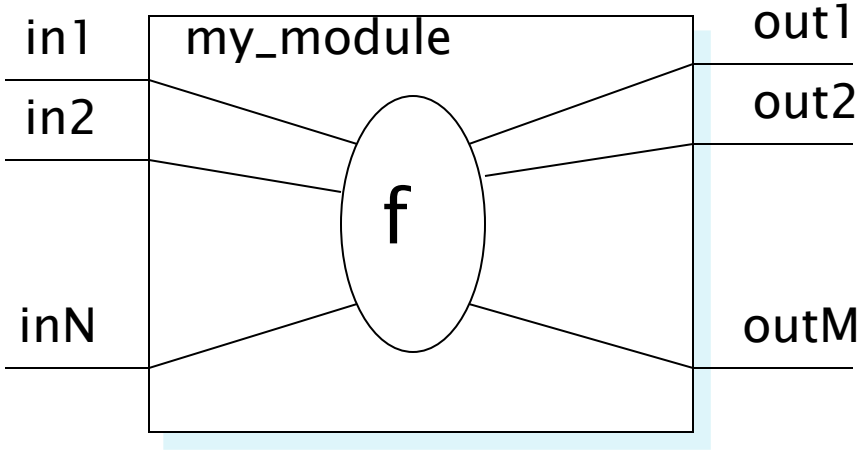
Basi di Verilog HDL

Descrizioni di un sistema digitale

Module



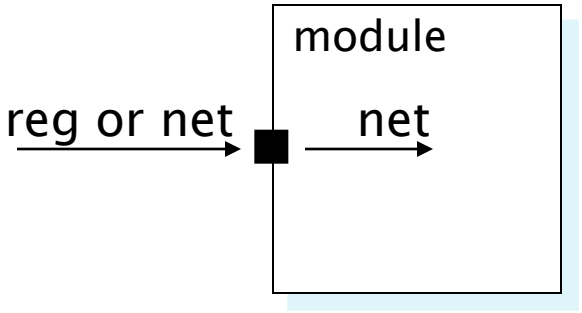

```
module my_module(out1, .., inN);
output out1, .., outM;
input in1, .., inN;

.. // declarations
.. // description of f (maybe
.. // sequential)

endmodule
```

Tutto quanto è scritto in Verilog deve risiedere all'interno di un "*module*"
Fanno eccezione solo eventuali direttive al compilatore

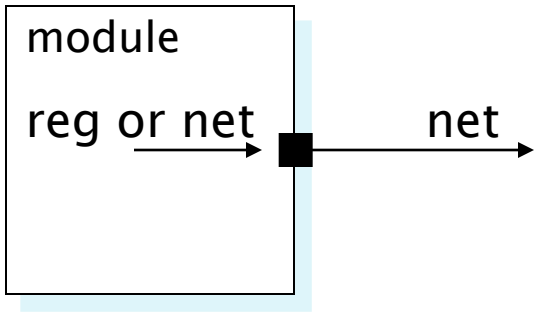
All'interno di un file Verilog possono esserci più "*module*"

Port Assignments

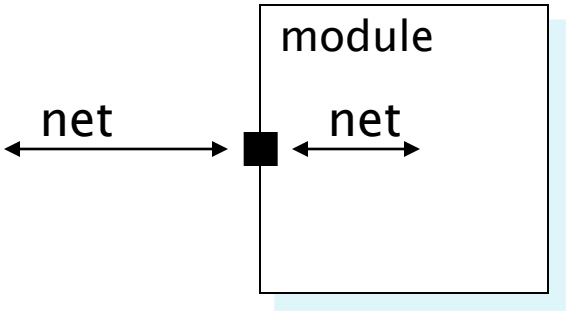
- Inputs



- Outputs



- Inouts



Assegnazioni

- In Verilog ci sono due modi fondamentali per effettuare un'assegnazione
  - Countinuous assignment

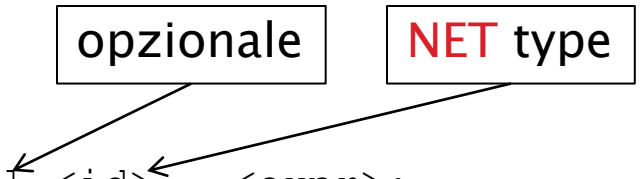
    assegnazione immediata

    descrive un funzionamento prettamente combinatorio
  - Procedural assignement

    assegnazione condizionata

    descrive un funzionamento per lo più sequenziale

Continuous Assignment



- **Sintassi**:

```
assign #del <id> = <expr>;
```

- **Dove e come deve essere scritto**:
  - dentro un “module”
  - fuori da una “procedure”
  - a sinistra dell’ “=“ deve esserci una NET
  - a destra dell’ “=“ può esserci NET, REGISTER, o una chiamata a funzione
- **Proprietà**:
  - sono eseguiti tutti in parallelo (sono concorrenti)
  - l’esecuzione è indipendente dall’ordine in cui sono stati scritti
  - sono sempre attivi se cambia uno degli operandi a destra, immediatamente viene aggiornato il risultato
  - I ritardi NON sono sintetizzati ma possono servire in simulazione per modellizzare ad esempio i ritardi dei gates.

Continuous Assignment (Es.)

```
wire adder_out = mut_out + out;
```

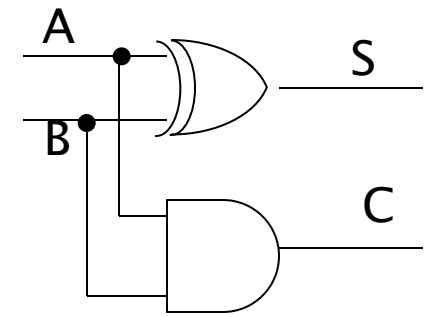
è equivalente a:

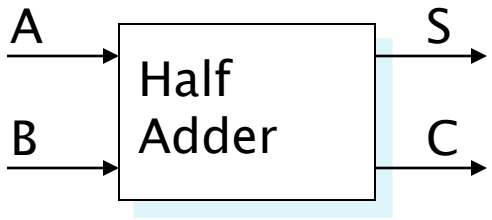
```
wire adder_out;
assign adder_out = mut_out + out;
```

oppure

```
assign #5 adder_out = mut_out + out;
```

Esempio: Half Adder






```
module half_adder(S, C, A, B);
output S, C;
input A, B;

wire S, C, A, B;

assign S = A ^ B;
assign C = A & B;

endmodule
```

Procedural Assignments

- Un modulo può contenere molteplici processi
- Ogni processo è eseguito in parallelo agli altri ma le istruzioni in esso contenute sono eseguite sequenzialmente
- Esistono 2 processi fondamentali:
  - **initial** → eseguito una sola volta
    (utile alla simulazione ma ignorato dalla sintesi)
  - **always** → eseguito a ciclo continuo
- Non possono essere annidati uno dentro l'altro
- Le variabili che vengono assegnate all'interno di un processo devono essere di tipo **reg** (integer, real, time,realtime)

Procedural Assignments (Es.)

```
module clk_gen(clk);
output clk;
reg clk;

initial
  clk = 1'b0;          // parte da 0
always
  #5 clk = !clk;       // ogni 5 units si inverte
initial
  #100 $finish         // istruzione per il simulatore
```

Naturalmente né i ritardi né l'inizializzazione possono venir sintetizzati … ma questa descrizione può risultare utile ad esempio per fornire un segnale ad un circuito sotto test

Always Block

Più comunemente il blocco always viene attivato in corrispondenza ad un evento (@)

```
always @(signal1 or signal2 or ..) begin
    ..
    end
```

Esecuzione sincronizzata sui segnali (circuiti **combinatori**) sensitivity list con **tutti** i segnali coinvolti

```
always @(posedge clk) begin
    ..
    end
```

Esecuzione sincronizzata sul fronte di salita del clk (circuiti **sequenziali**) sens. list con solo **segnali di sincronismo**

```
always @(negedge clk) begin
    ..
    end
```

Esecuzione sincronizzata sul fronte di discesa del clk (circuiti **sequenziali**) sens. list con solo **segnali di sincronismo**

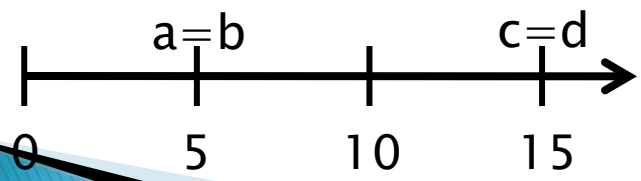
ALTERA

Block vs. NON-Block assignments

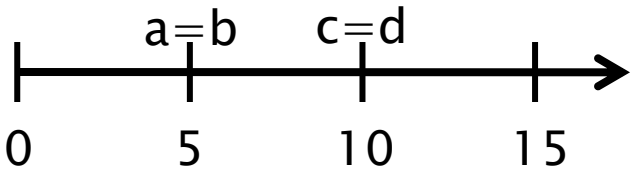
Vi sono due tipologie di assegnazione

- Block assignment (=)
  - l'esecuzione dell'istruzione segue l'ordine esatto con cui è stata scritta
- NON-Block assignment (<=)
  - consente di eseguire un istruzione senza dover attendere la conclusione dell'istruzione precedente
- N.B una variabile assegnata secondo uno dei due modi, NON può successivamente essere assegnata nell'altro modo

```
initial
begin
    #5 a=b;
    #10 c=d;
end
```

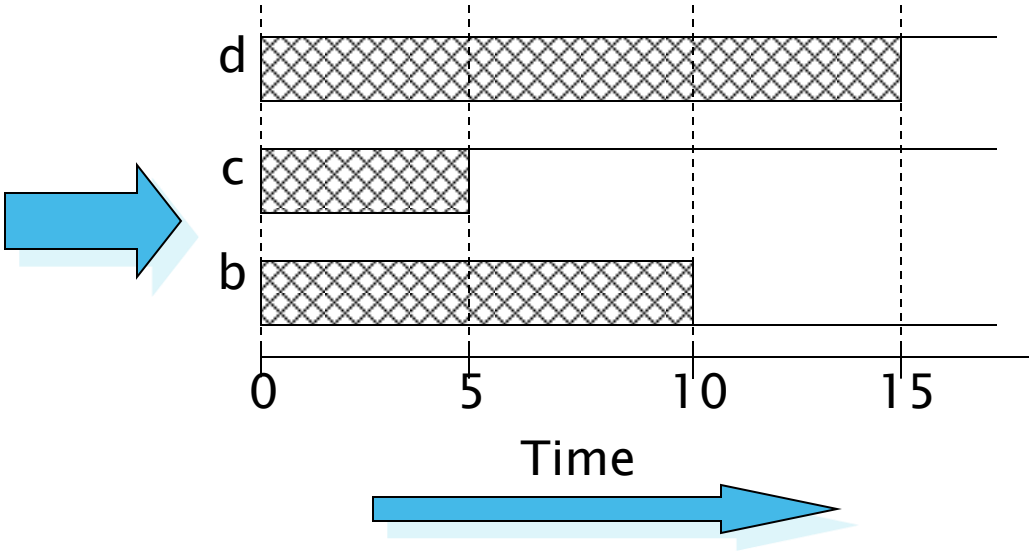



```
initial
begin
    #5 a<=b;
    #10 c<=d;
end
```



Timing

```
initial begin
        #5 c = 1;
        #5 b = 0;
        #5 d = c;
        end
```

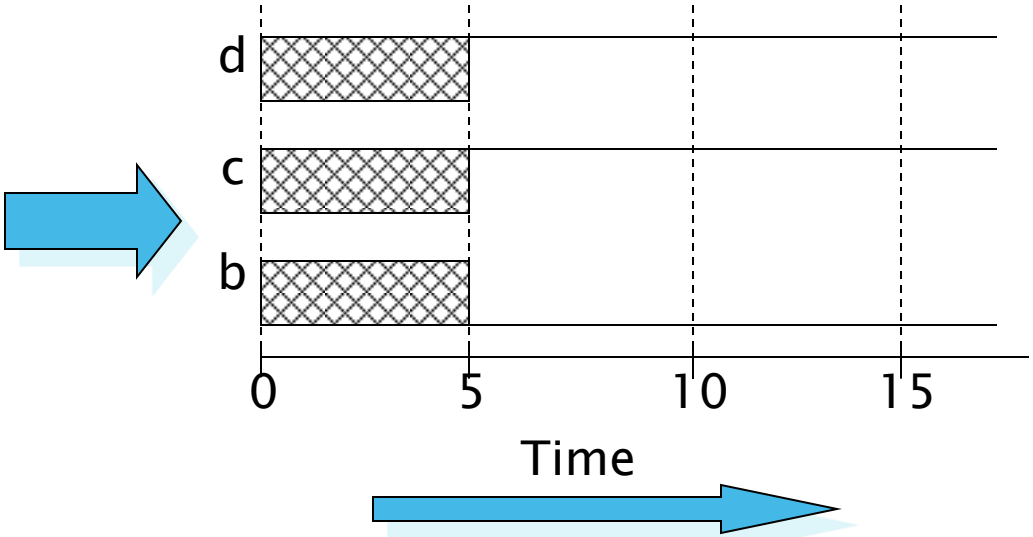


Each assignment is blocked by its previous one

Timing

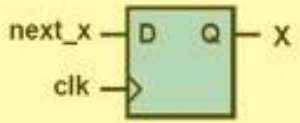
```
initial begin
        fork
        #5 c = 1;
        #5 b = 0;
        #5 d = c;
        join
        end
```

Assignments are not blocked here

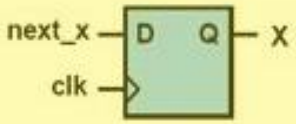


Block vs. NON Block assignments

```
always @( posedge clk )
begin
  x = next_x;
end
```

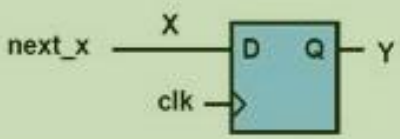



```
always @( posedge clk )
begin
  x <= next_x;
end
```

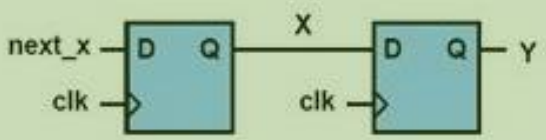


Same Behavior

```
always @( posedge clk )
begin
  x = next_x;
  y = x;
end
```




```
always @( posedge clk )
begin
  x <= next_x;
  y <= x;
end
```



Different Behavior

Descrizione comportamentale

- I processi o procedure si prestano particolarmente per descrizioni di tipo comportamentale del circuito
- Tipologie di dichiarazioni
  - **IF – ELSE**
    - condizioni valutate dall'alto verso il basso
    - ordine di priorità
  - **CASE**
    - condizioni valutate in un solo passo
    - senza alcuna priorità
  - **LOOP**
    - operazioni ripetitive

IF – ELSE

```
if (expr1)
      true_stmt1;

else if (expr2)
      true_stmt2;
..
else
      def_stmt;
```

Es. 4-to-1 mux:

```
module mux4_1(out, in, sel);
output out;
input [3:0] in;
input [1:0] sel;

reg out;
wire [3:0] in;
wire [1:0] sel;

always @(in or sel)
        if (sel == 0)
                out = in[0];
        else if (sel == 1)
                out = in[1];
        else if (sel == 2)
                out = in[2];
        else
                out = in[3];
endmodule
```

CASE

```
case (expr)

item_1, .., item_n:     stmt1;
item_n+1, .., item_m:   stmt2;
..
default:                def_stmt;

endcase
```

E.g. 4-to-1 mux:

```
module mux4_1(out, in, sel);
output out;
input [3:0] in;
input [1:0] sel;

reg out;
wire [3:0] in;
wire [1:0] sel;

always @(in or sel)
        case (sel)
        0: out = in[0];
        1: out = in[1];
        2: out = in[2];
        3: out = in[3];
        endcase
endmodule
```

CASEZ e CASEX

- CASEZ
  - i valori “z” sono interpretati come “don’t care”
  - i valori “z” possono essere sostituiti da “?”
- CASEX
  - i valori “z” e “x” sono interpretati come “don’t care”

```
Casez (encoder)
    4'b1???:   out =3;
    4'b01??:   out =2;
    4'b001?:   out =1;
    4'b0001:   out =0;
    default:   out =0;
endcase
```

Se encoder = 4’b1zzz out=3

```
Casex (encoder)
    4'b1xxx:   out =3;
    4'b01xx:   out =2;
    4'b001x:   out =1;
    4'b0001:   out =0;
    default:   out =0;
endcase
```

Se encoder = 4’b1xzx out=3

LOOP – for

```
for (init_assignment;cond;step_assignment)
     stmt;
```

Es.

```
module count(Y, start);
output [3:0] Y;
input start;

reg [3:0] Y;
wire start;
integer i;

initial
        Y = 0;

always @(posedge start)
        for (i = 0; i < 3; i = i + 1)
                #10 Y = Y + 1;
endmodule
```

LOOP – while

```
while (expr)
stmt;
```

Es.

```
module count(Y, start);
output [3:0] Y;
input start;

reg [3:0] Y;
wire start;
integer i;

initial
        Y = 0;

always @(posedge start) begin
        i = 0;
        while (i < 3) begin
                #10 Y = Y + 1;
                i = i + 1;
                end
        end
endmodule
```

LOOP – repeat

```
repeat (times)
stmt;
```

può essere un intero
o una variabile

E.g.

```
module count(Y, start);
output [3:0] Y;
input start;

reg [3:0] Y;
wire start;

initial
        Y = 0;

always @(posedge start)
        repeat (4) #10 Y = Y + 1;
endmodule
```

LOOP – forever

forever stmt;

Esecuzione continua fino alla fine della simulazione

Esempio tipico :
Generazione del clock

```
module test;

reg clk;

initial begin
        clk = 0;
        forever #10 clk = ~clk;
        end

other_module1 o1(clk, ..);
other_module2 o2(.., clk, ..);

endmodule
```

T_{clk} = 20 time units

Descrizione stutturale (Gate Level)

- Usa primitive di libreria oppure moduli già compilati:

  `and, nand, nor, or, xor, xnor, buf, not, bufif0, bufif1, notif0, notif1`

- Uso:

  `nand (out, in1, in2);` 2-input NAND without delay

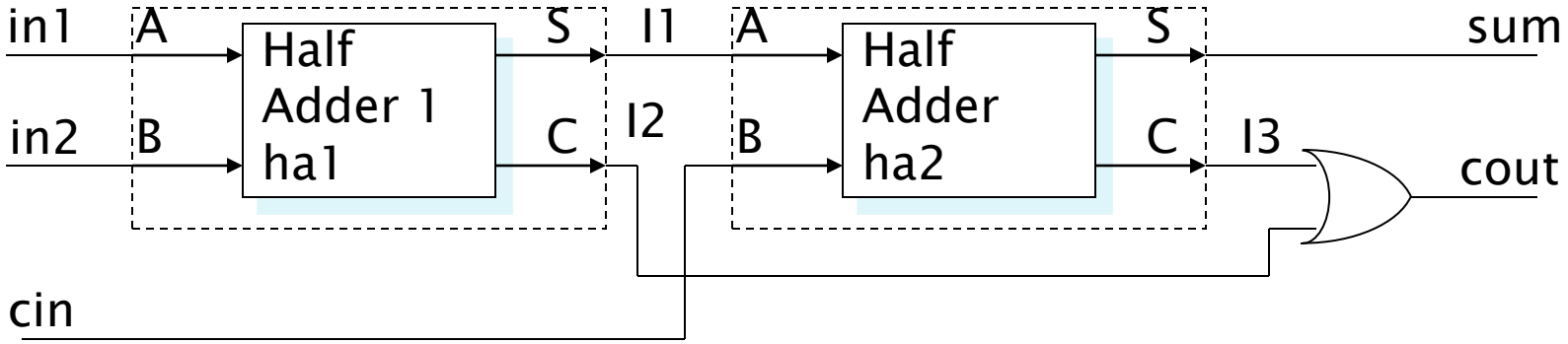
  `and #2 (out, in1, in2, in3);` 3-input AND with 2 t.u. delay

  `not #1 N1(out, in);` NOT with 1 t.u. delay and instance name

  `xor X1(out, in1, in2);` 2-input XOR with instance name

- Vanno scritti internamente al module ma fuori dalle procedures
- Fa largamente uso di continuous assignments

Esempio: Full Adder

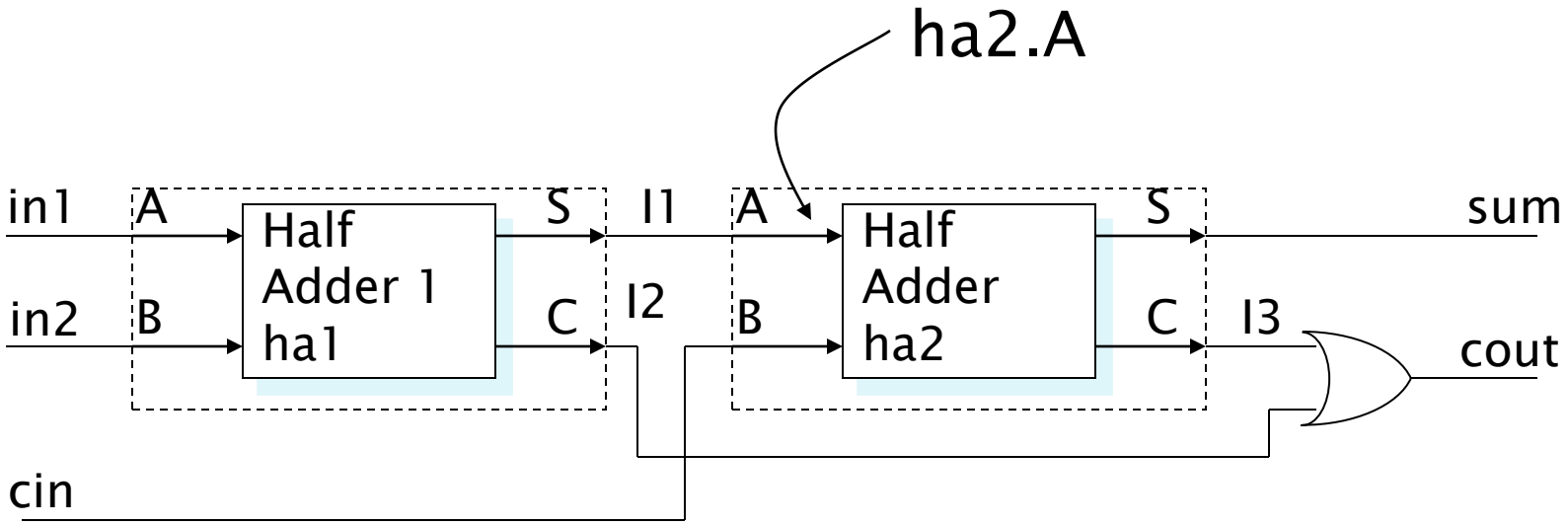



```
module full_adder(sum, cout, in1, in2, cin);
output sum, cout;
input in1, in2, cin;

wire sum, cout, in1, in2, cin;
wire I1, I2, I3;

half_adder ha1(I1, I2, in1, in2);
half_adder ha2(sum, I3, I1, cin);

assign cout = I2 || I3;

endmodule
```

Module name

Instance name

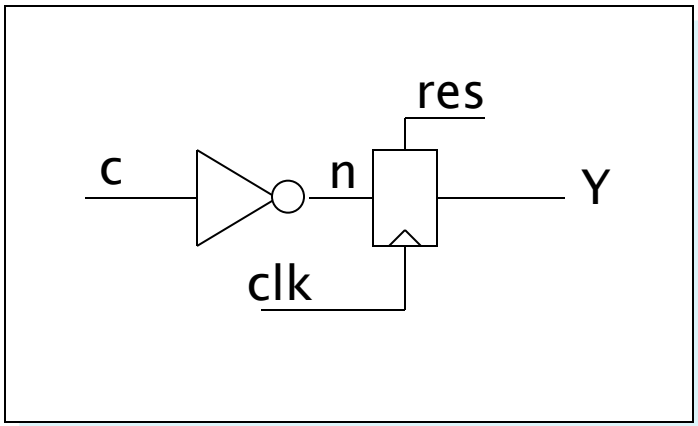
Hierarchical Names



Remember to use instance names, not module names

Mixed Model

Il codice può contenere contemporaneamente descrizioni strutturali e comportamentali




```
module simple(Y, c, clk, res);
output Y;
input c, clk, res;

reg Y;
wire c, clk, res;
wire n;

not(n, c); // gate-level

always @(res or posedge clk)
        if (res)
                Y = 0;
        else
                Y = n;
endmodule
```

Strutture sintattiche

- Il modo in cui si scrive il codice si ripercuote sulla realizzazione del circuito
- La sintesi riconosce particolari costrutti ed agisce di conseguenza
- Il codice va scritto già prevedendo la struttura finale del circuito e non sperando in un'intelligenza superiore (sintetizzatore) capace di realizzare ciò che noi stessi non siamo stati in grado di progettare
- il 70% della riuscita di un circuito risiede in come è stato scritto il codice sorgente, il 25% risiede nel tool di sintesi ed il rimanente 5% ad eventuali ottimizzazioni

Controlli sincroni ed asincroni

```
module sync (d,clk,clr,pre,q);
input d,clk,clr,pre;
output q;
reg q;

always @(posedge clk)
   begin
      if (clr)
          q <= 1'b0;
      else if (pre)
          q <= 1'b1;
      else
          q <= d;
   end
endmodule
```

```
module async (d,clk,clr,q);
input d,clk,clr;
output q;
reg q;

always @(posedge clk or posedge clr)
   begin
      if (clr)
          q <= 1'b0;
      else
          q <= d;
   end
endmodule
```

Clock Enable

```
module clk_ena(d,ena,clk,q);
input d,ena,clk;
output q;
reg q;

always @(posedge clk)
      if (ena)
        q <= d;

endmodule
```

Functional Counter

```
module cntr(q,aclr,clk,func,d);
input aclr,clk;
input [7:0] d;
input [1:0] func;
output [7:0] q;
reg [7:0] q;

always @(posedge clk or posedge aclr)
   begin
      if (aclr)
        q <= 8'h00;
      else
        case(funct)
            2'b00: q <= d;
            2'b01: q <= q+1;
            2'b10: q <= q-1;
            2'b11: q <= q;
        endcase
   end
endmodule
```

Tasks & Functions

- Similmente ad altri linguaggi implementano blocchi di codice (sottoprogrammi)
- Utili in caso di codice ripetitivo
- Favoriscono la leggibilità del codice
- Functions
  - Restituisce un solo valore basato sui suoi ingressi
  - Utilizzate soprattutto per logica combinatoria
  - Utilizzate in espressioni (Es: `assign c=mult(a,b)`)
- Task
  - Sono come le “procedure” in altri linguaggi
  - Utilizzate sia per logica sequenziale che combinatoria
  - Utilizzate come “dichiarazione” (statement)

(Es: `mult_out(a,b)`)

ALTERA

Functions def. (Es.)

```
function [15:0] mult
  input [7:0] a,b;
  reg [15:0] r;
  integer i;
begin
  if (a[0] ==1)
      r=b;
  else
      r=0;
  for (i=1;i<=7; i=i+1)
      begin
        if (a[i] ==1)
            r=r +(b <<i);
      end
  mult=r;
end
endfunction
```

- mult: nome della funzione ed uscita della stessa
- a, b : ingressi
- r, i : variabili locali
- realizzazione di un algoritmo combinatorio

Assegnazione:

```
assign c=mult(a,b);
```

Task

```
module test;

task add;
    input a,b;
    output c;
begin
     c=a+b;
endtask

initial
begin: init1
    reg p;
    add(1,0,p);
    $display ("p=%b",p);
end
```

Non vengono passati dati al task

Definizione del Task

Utilizzo del Task

I valori vengono passati nell'ordine stesso in cui compaiono

- Il richiamo ad un task viene "rimpiazzato" col testo descritto nel task stesso

Differenze fra Tasks e Functions

Functions

- Una funzione può richiamare un'altra funzione ma non un altro task
- Non può contenere delay, eventi (@) o controlli temporali (realizza solo circuiti combinatori ovvero una funzione viene eseguita in tempo zero
- Deve avere almeno un ingresso
- Restituisce un valore

Tasks

- Un task può richiamare altri task o funzioni
- Può contenere delay, eventi o controlli temporali (circuiti combinatori o sequenziali)
- Equivale ad una sostituzione di testo
- può avere zero o più ingressi.
- può avere più uscite
- può utilizzare segnali di in-out

System Tasks

Always written inside procedures

- $display(“..”, arg2, arg3, ..); → much like printf(), displays formatted string in std output when encountered
- $monitor(“..”, arg2, arg3, ..); → like $display(), but .. displays string each time any of arg2, arg3, .. Changes
- $stop; → suspends sim when encountered
- $finish; → finishes sim when encountered
- $fopen(“filename”); → returns file descriptor (integer); then, you can use $fdisplay(fd, “..”, arg2, arg3, ..); or $fmonitor(fd, “..”, arg2, arg3, ..); to write to file
- $fclose(fd); → closes file
- $random(seed); → returns random integer; give her an integer as a seed

$display & $monitor string format

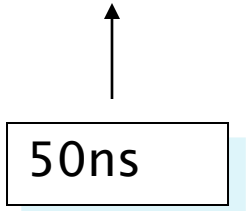
Format	Display
%d or %D	Display variable in decimal
%b or %B	Display variable in binary
%s or %S	Display string
%h or %H	Display variable in hex
%c or %C	Display ASCII character
%m or %M	Display hierarchical name
%v or %V	Display strength
%o or %O	Display variable in octal
%t or %T	Display in current time format
%e or %E	Display real number in scientific format
%f or %F	Display real number in decimal format
%g or %G	Display scientific or decimal, whichever is shorter

Compiler Directives

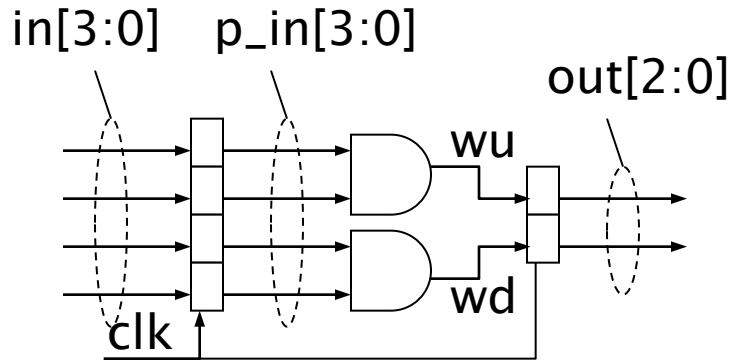
- include “filename” → inserts contents of file into current file; write it anywhere in code ..

- `define <text1> <text2> → text1 substitutes text2;
  - e.g. `define BUS reg [31:0]` in declaration part: `BUS data;`

- `timescale <time unit>/<precision>
  - e.g. `timescale 10ns/1ns` later: `#5 a = b;`



Parameters

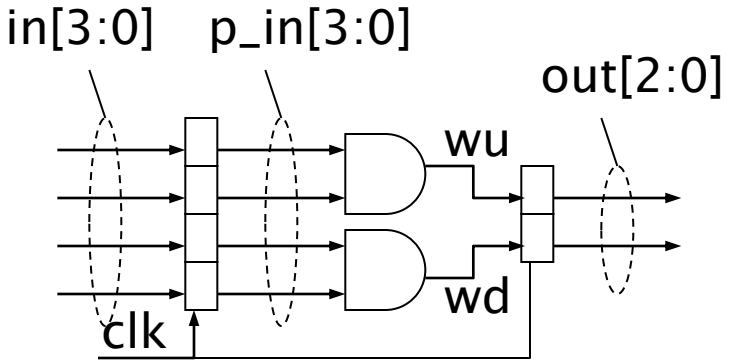


A. Implelementation without parameters

```
module dff4bit(Q, D, clk);
output [3:0] Q;
input [3:0] D;
input clk;

reg [3:0] Q;
wire [3:0] D;
wire clk;

always @(posedge clk)
        Q = D;
endmodule
```

```
module dff2bit(Q, D, clk);
output [1:0] Q;
input [1:0] D;
input clk;

reg [1:0] Q;
wire [1:0] D;
wire clk;

always @(posedge clk)
        Q = D;
endmodule
```

Parameters



A. Implelementation without parameters (cont.)

```
module top(out, in, clk);
output [1:0] out;
input [3:0] in;
input clk;

wire [1:0] out;
wire [3:0] in;
wire clk;

wire [3:0] p_in;     // internal nets
wire wu, wd;

assign wu = p_in[3] & p_in[2];
assign wd = p_in[1] & p_in[0];

dff4bit instA(p_in, in, clk);
dff2bit instB(out, {wu, wd}, clk);
// notice the concatenation!!

endmodule
```

Parameters

B. Implelementation with parameters

```
module dff(Q, D, clk);
parameter WIDTH = 4;
output [WIDTH-1:0] Q;
input [WIDTH-1:0] D;
input clk;

reg [WIDTH-1:0] Q;
wire [WIDTH-1:0] D;
wire clk;

always @(posedge clk)
       Q = D;

endmodule
```

```
module top(out, in, clk);
output [1:0] out;
input [3:0] in;
input clk;

wire [1:0] out;
wire [3:0] in;
wire clk;

wire [3:0] p_in;
wire wu, wd;

assign wu = p_in[3] & p_in[2];
assign wd = p_in[1] & p_in[0];

dff instA(p_in, in, clk);
// WIDTH = 4, from declaration

dff instB(out, {wu, wd}, clk);
       defparam instB.WIDTH = 2;
// We changed WIDTH for instB only

endmodule
```

ALTERA

Testing Modules

```
module top_test;
wire [1:0] t_out;    // Top's signals
reg [3:0] t_in;
reg clk;

top inst(t_out, t_in, clk); // Top's instance

initial begin          // Generate clock
       clk = 0;
       forever #10 clk = ~clk;
end

initial begin          // Generate remaining inputs
       $monitor($time, " %b -> %b", t_in, t_out);
       #5 t_in = 4'b0101;
       #20 t_in = 4'b1110;
       #20 t_in[0] = 1;
       #300 $finish;
end

endmodule
```