rfd, dav_,X,Y, clock,reset_);
 input         rfd, clock,reset_;
 output        dav_;
 output [2:0]  X,Y;
 reg DAV_;    assign dav_=DAV_;
 reg [2:0]    APP1_X,APP1_Y,APP2_X,APP2_Y; assign X=APP1_X, Y=APP1_Y;
 reg [1:0] STAR;      parameter S0=0, S1=1, S2=2;

 always @(reset_==0) begin APP2_X<='B111; APP2_Y<=001; DAV_<=1; STAR<=S0; end

 always @(posedge clock) if (reset_==1)  #0.2
  casex (STAR)
   S0: begin DAV_=1; APP1_X<=APP2_X; APP1_Y<=APP2_Y;     STAR<=(rfd==1)?S1:S0; end
   S1: begin DAV_=0; APP2_X<=APP1_X+1; APP2_Y<=APP1_Y+1; STAR<=S2; end
   S2: begin STAR<=(rfd==1)?S2:S0;end
  endcase
endmodule
```

Per il Consumatore XXX:

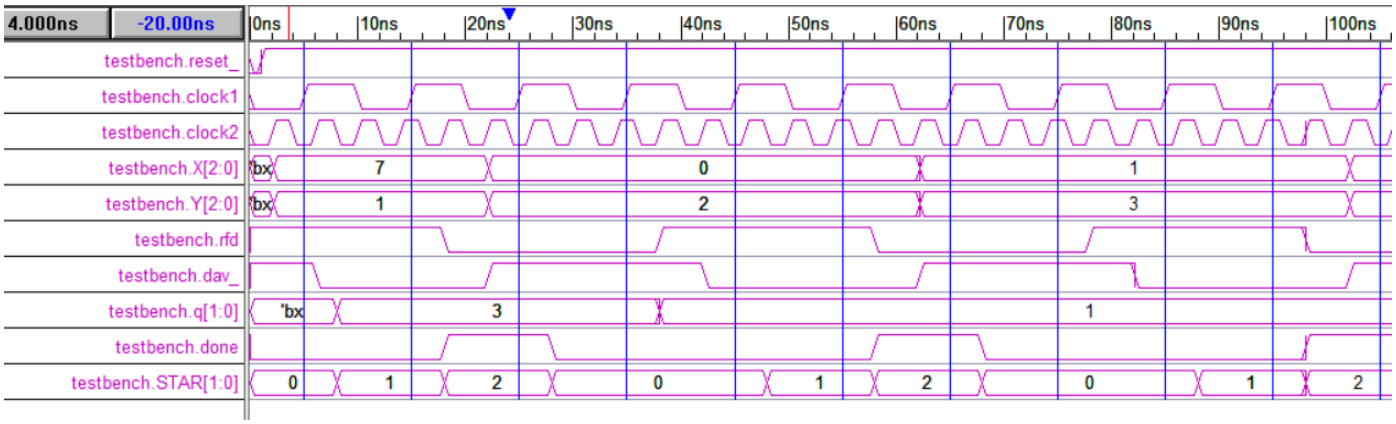
```
module Consumatore(dav_,X,Y, rfd,out,done, clock,reset_);
 input        clock, reset_;
 output [1:0] out;
 output       rfd,done;
 input        dav_;
 input  [2:0] X,Y;
 reg [1:0] OUTR;      assign    out=OUTR;
 reg       RFD,DONE;  assign    rfd=RFD, done=DONE;
 reg [1:0] STAR;      parameter S0=0, S1=1, S2=2;

 function [1:0] quadrant;
   input [2:0] x,y;
   quadrant = {x[2],~y[2]};
 endfunction

 always @(reset_==0) begin DONE<=0; RFD<=1; STAR<=S0; end

 always @(posedge clock) if (reset_==1)  #3
  casex (STAR)
   S0:     begin RFD<=1;                                STAR<=(dav_==1)?S0:S1; end
   S1:     begin RFD<=0; OUTR<=quadrant(X,Y);DONE<=1; STAR<=S2; end
   S2:     begin DONE<=0;                               STAR<=(dav_==0)?S2:S0; end
  endcase
endmodule
```

Si può verificare che non c'è dipendenza fra i clock dei due moduli Produttore e Consumatore (salvo il leggero ritardo posto al reset che deve eventualmente essere commisurato al periodo di clock). Nei seguenti testo sono stati provati i periodi di clock 4,2,10,16 per il Produttore mantenendo fisso il periodo di clock del Consumatore a 10 unità di tempo (1 unità di tempo= 1ns di default).

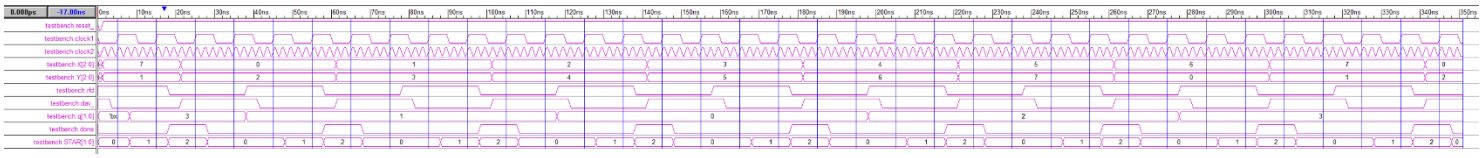
T(clockp)=4 – diagramma completo



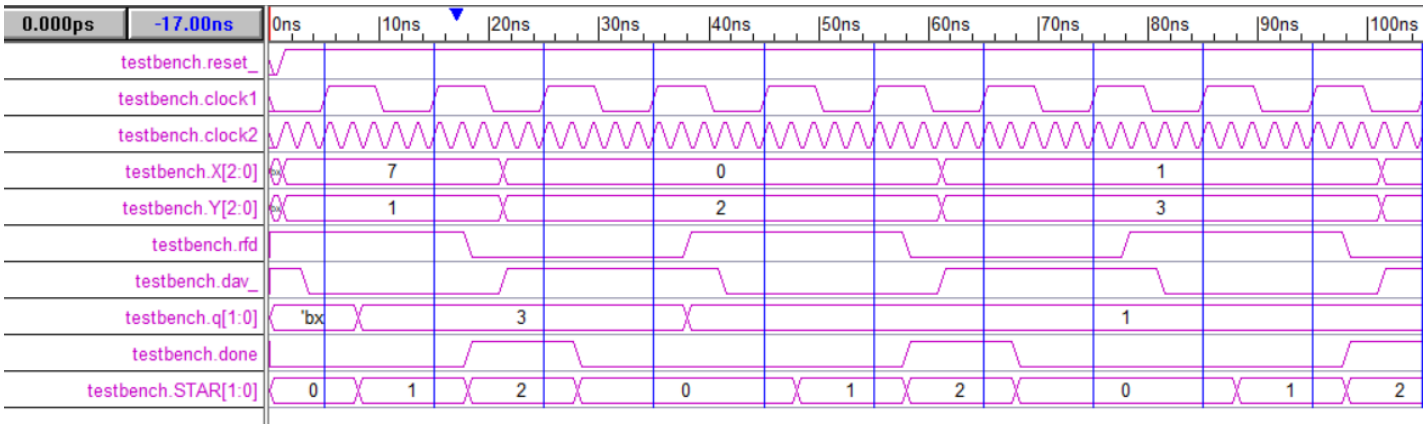
T(clockp)=4 – zoom



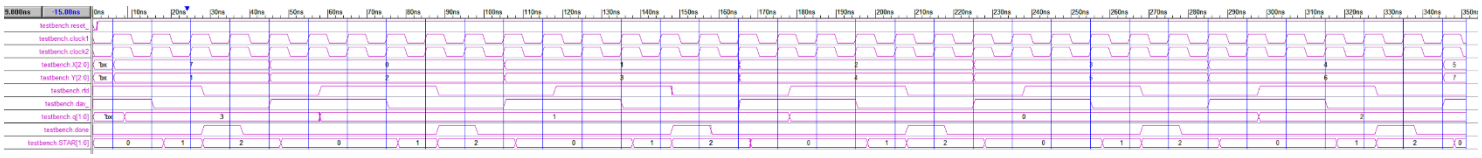
T(clockp)=2– diagramma completo



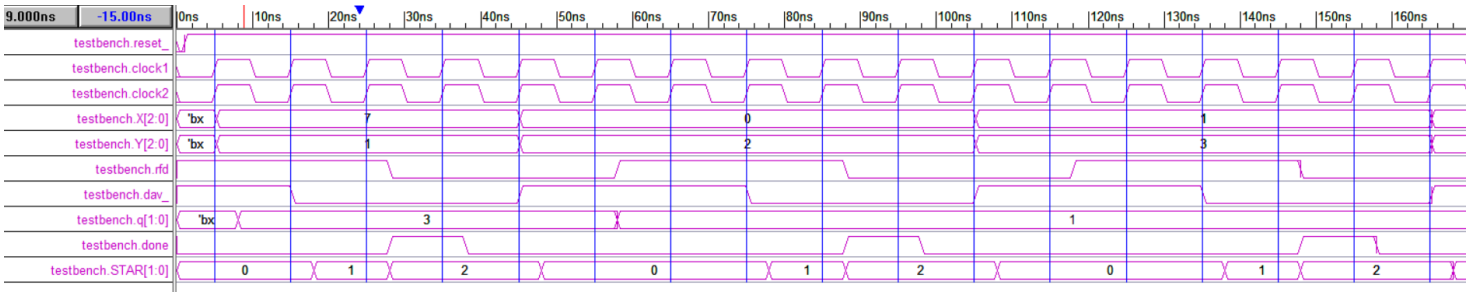
T(clockp)=2 – zoom



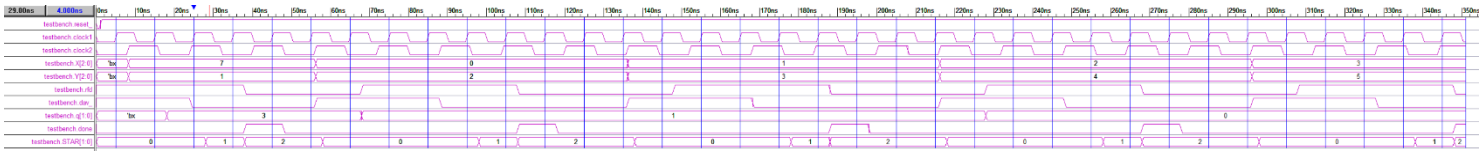
T(clockp)=10– diagramma completo



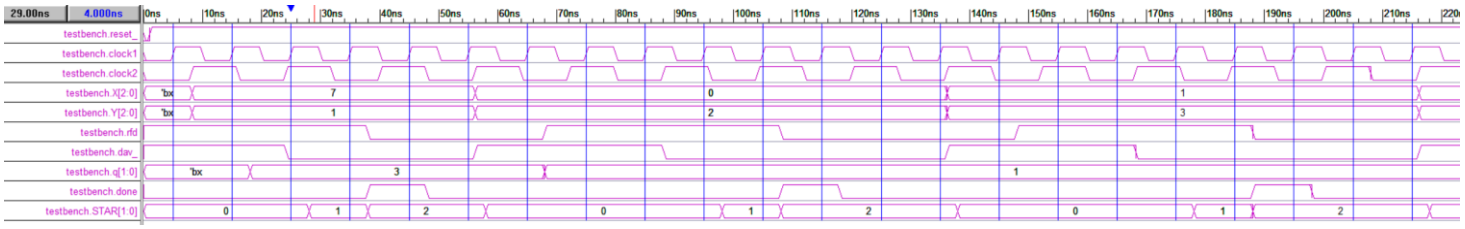
T(clockp)=10 – zoom



T(clockp)=16– diagramma completo



T(clockp)=16 – zoom



APPENDICE – STESSO MODULO SINTETIZZATO CON LE EQUAZIONI BOOLEANE

Tabella delle Transizioni (Tabella degli Stati)

TABELLA DELLE TRANSIZIONI

STATO ATTUALE \ x	0	1
S0	S1/01	S0/10
S1	S2/00	S2/00
S2	S2/00	S0/10
S3	--/--	--/--

Scelta della codifica degli Stati

STATO	CODIFICA y_1y_0
S0	00
S1	01
S2	11
X	10

Rappresentazione della Tabella degli Stati in Formato più comodo per la sintesi

OVVERO

y_1y_0 \ dav_	0	1
00	S1/01	S0/10
01	S2/00	S2/00
11	S2/00	S0/10
10	xx/xx	xx/xx

STATO SUCCES-SIVO/USCITA

OVVERO

y_1y_0 \ dav_	00	01
00	01/01	00/10
01	11/00	11/00
11	11/00	00/10
10	xx/xx	xx/xx

a1,a0,b1,b0

Sintesi della variabile ‘a’ (stato successivo o ingresso dello STATUS REGISTER -- STAR):

y_1y_0 \ dav_	0	1
00	0	0
01	1	1
11	1	0
10	X	X

a_1

```
a[1]=((~dav_)&y[0]) + ((~y[1])&y[0])
```

y_1y_0 \ dav_	0	1
00	1	0
01	1	1
11	1	0
10	X	X

a_0

```
a[0]= (~dav_) + ((~y[1])&y[0])
```

Sintesi della variabile ‘b’ (uscita successiva o ingresso dell’OUTPUT REGISTER -- OUTR):

y_1y_0 \ dav_	0	1
00	0	1
01	0	0
11	0	1
10	X	X

b_1

```
b[1]= (~dav_) & (~y[0])    //done
```

y_1y_0 \ dav_	0	1
00	1	0
01	0	0
11	0	0
10	X	X

b_0

```
b[0]= dav_ & ((~y[0])+y[1])    //rfd
```

La descrizione in VERILOG con le equazioni booleane risulta:

```
module Consumatore(dav_,X,Y, rfd,out,done, clock,reset_);
 input        clock, reset_;
 output [1:0] out;
 output       rfd,done;
 input        dav_;
 input  [2:0] X,Y;
 reg [1:0] OUTR;      assign    out=OUTR;
 reg       RFD,DONE;  assign    rfd=RFD, done=DONE;
 reg [1:0] STAR;      parameter S0=0, S1=1, S2=2;

 function [1:0] quadrant;
   input [2:0] x,y;
   quadrant = {x[2],~y[2]};
 endfunction
 always @(reset_==0) begin DONE<=0; RFD<=1; STAR<=S0; end

 wire [1:0] y, b, a;
 assign y=STAR;
 always @(posedge clock) if (reset_==1)  #3
  begin RFD<=b[0]; DONE<=b[1]; STAR<=a; OUTR<=quadrant(X,Y); end

 assign a[1]= ((~dav_)&y[0]) + ((~y[1])&y[0]);
 assign a[0]= (~dav_) + (~(y[1])&y[0]);
 assign b[1]= (~dav_) & (~y[0]); //done
 assign b[0]= dav_ & (~(y[0])+y[1]); //rfd

endmodule
```

Diagramma di Temporizzazione: (template)

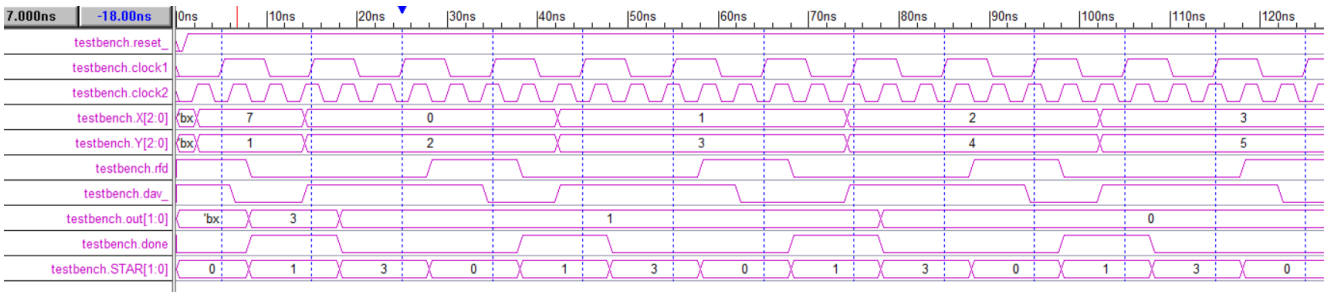


Diagramma di Temporizzazione: (equazioni booleane)