Basi di Verilog HDL

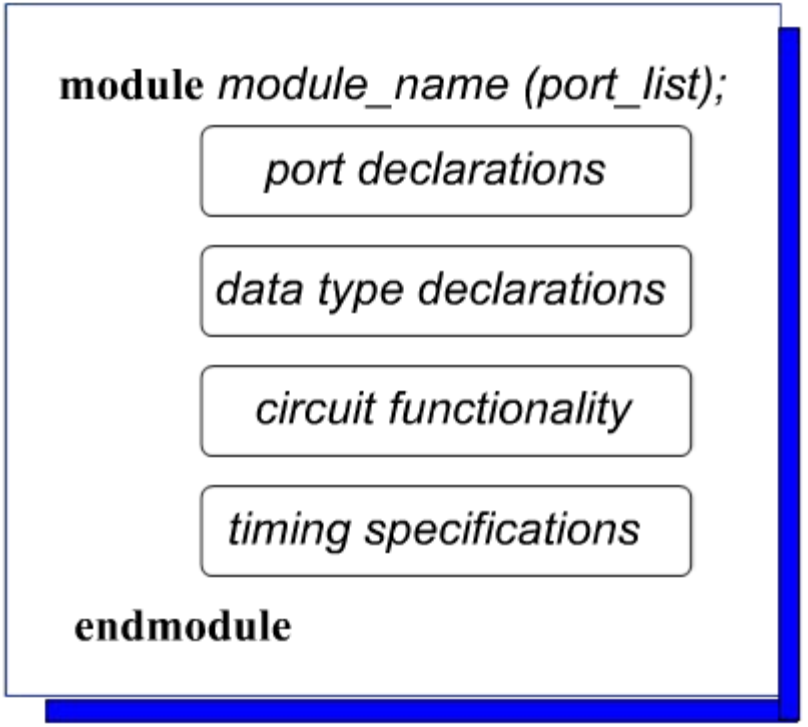
Introduzione alla sintassi di Verilog HDL

Basi di VERILOG HDL

- Verilog HDL è CASE sensitive
- Le “keywords” sono tutte in “minuscolo”
- Il carattere ‘;’ è il un terminatore di riga
- // è usato per commentare una singola linea
- /* */ è usato per commentare più linee
- Le specifiche temporali sono utili **solo** ai fini della simulazione – esse **NON** vengono sintetizzate

Module

- Rappresenta un processo, pertanto qualsiasi elemento o istruzione deve stare dentro un "Module"
- E' concorrente agli altri processi
- I vari "modules" possono risiedere in un medesimo file o su più files separati



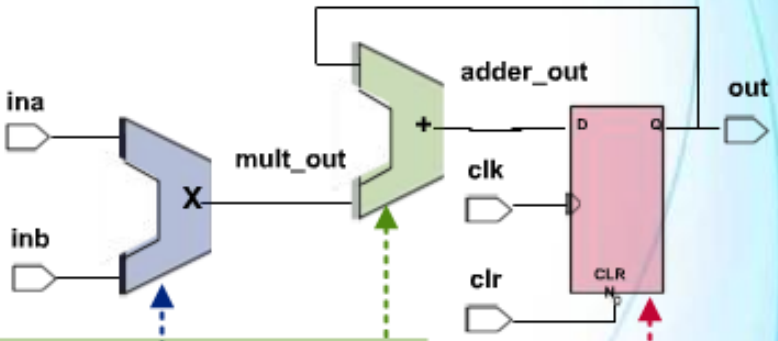
Un primo esempio

```
`timescale 1 ns/ 10 ps
module mult_acc (out, ina, inb, clk, clr);

input [7:0] ina, inb;
input clk, clr;
output [15:0] out;

wire [15:0] mult_out, adder_out;
reg [15:0] out;

parameter set = 10;
parameter hld = 20;

assign adder_out = mult_out + out;

always @ (posedge clk or posedge clr)
        if (clr)    out = 16'h0000;
        else        out = adder_out;

multa u1(.in_a(ina), .in_b(inb), .m_out(mult_out));

endmodule
```

Outline

Search

Ports and data types



- Continuous assignment
- Sequential block
- Module instanstation

Ports

- Port List
  - List con i nomi delle varie porte
  - Esempio:

```
module mult_acc(out,ina,inb,clk,clr);
```

- Port Types
  - input
  - output
  - inout
- Port Declaration
  - <port type> <port name>;
  - Esempio

```
input [7:0] ina,inb;
input clk,clr;
output [15:0] out;
```

User Identifiers

- Composti da {[A-Z], [a-z], [0-9], _, $}, ma ..
- .. Non possono iniziare con $ or [0-9]
  - `myidentifier` OK
  - `m_y_identifier` OK
  - `3my_identifier` NO
  - `$my_identifier` NO
  - `_myidentifier$` OK
- Case sensitivity
  - `myid` ≠ `Myid`

Classi di segnali

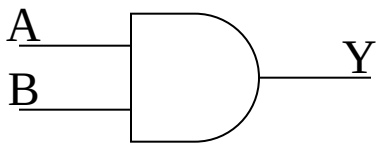
- Nets
- Registers

La principale differenza è che mentre le “Nets” devono essere collegate ad un driver per fornire un valore in uscita, i “registers” mantengono il valore anche quando risultano scollegati da questo

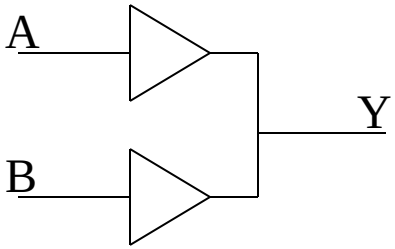
Nets (collegamenti)

- Rappresentano i collegamenti fisici fra i vari blocchi logici
- Assumono il valore *z* quando sono disconnesse
- Le "Nets" possono essere di diversi tipi
  - `wire`
  - `tri` (tri-state)
  - `wand/triand` (wired-AND)
  - `wor/trior` (wired-OR)
  - `Supply0/tri0` (stato alto)
  - `Supply1/tri1` (stato basso)
- Nei seguenti esempi Y viene valutato ogni qualvolta cambia A o B

Esempi




```
wire Y;  // declaration
assign Y = A & B;
```

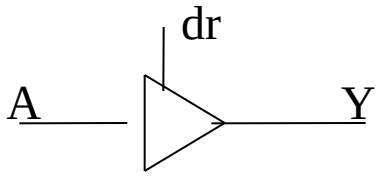



```
wand Y;  // declaration
assign Y = A;
assign Y = B;
```

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

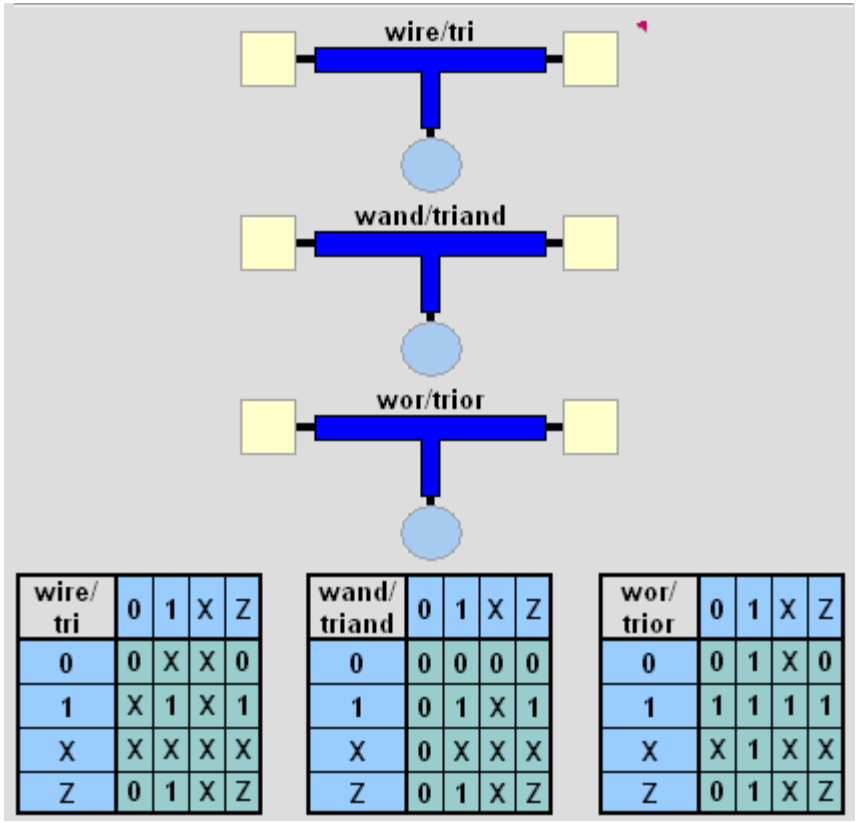
```
wor Y;  // declaration
assign Y = A;
assign Y = B;
```

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




```
tri Y;  // declaration
assign Y = (dr) ? A : z;
```

Nets



wire/ tri	0	1	X	Z
0	0	X	X	0
1	X	1	X	1
X	X	X	X	X
Z	0	1	X	Z

wand/ triand	0	1	X	Z
0	0	0	0	0
1	0	1	X	1
X	0	X	X	X
Z	0	1	X	Z

wor/ trior	0	1	X	Z
0	0	1	X	0
1	1	1	1	1
X	X	1	X	X
Z	0	1	X	Z

Registers

- Rappresentano variabili atte a memorizzare un valore
- Vi è un solo tipo: `reg`

```
reg A, C; // declaration
// assignments are always done inside a procedure
A = 1;
C = A; // C gets the logical value 1
A = 0; // C is still 1
C = 0; // C is now 0
```

- I valori dei “Reg” vengono aggiornati esplicitamente all’interno di procedure

Regole per le porte di I/O

Le porte di I/O di un modulo sottostanno a determinati vincoli

- I segnali di tipo **NETS** possono essere impiegati come porte di **input, output, inout**
- I segnali di tipo **REGISTERS** possono essere impiegati solo come porte di **output**.

Set di Valori

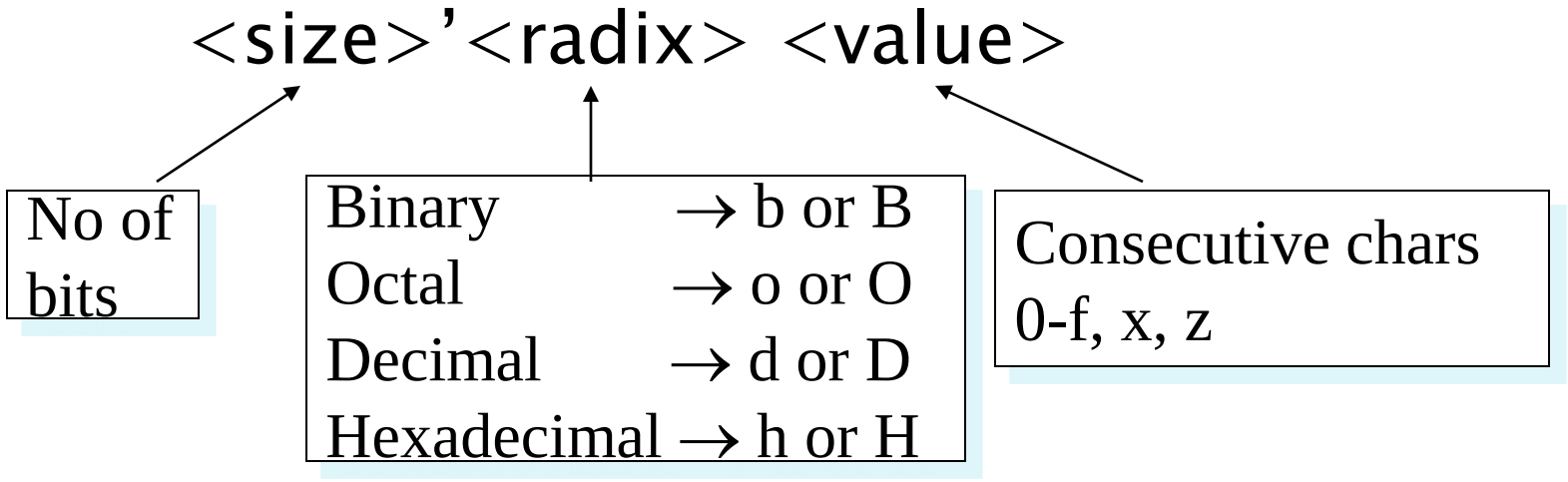
- 0: rappresenta il livello logico basso
- 1: rappresenta il livello logico alto
- x: rappresenta il livello sconosciuto
- z: rappresenta la linea in alta impedenza ovvero la linea aperta

Si noti che non non tutto ciò che viene scritto in Verilog, anche se sintatticamente corretto possa poi essere realizzato

Es: `if (sel == x)` …

Anche se sintatticamente potrebbe avere un certo senso non vi è assolutamente modo di realizzare tale funzione

Numeri in Verilog



- 4'b010x = 010x
- 8'd15 = 00001110
- 8'h b1 = 10110001
- 12'o 5zx0 = 101zzzxxx000

Numeri in Verilog

- Si può inserire "_" for migliorare la leggibilità
  - 12'b 000_111_010_100
  - 12'b 000111010100
  - 12'o 07_24

  } Rappresentano lo stesso numero

- Bit extension
  - MS bit = 0, x or z ⇒ extend this
    - 4'b x1 = 4'b xx_x1
  - MS bit = 1 ⇒ zero extension
    - 4'b 1x = 4'b 00_1x

Numeri in Verilog

- Se *size* viene omesso (sconsigliato)
  - O viene ricavato dal valore
  - Oppure assume in numero di bit specificato dal simulatore
  - Oppure prende un numero di bits di default

  (in Quartus-Altera il default è 32-bits)
- Se anche *radix* viene omesso si assume esso sia in decimale
  - 15 = <size>'d 15

Numeri negativi

- Sono contrassegnati dal segno “-” che precede il <size>

  Es:

  -8’d3; OK

  4’d-5; NO

- Vengono rappresentati in complemento a due

Vettori

- Rappresentano dei buses

```
wire[3:0]  busA;
reg [1:4]  busB;
reg [1:0]  busC;
```

- La cifra a sinistra rappresenta il bit più significativo
- Assegnazione a sezioni

```
busC = busA[2:1];   ⇔   { busC[1] = busA[2];
                         busC[0] = busA[1];
```

- Assegnazione tramite posizione

```
busB = busA;   ⇔   { busB[1] = busA[3];
                    busB[2] = busA[2];
                    busB[3] = busA[1];
                    busB[4] = busA[0];
```

Operatori aritmetici

- +: somma
- -: sottrazione, negativo
- *: moltiplicazione
- /: divisione
- %: modulo

- Se uno degli operandi è “x” o “z” il risultato sarà “x”
- Se operandi e risultato sono della stessa dimensione il bit di “carry” viene perso
- I valori negativi vengono salvati (in reg) come complemento a due, ma in operazioni successive vengono considerati come “unsigned”

Operatori aritmetici (Esempi)

```
ain =5; bin =10; cin =2'b10; din=2'b0x;

ain+cin = 7;
bin-cin =8;
-bin = -10;
ain*bin=50:
bin/ain =2;
bin%ain =0;


reg [15:0] regA;
..
regA = -4'd12;     // stored as 2^16-12 = 65524
regA/3             // evaluates to 21861
```

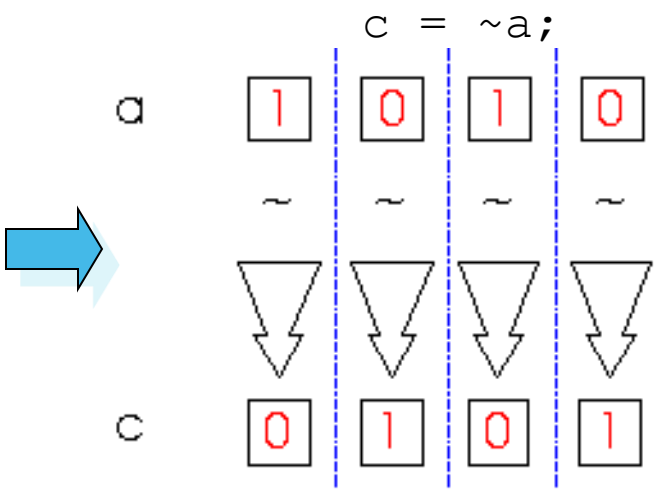
Bitwise Operators

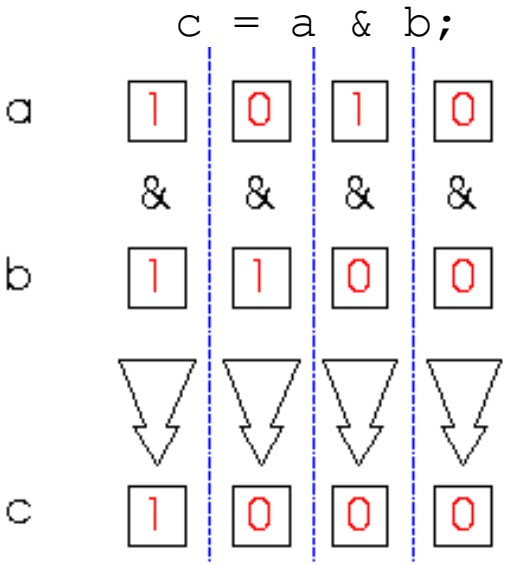
- & → bitwise AND
- | → bitwise OR
- ~ → bitwise NOT
- ^ → bitwise XOR
- ~^ or ^~ → bitwise XNOR

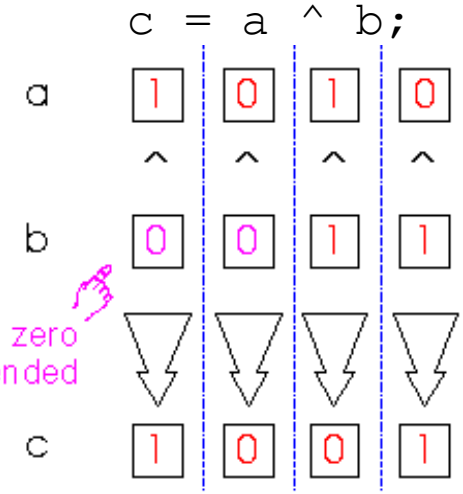
- Operazioni bit a bit
- Se le dimensioni sono diverse
  - I risultato assume la dimensione dell'operatore più grande
  - L'operando di dimensione più piccola viene "esteso" verso sinistra con degli zeri aggiuntivi

Bitwise Operators (esempi)

```
a = 4'b1010;
b = 4'b1100;
```








```
a = 4'b1010;
b = 2'b11;
```

Reduction Operators

- `&` → AND
- `|` → OR
- `^` → XOR
- `~&` → NAND
- `~|` → NOR
- `~^` or `^~` → XNOR

- One multi-bit operand → One single-bit result

```
a = 4'b1001;
..
c = |a; // c = 1|0|0|1 = 1
```

Relational Operators

- `>` → greater than
- `<` → less than
- `>=` → greater or equal than
- `<=` → less or equal than

- Result is one bit value: *0*, *1* or *x*

```
1 > 0          → 1
'b1x1 <= 0     → x
10 < z         → x
```

Equality Operators

==	→ logical equality	Return *0*, *1* or *x*
!=	→ logical inequality	
===	→ case equality	Return *0* or *1*
!==	→ case inequality	

Il "case" include anche la valutazione su x e z …
ovviamente tale operazione NON è sintetizzabile

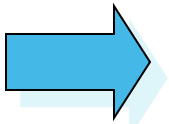
- `4'b 1z0x == 4'b 1z0x` → *x*
- `4'b 1z0x != 4'b 1z0x` → *x*
- `4'b 1z0x === 4'b 1z0x` → *1*
- `4'b 1z0x !== 4'b 1z0x` → *0*

ALTERA

Logical Operators

- `&&` → logical AND
- `||` → logical OR
- `!` → logical NOT
- Lavorano con valori a 1 bit: *0*, *1* or *x*
- Il risultato è di 1 solo bit: *0*, *1* or *x*

```
A = 6;
B = 0;
C = x;
```



```
A && B → 1 && 0 → 0
A || !B → 1 || 1 → 1
C || B → x || 0 → x
```

but C&&B=0

Shift Operators

>> → shift right

<< → shift left

Il risultato è della stessa dimensione del dato originale le posizioni vuote sono riempite da “0”

```
a = 4'b1010;
...
d = a >> 2;     // d = 0010
c = a << 1;     // c = 0100
```

Concatenation Operator

{op1, op2, ..} → concatenates op1, op2, .. to single number

Operands must be sized !!

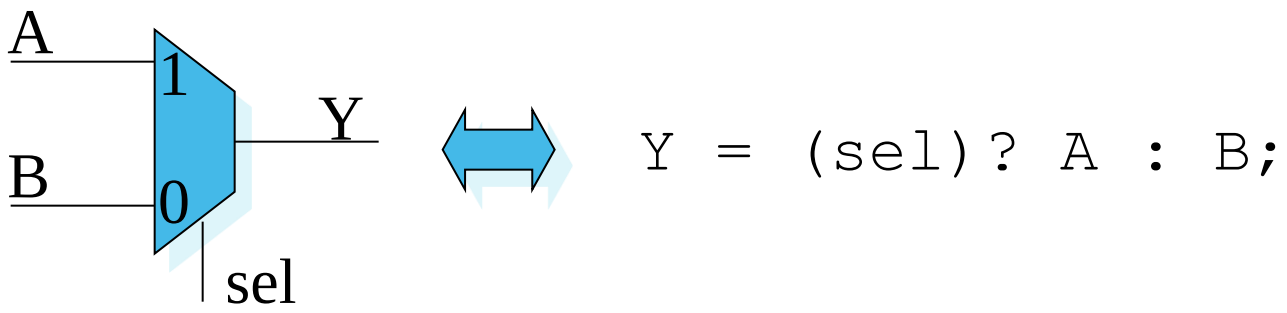
```
reg a;
reg [2:0] b, c;
..
a = 1'b 1;
b = 3'b 010;
c = 3'b 101;
catx = {a, b, c};       // catx = 1_010_101
caty = {b, 2'b11, a}; // caty = 010_11_1
catz = {b, 1};          // WRONG !!
```

Replication ..

```
catr = {4{a}, b, 2{c}};       // catr = 1111_010_101101
```

Conditional Operator

- `cond_expr ? true_expr : false_expr`
- Descrive un Multiplexer 2-in-1..



Operator Precedence

+-!~unary	highest precedence
*/%	
+-(binary)	
<< >>	
< <= => >	
== != === !==	
& ~&	
^ ^~ ~^	
| ~|	
&&	
||	
?: conditional	lowest precedence

Utilizzo delle parentesi per modificare la priorità

Integer & Real Data Types

- Dichiarazione

```
integer i,k;
real r;
```

- Impiegati come registers (all'interno di procedures)

```
i = 1; // assignments occur inside procedure
r = 2.9;
k = r; // k is rounded to 3
```

- Integers non sono inizializzati!!
- Reals sono inizializzati a *0.0*
- ***NON è detto siano sintetizzabili***

Time Data Type

- Impiegati solo in fase di simulazione
- NON vengono sintetizzati
- Declaration

```
time my_time;
```

- Utilizzati internamente a procedure

```
my_time = $time; // get current sim time
```

- La simulazione funziona in base al tempo simulato, non al tempo reale.

Arrays

- Sintassi

```
integer count[1:5]; // 5 integers
reg var[-15:16]; // 32 1-bit regs
reg [7:0] mem[0:1023]; // 1024 8-bit regs
```

- Accesso agli elementi
  - Elemento intero: `mem[10] = 8'b 10101010;`
  - Campo di un elemento (uso di variabile temp.):

```
reg [7:0] temp;
..
temp = mem[10];
var[6] = temp[2];
```

Arrays

- Limitazione: Non si può accedere all'intero array o ad un suo sottoinsieme di elementi

```
var[2:9] = ???; // WRONG!!
var = ???; // WRONG!!
```

- Non sono ammessi arrays multi-dimentionali

```
reg var[1:10] [1:100]; // WRONG!!
```

- Non sono ammessi per i Real

```
real r[1:10]; // WRONG !!
```

Strings

- Implemented with regs:

```
reg [8*13:1] string_val; // can hold up to 13 chars
..
string_val = “Hello Verilog”;
string_val = “hello”; // MS Bytes are filled with 0
string_val = “I am overflowed”; // “I ” is truncated
```

- Escaped chars:
  - \n newline
  - \t tab
  - %%%
  - \\\
  - \“”