Verilog 2001

Principali integrazioni rispetto Verilog95
e Signed Verilog

Introduzione

- Comprende 21 migliorie
  - Per supportare il disegnatore nella realizzazione di moduli Verilog
  - Alcune sono orientate a migliorare l'impiego e l'accuratezza nella generazione di modelli RTL sintetizzazbili
  - Altre orientate alla "scalabilità" ed alla "ri-usabilità"

Configuration block

- Introdotto per creare un'associazione tra i vari blocchi e le loro differenti versioni in libreria senza "toccare" il sorgente Verilog
- Impiega librerie virtuali per contenere i vari moduli
  - Un library map file è usato per associare la libreria ad un particolare direttorio su file-system
- **Keywords**: config, endconfig, design, instance, cell, use, liblist.

```
module test;
  ...
  myChip dut (...); /* instance of design */
  ...
endmodule

module myChip(...);
  ...
  adder a1 (...);
  adder a2 (...);
  ...
endmodule
```

```
/* location of RTL models (current directory) */
library rtlLib ./*.v;

/* Location of synthesized models */
library gateLib ./synth_out/*.v;

/* define a name for this configuration */
config cfg4

  /* specify where to find top level modules */
  design rtlLib.top

  /* set the default search order for finding
      instantiated modules */
  default liblist rtlLib gateLib;

  /* explicitly specify which library to use
      for the following module instance */
  instance test.dut.a2 liblist gateLib;
endconfig
```

Modelli scalabili

- Vi è spesso la necessità di realizzare modelli “scalabili”
  - Es: sommatore a 4 bit o a 16 bit … O N bit
- **Generate loop**: permette la generazione di istanze multiple o di moduli, ma si possono realizzare richiami mulltipli di variabili, nets, functions, tasks, continuous assign. , procedure (initial, always).
- **Keywords**: generate, endgenerate, genvar, localparam
- Esempio

```
module multiplier (a, b, product);
  parameter a_width = 8, b_width = 8;
  localparam product_width = a_width+b_width;
  input [a_width-1:0]     a;
  input [b_width-1:0]     b;
  output[product_width-1:0]product;

  generate
    if((a_width < 8) || (b_width < 8))
      CLA_multiplier #(a_width, b_width)
        u1 (a, b, product);
    else
      WALLACE_multiplier #(a_width, b_width)
        u1 (a, b, product);
  endgenerate
endmodule
```

Generate (cont.)

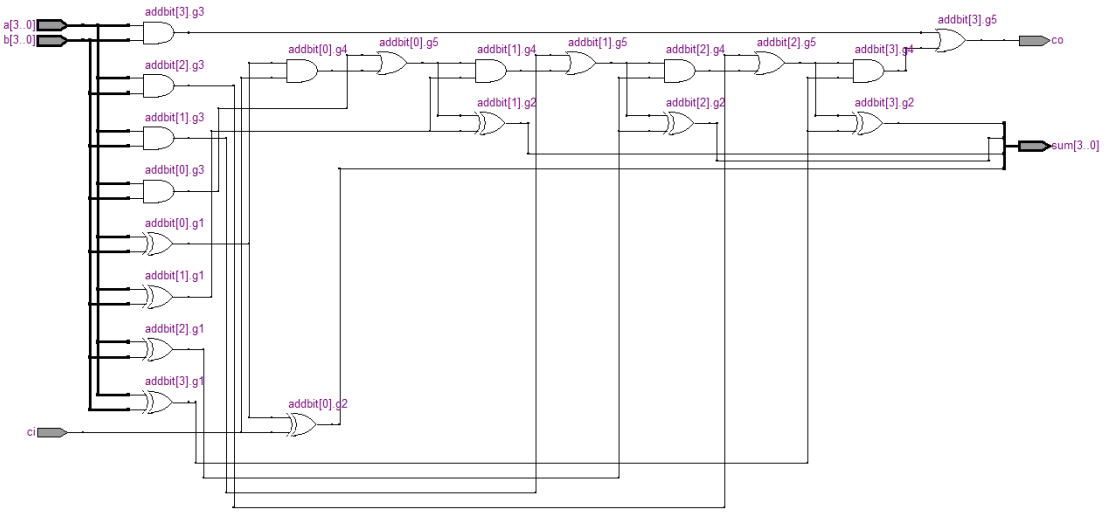
- Esempio:

```
module Nbit_adder (co, sum, a, b, ci);
  parameter SIZE = 4;
  output  [SIZE-1:0]  sum;
  output              co;
  input   [SIZE-1:0]  a, b;
  input               ci;
  wire    [SIZE:0]    c;

  genvar i;

  assign c[0] = ci;
  assign co = c[SIZE];

  generate
    for(i=0; i<SIZE; i=i+1)
      begin:addbit
        wire n1,n2,n3; //internal nets
        xor g1 (   n1, a[i], b[i]);
        xor g2 (sum[i],n1,   c[i]);
        and g3 (   n2, a[i], b[i]);
        and g4 (   n3, n1,   c[i]);
        or  g5 (c[i+1],n2,   n3);
      end
  endgenerate
endmodule
```



Nota: ogni istanza o rete generata ha un nome unico:

addbit[0].n1	addbit[0].g1
addbit[1].n1	addbit[1].g1
addbit[2].n1	addbit[2].g1
addbit[3].n1	addbit[3].g1

Constant Functions

- In Verilog 95 si può usare un parametro per definire le dimensioni ad esempio di un BUS, ma questo valore deve essere fisso o al più impiegare operazioni aritmetiche
- In Verilog 2001 si può definire una funzione che restituisca una costante (**constant function**)
- Aiuta nella realizzazione di modelli ri-utilizzabili

```
parameter WIDTH = 8;
wire [WIDTH-1:0] data;
```

```
module ram (address_bus, write, select, data);
  parameter SIZE = 1024;
  input [clogb2(SIZE)-1:0] address_bus;
  ...
function integer clogb2 (input integer depth);
    begin
      for(clogb2=0; depth>0; clogb2=clogb2+1)
        depth = depth >> 1;
    end
  endfunction
...
endmodule
```

Arrotondamento di log2(x)

Utile ad esempio a calcolare quanti bit impiegare in una memoria partendo dalla dimensione della stessa

Indexed vector part select

- In Verilog 95 si può accedere ad una sottoporzione di un BUS, ma l'indice deve essere costante
- In Verilog 2001 si può indicizzare in forma variabile (**indexed part select**) usando
  - Base expression – può essere variabile
  - Width expression – deve essere costante
  - Direction (+/–) indica la direzione
- Sintassi:

```
[base_expr +: width_expr] //positive offset
[base_expr -: width_expr] //negative offset
```

Indexed vector part select (cont.)

- Esempio

```
reg [63:0] word;
reg [3:0] byte_num; //a value from 0 to 7
wire [7:0] byteN = word[byte_num*8 +: 8];
```

Supponendo che il valore di byte_num sia = 4

`byteN = word[39:32]`

Dove 32 deriva dal valore di byte_num

E 39 in base alla dimensione imposta e alla direzione

Array Multidimensionali

- In Verilog 95 sono ammessi solo arrays 1D di variabili
- In Verilog 2001 sono ammessi array multidimensionali sia di variabili che di nets

```
//1-dimensional array of 8-bit reg variables
//(allowed in Verilog-1995 and Verilog-2001)
reg [7:0] array1 [0:255];
wire [7:0] out1 = array1[address];

//3-dimensional array of 8-bit wire nets
//(new for Verilog-2001)
wire [7:0] array3 [0:255][0:255][0:15];
wire [7:0] out3 = array3[addr1][addr2][addr3];
```

Accesso parziale ad un array

- In Verilog 95 NON si può accedere ad una parte di un array se non ricorrendo a variabili temporaneee
- In Verilog 2001 ciò è consentito

```
//select the high-order byte of one word in a
//2-dimensional array of 32-bit reg variables
reg [31:0] array2 [0:255][0:15];
wire [7:0] out2 = array2[100][7][31:24];
```

Signed Arithmetic extension

Nelle operazioni matematiche Verilog usa le tipolgie degli operandi per definire se operare in modo **signed** o **unsigned**

Una regola generale (con alcune eccezioni)

- se un solo operando è “unsigned” il risultato sarà “unsigned”
- per operare in modo “signed” tutti gli operandi devono essere “signed”

In Verilog 95

- **Integer** : sono di tipo **signed**
- **Regs** e **Nets** sono **unsigned**

Pertanto l’aritmetica “signed” è limitata all’uso di vettori a 32 bits (default)

In Verilog 2001 ci sono 5 migliorie:

- Reg and net data types can be declared as signed
- Function return values can be declared as signed
- Numbers in any radix can be declared as signed
- Operands can be converted from unsigned to signed
- Arithmetic shift operators have been added

Signed Arithmetic extension

Si è introdotta la keyword: **signed**

```
reg signed [63:0] data;
wire signed [7:0] vector;
input signed [31:0] a;
function signed [128:0] alu;
```

Nella definizione dei numeri si è introdotta la lettera 's' per indicare se sia da considerarsi "signed" o "unsigned"

```
16'hC501    //an unsigned 16-bit hex value
16'shC501   //a signed 16-bit hex value
```

Sono state introdotte due nuove "sistem functions" (`$signed e $unsigned`)

```
reg [63:0] a; //unsigned data type
always @(a) begin
  result1 = a / 2;          //unsigned arithmetic
  result2 = $signed(a) / 2;//signed arithmetic
end
```

E' stato introdotto lo "shift aritmetico"

```
D >> 3  //logical shift yields   8'b00010100
D >>> 3 //arithmetic shift yields 8'b11110100
```

Operatore **

- E' stato introdotto l'operatore di elevamento a potenza (simile al C)

```
always @(posedge clock)
  result = base ** exponent;
```

- Restituisce un valore reale se uno deglio operatori è reale.
- Se entrambi gli operatori sono interi restituisce un intero

Task e funzioni ricorsive

- E' stata introdotta la keyword **automatic** per dichiarare un task ricorsivo

```
function automatic [63:0] factorial;
  input [31:0] n;
  if (n == 1)
    factorial = 1;
  else
    factorial = n * factorial(n-1);
endfunction
```

recursive call

- In Verilog 95 le funzioni ed i task sono allocate staticamente
- In Verilog 2001 usando la keyword "automatic" task e funzioni vengono allocate dinamicamente

Always @*

- Ad ogni procedura va associata una **sensitivity list**
- Nelle procedure combinatorie questa lista deve integrare tutte le variabili coinvolte e potrebbe risultare piuttosto lunga
  - Altrimenti si possono creare dei latches non voluti
  - Oppure una discrepanza tra simulazione comportamentale e simulazione RTL

```
always @*   //combinational logic sensitivity
  if (sel)
    y = a;
  else
    y = b;
```

Comma separated sensitivity list

- Non aggiunge nuove funzionalità ma rende il sorgente più snello ed intuitivo

```
always @(a or b or c or d or sel)

always @(a, b, c, d, sel)
```

Gestione file I/O

- In Verilog 95
  - La capacità di accesso ai files è molto limitata
  - Inoltre non si possono aprire più di 31 files
- In Verilog 2001
  - sono stati integrati diversi system tasks e system functions
    - Dedicati ai files

    $ferror, $fgetc, $fgets, $fflush, $fread, $fscanf, $fseek, $fscanf, $ftel, $rewind $ungetc
    - Dedicati alla gestione delle stringhe

    $sformat, $swrite,$swriteb, $swriteh, $swriteo ,$sscanf
  - Il numero di file apribili contemporaneamente è 2^{30}

Estensione Automatica oltre 32bits

- In Verilog 95 l'estensione automatica si ferma a 32 bits, mentre i bits superiori sono posti a 0, per riempire bus di dimesioni maggiori ci vuole un'assegnazione specifica

```
Verilog-1995:
  parameter WIDTH = 64;
  reg [WIDTH-1:0] data;
  data = 'bz;  //fills with 'h00000000zzzzzzzz
  data = 64'bz; //fills with 'hzzzzzzzzzzzzzzzz
```

- In Verilog 2001 l'estensione automatica è estesa anche per bus con più di 32 bits

```
Verilog-2001:
  parameter WIDTH = 64;
  reg [WIDTH-1:0] data;
  data = 'bz;  //fills with 'hzzzzzzzzzzzzzzzz
```

Assegnazione dei parametri

- Per modificare un parametro di default
  - In Verilog95 ci sono due metodi:
    - Explicit redefinition
    - In line implicit redefinition (using #)

      Più conciso ma essendo posizionale più suscettibile di errori

```
module ram (...);
  parameter WIDTH = 8;
  parameter SIZE = 256;
  ...
endmodule

module my_chip (...);
  ...
  //Explicit parameter redefinition by name
  RAM ram1 (...);
  defparam ram1.SIZE = 1023;

  //Implicit parameter redefintion by position
  RAM #(8,1023) ram2 (...);
endmodule
```

Assegnazione dei parametri (cont.)

- Per modificare un parametro di default
  - In Verilog2001 si aggiunge un terzo metodo
    - In line **explicit** redefinition (using #)

      Più conciso ma i valori possono essere assegnati con ordine diverso

```
module ram (...);
  parameter WIDTH = 8;
  parameter SIZE = 256;
  ...
endmodule

module my_chip (...);
  ...
//In-line explicit parameter redefintion
RAM #(.SIZE(1023)) ram2 (...);
```

I/O declaration

- In Verilog 95
  - Ogni segnale in in ingresso e uscita da un modulo deve avere due dichiarazioni separate
    - La direzione in,out, inout
    - Il Tipo: wire, wand, … reg.
- In Verilog 2001
  - Le due dichiarazioni possono essere condensate in una sola istruzione

```
module mux8 (y, a, b, en);
  output  reg   [7:0] y;
  input   wire  [7:0] a, b;
  input   wire        en;
```

- Inoltre la dichiarazione puo essere contenuta nelle parentesi che contengono l'ordine delle porte di I/O

```
module mux8 (output  reg   [7:0] y,
             input   wire  [7:0] a,
             input   wire  [7:0] b,
             input   wire        en );
```

Inizializzazione di Variabili

- In Verilog 95
  - Una variabile reg va prima dichiarata
  - E successivamente può essere inizializzata tramite la procedura **initial**
- In Verilog 2001
  - La dichiarazione può prevedere una inizializzazione implicita (che avviene al tempo 0)

```
Verilog-1995:
  reg clock;
  initial
    clk = 0;

Verilog-2001:
  reg clock = 0;
```

ALTERA

Register sostituito da Variable

- Da quando è nato Verilog il termine “register” è stato usato per indicare un insieme di data types
  - Reg, integer, real, time, realtime
- Questo ha comportato un po’ di confusione in quanto lo si portava ad associare al registro inteso com Flip Flop
- Register comunque NON è una keyword in Verilog
- Dal 2001 sui manuali il termine “**register**” è stato sostituito dal termine “**variable**” che genera meno confusione.

Enhanced Compilation

- Verilog-1995 supporta la conditional compilation, usando le direttive per la compilazione `**ifdef**, `**else** and `**endif** compiler directives.
- Verilog-2001 ha aggiunto ulteriori “compiler directives” con `**ifndef** and `**elsif**.

Attributes

- Verilog era nato soprattutto orientato alla simulazione
- Successivamente altri Tool lo hanno adottato
  - Spesso questi tool richiedono comandi specifici (non standard)
  - In Verilog 95 non vi era modo di includere questi comandi anche se alcuni tool specifici utilizzavano i commenti //
  - In Verilog 2001 è stato aggiunto un meccanismo per definire questi comandi chaimati **attributes** e sono racchiusi tra **(* *)**
  - Gli attributes NON sono standard

```
(* parallel case *) case (1'b1) //1-hot FSM
  state[0]: ...
  state[1]: ...
  state[2]: ...
endcase
```

Gestione dei tempi

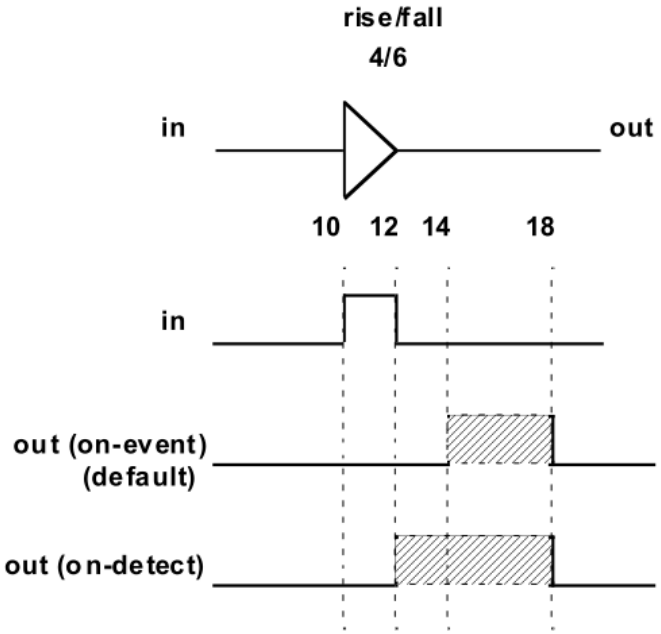
- Verilog è nato quando la tecnologia era tra i 2u ed i 5u e la valutazione dei tempi di ritardo poteva essere meno accurata.
- Con Verilog 2001 è nata l'esigenza di modellare altri casi:

Es: se in ingresso c'e' un impulso più breve del tempo di ritardo della porta come ci si deve comportare ?

Si può specificare al sistema se effettuare una detection di tipo **pulsestyle_onevent** and **pulsestyle_ondetect.**

Ci sono molti altri "upgrades"

```
specify
  pulsestyle_ondetect out;
  (in => out) = (4,6);
endspecify
```



Signed Verilog

Considerazioni sull'estensione Signed Verilog
Pregi e Difetti

Type Casting

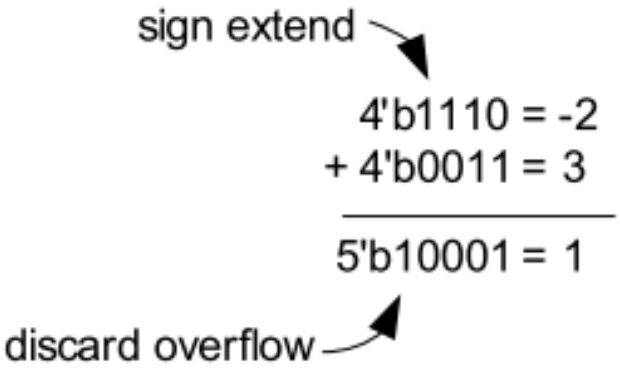
- $signed e $unsigned

- Se il risultato ha una dimensione maggiore dell'originale
  - A = $unsigned(B) → fill with zeroes
  - A = $signed(B) → sign extension

  (se il bit di segno è X o Z : si estendono questi)
- Se il risultato ha una dimensione minore dell'originale
  - Si effettua una semplice troncatura
- Se sono uguali
  - Non ci sono WARNING in fase di sintesi

Signed Addition

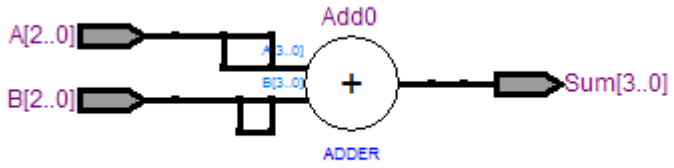
- La somma di due dati a (n) bit produce un dato a (n+1) bit
  - I dati in ingresso devono venir “signed extended”
  - L’ultimo carry va eliminato
  - ES: adding –2 (3’b110) to 3 (3’b011) = 1 (4’b0001)



Signed Addition (cont)

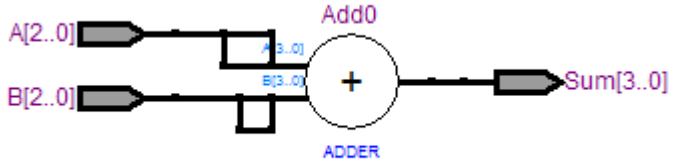
- Verilog 95

```
module add_signed_1995 (
                        input [2:0] A,
                        input [2:0] B,
                        output [3:0] Sum
                        );
  assign                Sum = {A[2],A} + {B[2],B};
endmodule // add_signed_1995
```



- Verilog 2001

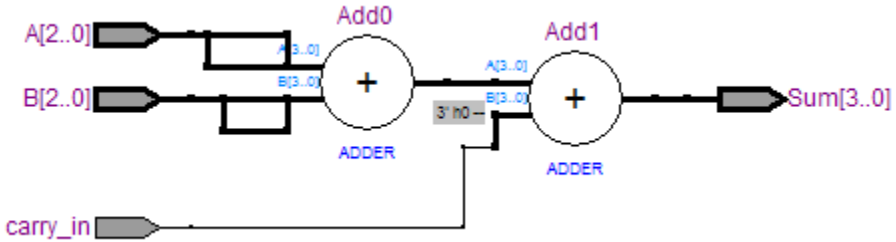
```
module add_signed_2001 (
                        input signed [2:0] A,
                        input signed [2:0] B,
                        output signed [3:0] Sum
                        );
  assign                Sum = A + B;
endmodule // add_signed_2001
```



Signed Addition + carry_in

- Verilog 95

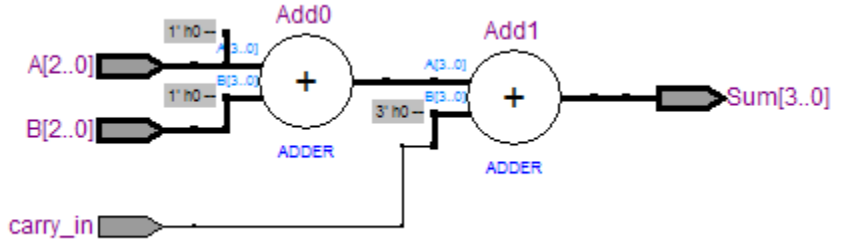
```
module add_carry_signed_1995 (
                        input [2:0] A,
                        input [2:0] B,
                        input carry_in,
                        output [3:0] Sum
                        );
  assign                Sum = {A[2],A} + {B[2],B} + carry_in;
endmodule // add_carry_signed_1995
```



- Verilog 2001

```
module add_carry_signed_2001 (
                        input signed [2:0] A,
                        input signed [2:0] B,
                        input carry_in,
                        output signed [3:0] Sum
                        );
  assign                Sum = A + B + carry_in;
endmodule // add_carry_signed_2001
```

- Problema: l'operazione somma usa sia valori signed che unsigned → risultato unsigned (0 extension)



Signed Addition + carry_in

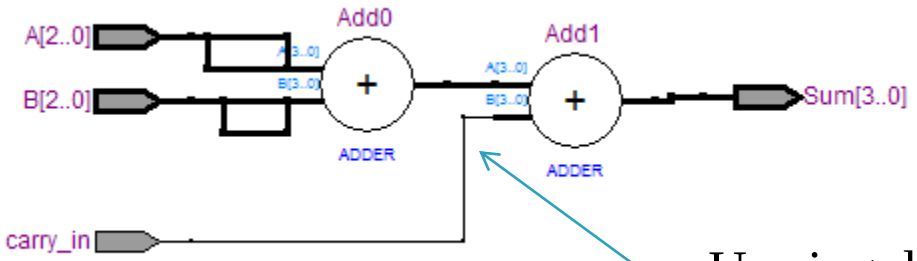
Idea: forse si potrebbe scrivere:

```
A + B + $signed(carry_in)
```

Ma questo crea un vero errore: cosa accade se `carry_in = 1` ?

- L'operatore `$signed` lo trasforma in `4'b1111` che equivale a `-1` e pertanto lo sottrae

Analogo errore se carry_in fosse dichiarato di tipo signed.

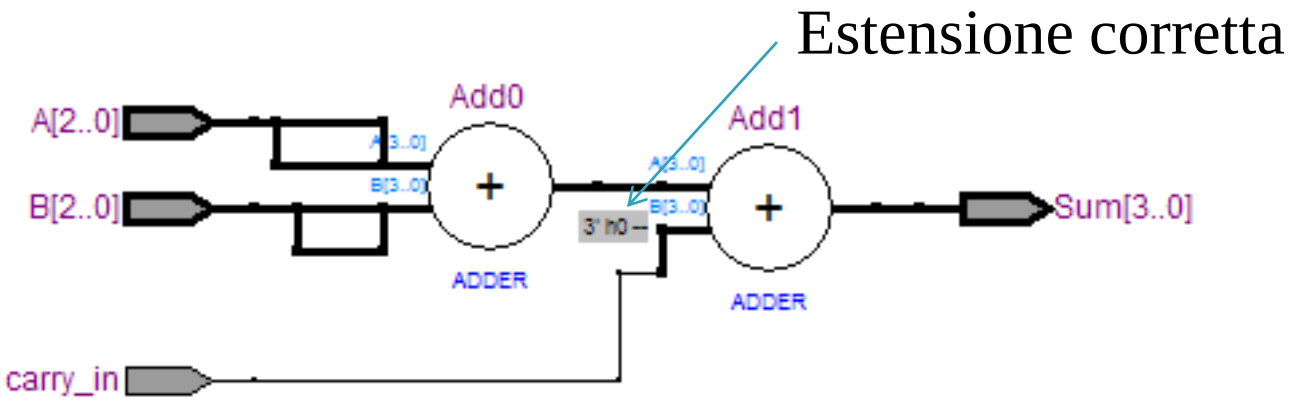


Un singolo bit viene "esteso" a 4 bits tramite replica

Signed Addition + carry_in

- La soluzione corretta è:

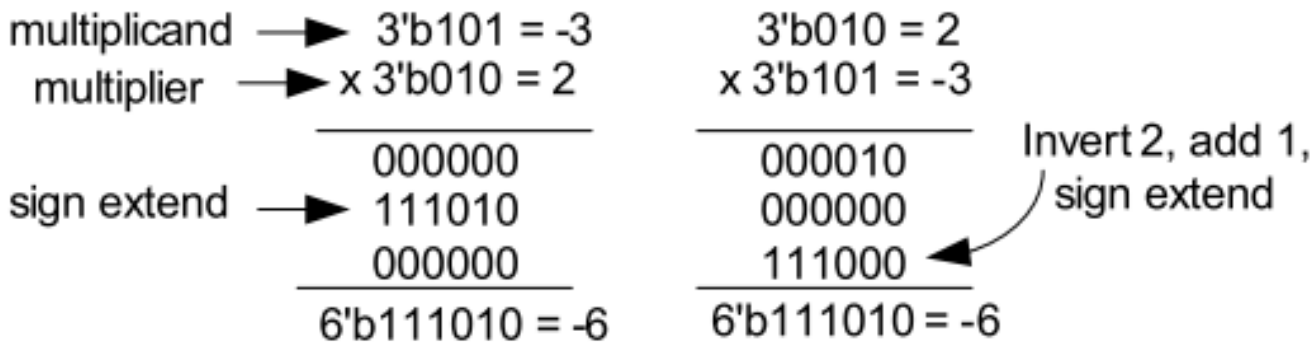
```
module add_carry_signed_final (
                              input signed [2:0] A,
                              input signed [2:0] B,
                              input carry_in,
                              output signed [3:0] Sum
                              );
  assign      Sum = A + B + $signed({1'b0,carry_in});
endmodule // add_carry_signed_final
```



Signed Multiplication

- Il prodotto di due dati a (n) bit produce un dato a (2n) bit
- Il moltiplicatore (2o fattore) va analizzato un bit alla volta da dx (LSB) a sx (MSB).
  - Il moltiplicando va in AND col bit del moltiplicatore e shiftato
  - se il moltiplicando è neg. → sign extended
  - Se l'ultimo bit del moltiplicatore (bit di segno) è 1: complemento a 2 e sign extended

  - Es: -3 (3'b101) * 2 (3'b010) = -6 (6'b111010)



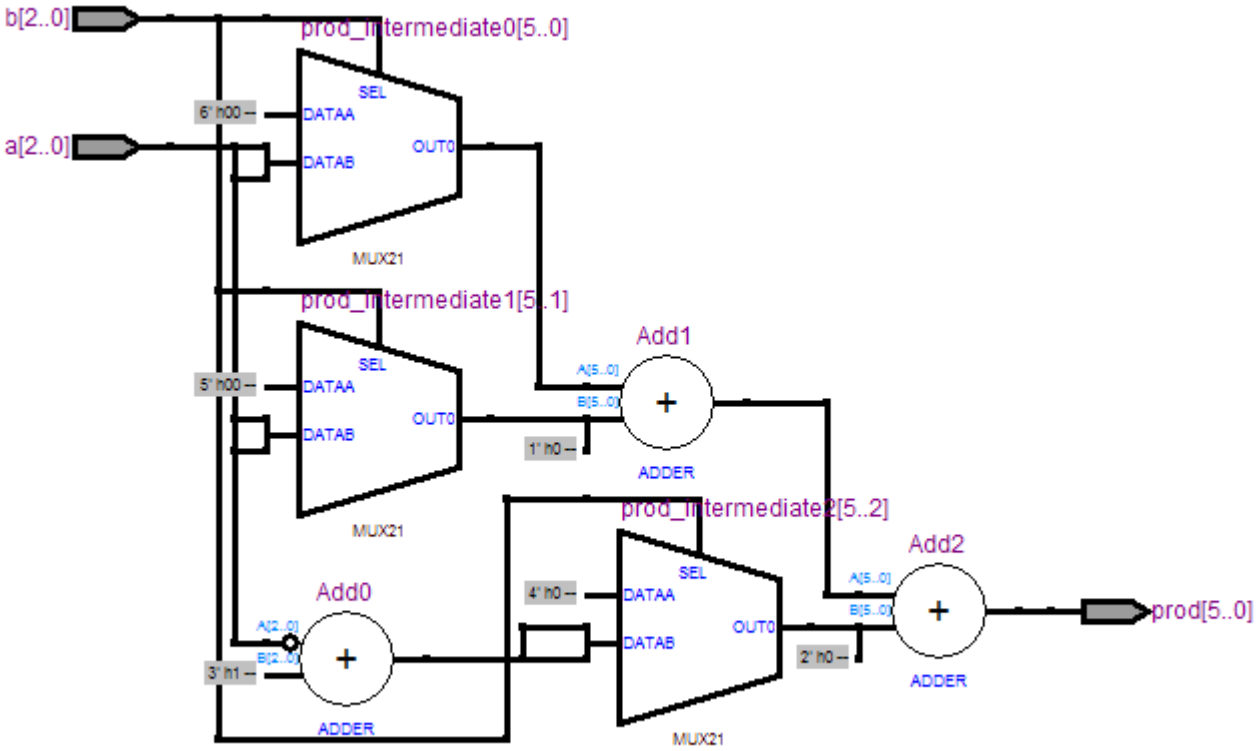
Signed Multiplication

- Verilog 95

```
module mult_signed_1995 (
                    input [2:0] a,
                    input [2:0] b,
                    output [5:0] prod
                    );
  wire [5:0] prod_intermediate0;
  wire [5:0] prod_intermediate1;
  wire [5:0] prod_intermediate2;
  wire [2:0] inv_add1;
  assign     prod_intermediate0 = b[0] ? {{3{a[2]}}, a}       : 6'b0;
  assign     prod_intermediate1 = b[1] ? {{2{a[2]}}, a, 1'b0} : 6'b0;
  // Do the invert and add1 of a.
  assign     inv_add1 = ~a + 1'b1;
  assign     prod_intermediate2 = b[2] ? {{1{inv_add1[2]}},
                                          inv_add1, 2'b0} : 6'b0;
  assign     prod =  prod_intermediate0 +  prod_intermediate1 +
                                           prod_intermediate2;
endmodule
```

Signed Multiplication

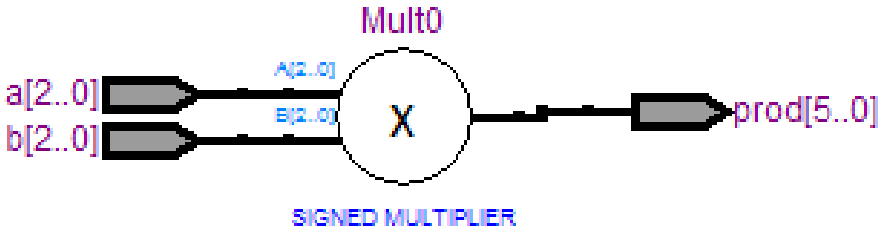
- Verilog 95



Signed Multiplication

- Verilog 2001

```
module mult_signed_2001 (
                  input  signed [2:0] a,
                  input  signed [2:0] b,
                  output signed [5:0] prod
                  );
  assign          prod = a*b;
endmodule
```

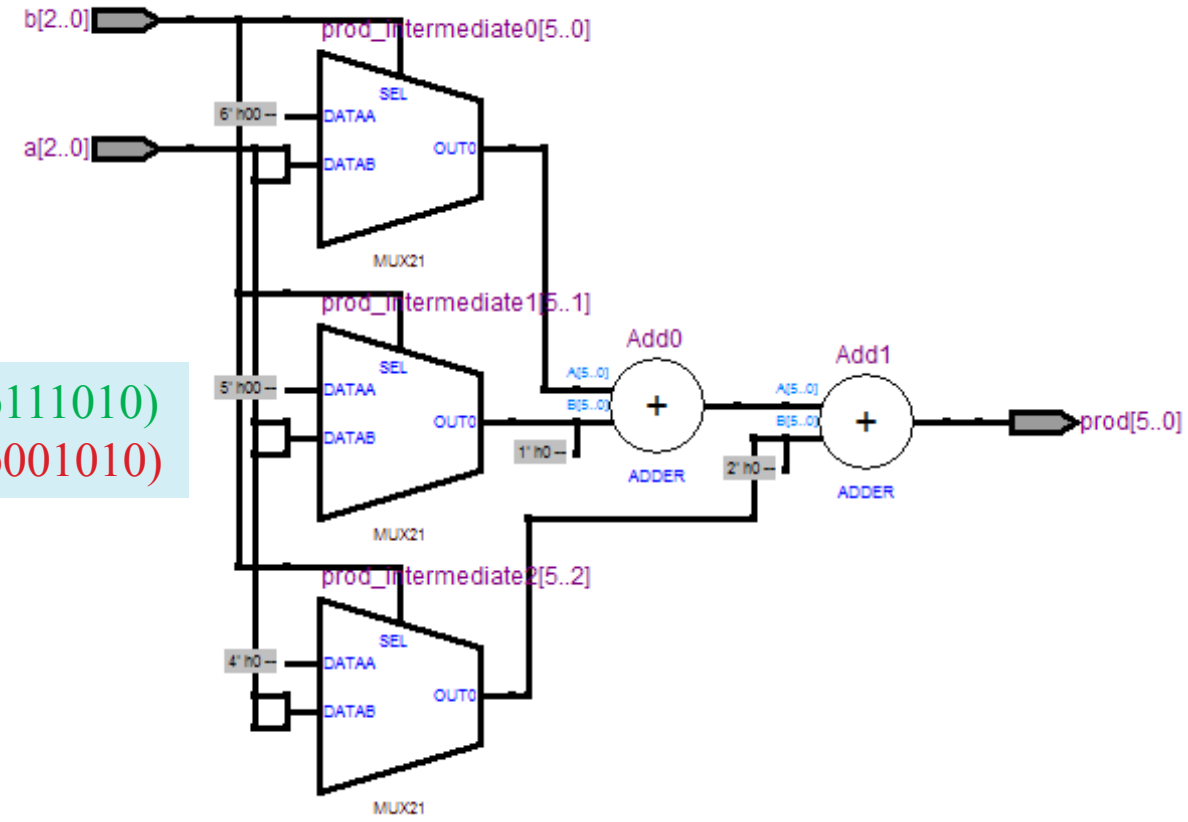


Signed by Unsigned Multiplier

Verilog 95

Bisogna conoscere
quale porta è signed
e quale unsigned

–3 (3'b101) by 2 (3'b010) = –6 (6'b111010)
2 (3'b010) by –3 (3'b101) = 10 (6'b001010)




```
module mult_signed_unsigned_1995 (
                input [2:0] a,
                input [2:0] b,
                output [5:0] prod
                );
  wire [5:0] prod_intermediate0;
  wire [5:0] prod_intermediate1;
  wire [5:0] prod_intermediate2;
  assign     prod_intermediate0 = b[0] ? {{3{a[2]}}, a}      : 6'b0;
  assign     prod_intermediate1 = b[1] ? {{2{a[2]}}, a, 1'b0} : 6'b0;
  assign     prod_intermediate2 = b[2] ? {{1{a[2]}}, a, 2'b0} : 6'b0;
  assign     prod =  prod_intermediate0 +  prod_intermediate1 +
                                           prod_intermediate2;
endmodule
```

Signed by Unsigned Multiplier

Verilog 2001 (ERRONEO): vi è implicitamente una conversione del valore da signed ad unsigned

```
module mult_signed_unsigned_2001 (
                    input  signed [2:0] a,
                    input  [2:0] b,
                    output signed [5:0] prod
                    );
  assign           prod = a*b;
endmodule
```

–3 (3'b101) by 2 (3'b010) = 10 (6'b001010)
2 (3'b010) by –3 (3'b101) = 10 (6'b001010)

ALTERA

Signed by Unsigned Multiplier

Verilog 2001:

```
module mult_signed_unsigned_2001 (
                    input  signed [2:0] a,
                    input  [2:0] b,
                    output signed [5:0] prod
                    );
  assign            prod = a*$signed(b);
endmodule
```

Ma cosa accade se MSB(b) ==1 ? …
verrebbe interpretato come negativo!

–3 (3'b101) by 2 (3'b010) = –6 (6'b111010)
2 (3'b010) by 7 (3'b111) = – 2 (6'b1111101)
–2 (3'b110) by 7(3'b111) = –6 (6'b111010)

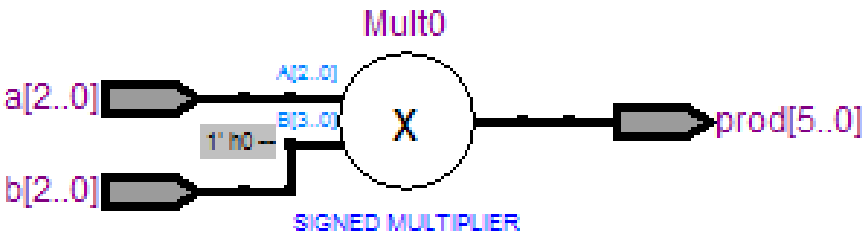
Signed by Unsigned Multiplier

Verilog 2001:

```
module mult_signed_unsigned_2001 (
                    input  signed [2:0] a,
                    input  [2:0] b,
                    output signed [5:0] prod
                    );
  assign          prod = a*$signed({1'b0,b});
endmodule
```

Soluzione corretta!

ma richiede un moltiplicatore un pochino più grande



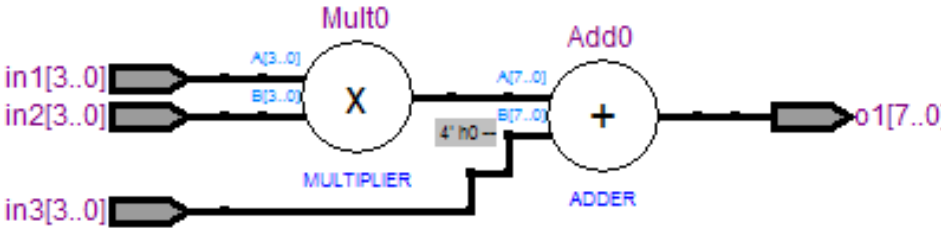
Espressioni

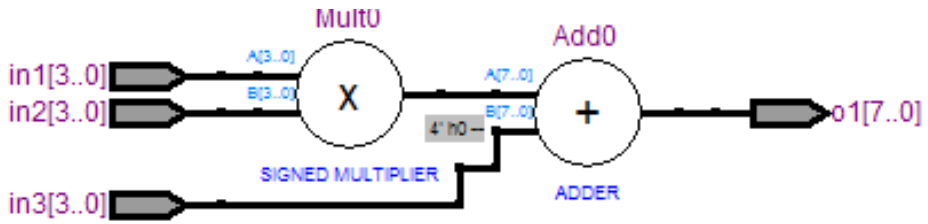
- Come viene interpretato il seguente codice ?

```
module mult_add (
                input signed [3:0] in1, in2,
                 input [3:0] in3,
                 output [7:0] o1;
                 );
assign o1 =  in1 * in2 + in3;
endmodule
```

  - Signed multiplier and unsigned adder
  - Unsigned both multiplier and adder

(software diversi possono interpretare in modi diversi)





Regole nelle espressioni

- Le seguenti operazioni sono sempre considerati "**unsigned**" indipendentemente dalla natura degli operandi.

  1. Bit-select results
  2. Part-select results,

     (anche se si seleziona l'intero vettore)
  3. Concatenation results
  4. Comparison results

Signed Saturation

- Assegnare una variabile con meno bit ad una con più bit può essere fatto facilmente tramite “sign extension”

```
4'b 0101 → 8'b 00000101
4'b 1001 → 8'b 11111001
```

- L’opposto deve tener conto anche di una eventuale saturazione rispettivamente al massimo positivo (**overflow**) o al minimo negativo (**underflow**)

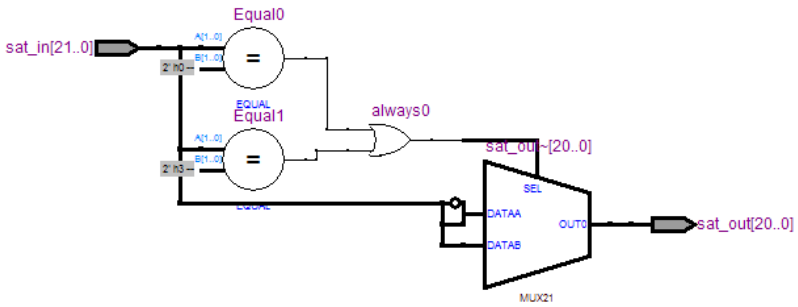
```
8'b 00000101 → 4'b0101
8'b 11111001 → 4'b1001
8'b 01100101 → 4'b0111
8'b 11011001 → 4'b1000
```

Signed Saturation

Regole:

Si esaminino i bit da “saturare” più uno dal MSB al LSB

- Se sono tutti uguali si possono troncare
- Se sono diversi si guardi al MSB
  - Se questo vale 0 si saturi al MAX pos.
  - Se questo vale 1 si saturi al MIN neg.




```
module sat (sat_in, sat_out);

parameter IN_SIZE = 21;  // Default is to saturate 22 bits to 21 bits
parameter OUT_SIZE = 20;
input      [IN_SIZE:0] sat_in;
output reg [OUT_SIZE:0] sat_out;

wire [OUT_SIZE:0]   max_pos = {1'b0,{OUT_SIZE{1'b1}}};
wire [OUT_SIZE:0]   max_neg = {1'b1,{OUT_SIZE{1'b0}}};

always @* begin
   // Are the bits to be saturated + 1 the same?
   if ((sat_in[IN_SIZE:OUT_SIZE]=={IN_SIZE-OUT_SIZE+1{1'b0}}) ||
     (sat_in[IN_SIZE:OUT_SIZE]=={IN_SIZE-OUT_SIZE+1{1'b1}}))
           sat_out = sat_in[OUT_SIZE:0];
    else if (sat_in[IN_SIZE])  // neg underflow.  go to max neg
           sat_out = max_neg;
    else // pos overflow, go to max pos
           sat_out = max_pos;
  end
endmodule
```

Saturation Module (impiego)

- Esempio di impiego

```
wire signed [4:0] A, B, C;
reg signed [2:0] D, E, F;

A = 5'sb11101;
B = 5'sb01001;
C = 5'sb10001;

// Drop two MSB's. D will equal 3'sb101
sat  #(.IN_SIZE(4), .OUT_SIZE(2))  satA (.sat_in(A), .sat_out(D));

// Go to max positive .  E will equal 3'sb011
sat  #(.IN_SIZE(4), .OUT_SIZE(2))  satB
                (.sat_in(B), .sat_out(E));

// Go to max negative .  F will equal 3'sb100
sat  #(.IN_SIZE(4), .OUT_SIZE(2))  satC
                (.sat_in(C), .sat_out(F));
```

Regole riassuntive

1. Se in un'operazione un operando è unsigned l'intera operazione sarà unsigned
2. Controllare eventuali synthesis warnings
3. Gli operandi "signed" vengono "signed extended" per essere compatibili con operandi a più bits
4. Trasformare un operando usando **$unsigned** rende l'operazione unsigned. Eventualmente estendere con '0'
5. Trasformare un operando usando **$signed** rende l'operando signed. Per garantire che non diventi negativo aggiungere uno 0 come MSB prima della trasformazione
6. Attenzione al risultato di espressioni "miste"