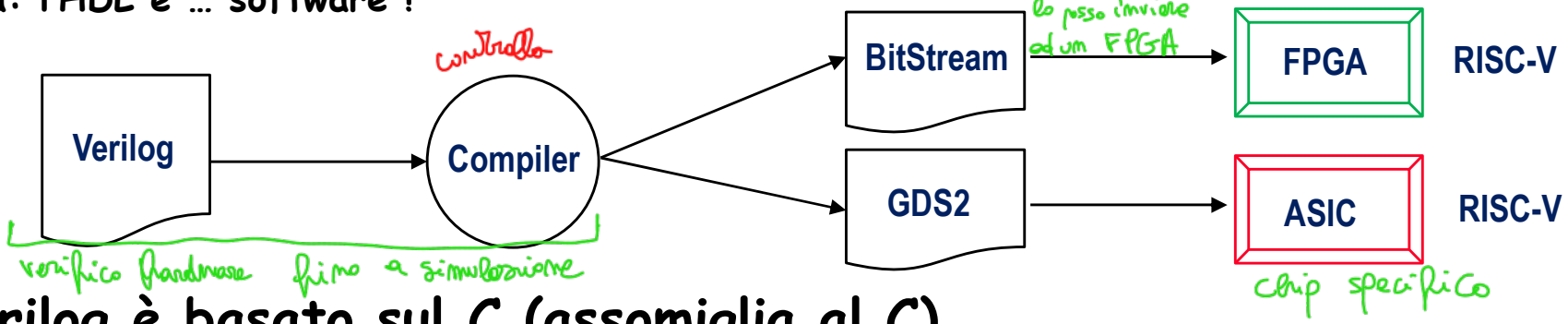
Lezione 4: Introduzione agli Hardware Description Languages (HDL)

(v. PHRV1, appendice A.4)

- **Il Verilog: obiettivi principali**
  - Descrivere un'architettura con un linguaggio formale (es. come il C per gli algoritmi)
  - Standardizzare tale linguaggio per un'ampia adozione (Verilog-95→Verilog-2001→IEEE-1364-2005→IEEE-1800-2009→IEEE-1800-2017)
  - L'attuale standard è chiamato 'SystemVerilog' (IEEE-1800-2017)
- **Vantaggi di un linguaggio di descrizione dell'hardware (HDL)**
  - Permette di simulare il comportamento di un'architettura, eliminarne i bug
  - La rete progettata viene compilata («sintetizzata») generando automaticamente hardware (molto più veloce e con consumi energetici estremamente più bassi rispetto al solo software)
  - Nota: l'HDL è … software !

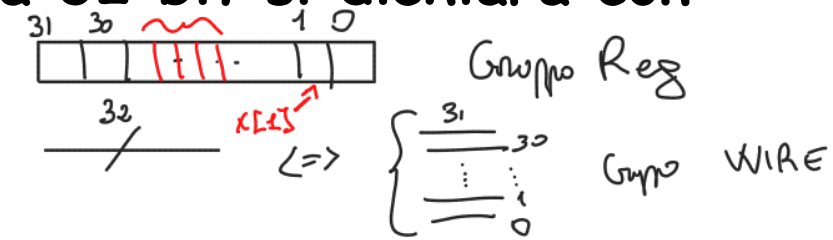


- **Il Verilog è basato sul C (assomiglia al C)**
  - Permette di costruire dalle reti usando la programmazione strutturata
  - Il VHDL (standard IEEE-1067-2019) assomiglia al linguaggio Ada
- **La differenza fondamentale Verilog vs. C**
  - Il C descrive un'evoluzione temporale di un algoritmo che usa risorse computazionali «fisse», mentre il Verilog descrive un'estensione spaziale di un sistema con risorse potenzialmente tutte operanti in parallelo

Verilog: Tipi di dato e Operatori

- **Tipi di dato**
  - **wire** indica un segnale (che viaggia su un filo, quindi in hardware è uno «wire»)
  - **reg** (register) indica una variabile che memorizza un valore (un registro a 1 bit)
    - Un reg non necessariamente corrisponde ad un registro fisico, anche se spesso è così

- **Un registro o wire a 32 bit si dichiara con**
  - wire[31:0] Y
  - reg[31:0] X



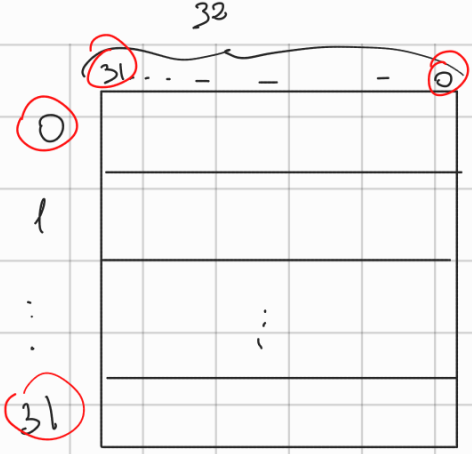
  - Si può fare accesso ad un sottoinsieme di tali bit indicandolo con la notazione [starting_bit : ending_bit]

- **Si possono combinare più registri per formare una struttura "register file" ovvero una "memoria" ***
  - reg[31:0] registerfile[0:31]
  - Si fa accesso al singolo registro con la classica notazione registerfile[num_registro]

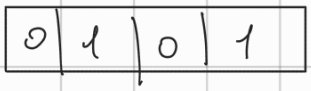
* Nota: Memorie e Reti sequenziali verranno affrontate in una lezione successiva.
Per ora si può assumere che una memoria o registro sia l'equivalente di una "lavagna"

Register file [0:31]

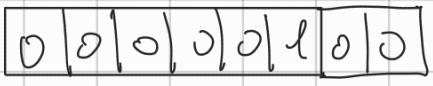
32 Registri che vanno da 0 a 31



4'b 0101



8'h4



Verilog: valori e costanti

- I possibili valori di un registro o un filo sono:
  - 0 o 1 per rapprentare i valori logici falso o vero rispettivamente
  - X per rappresentare un "don't care" (non-specificato) o un valore non noto
  - Z per rappresentare lo stato di alta impedenza (filo "scollegato" o uscita "libera")

  (dalla Teoria)

- I nomi simbolici delle costanti si dichiarano con parameter
  - parameter A=1

  nome simbolico dei valori

  {parameter, nome simbolico costante}

- I valori delle costanti possono essere specificati in varie basi
  - (la <base> 2,8,10,16 si indica rispettivamente con b,o,d,h)

    Notazione: <n.bit>" ' "<base><valore>

    (binario, ottale, decimale, esadecimale)

    Esempio:

    (apice normale, semplice)
  - 4'b0101 indica una costante binaria a 4 bit che vale 5
  - 8'h4 indica una costante binaria a 8 bit che vale 4

    (# bit costante)

    cosa ho generato? → costante estesa su 8 bit

    {3{x[1]}}

- Repliche e Concatenazioni
  - {x{ bitfield }} replica i bit indicati x volte, es. {3{2'b01}} crea 010101

    costante 01 ripetuta 3 volte
  - {bitfiled1, bitfield2} concatena i due bitfield, es. {A[7:4],B[5:2] }

    4 bit da reg. A

    4 bit del reg. B posiz. tra 5 e 2

    es. concatena frettine di registri

Verilog: operatori (1/2)

- **In gran parte simile al linguaggio C**

Relational Operators

Operator	Application
<	a < b // is a less than b? // return 1-bit true/false
>	a > b // is a greater than b?
>=	a >= b // is a greater than or // equal to b
<=	a <= b // is a less than or // equal to b

Arithmetic Operators

Operator	Application
*	c = a * b ; // multiply a with b
/	c = a / b ; // int divide a by b
+	sum = a + b ; // add a and b

-	diff = a - b ; // subtract b
	// from a

%	amodb = a % b ; // a mod(b)

Logical Operators

Operator	Application

&&	a && b ; // is a and b true?
	// returns 1-bit true/false

||	a || b ; // is a or b true?
	// returns 1-bit true/false

!	if (!a) ; // if a is not true
	c = b ; // assign b to c

AND

OR

NOT

Verilog: operatori (2/2)

- Gli operatori sono in gran parte simili a quelli del linguaggio C

Nota: evidenziate in blu le parti con comportamento diverso rispetto al C

Equality and Identity Operators

Operator	Application
=	`c = a ; // assign a to c`

==	`a == b; //equal ?`
	`//0==0→1,1==1→1,0==1→0,`
	`// 0==X→X,1==X→X`
	`//e.g., `hx == `h5 returns X`

!=	`c != a ; // is c not equal to`
	`// a, retruns 1-bit true/`
	`// false logic equality`

===	`a === b; //identical? (returns 1 bit)`
	`//also 0,1,X,Z must be identical to get 1`
	`//`
	`//e.g., `hx === `h5 returns 0`

!==	`a !== b; //not identical? (returns 1 bit)`
	`//also 0,1,X,Z must be identical to get 1`
	`//`
	`//e.g., `hx !== `h5 returns 1`

Unary, Bitwise and Reduction Operators

Operator	Application
+	Unary plus & arithmetic(binary) addition
-	Unary negation & arithmetic (binary) sub-traction
=&	b = &a ; // AND all bits of a *
=|	b = |a ; // OR all bits *
=^	b = ^a ; // Exclusive or all bits of a*
~&, ~|, ~^	NAND, NOR, EX-NOR all bits to-gether *

~, &, |, ^	bit-wise NOT, AND, OR, EX-OR
	`b = ~a ; // invert a`
	`c = b & a ; // bitwise AND a,b`
	`e = b | a ; // bitwise OR`
	`f = b ^ a ; // bitwise EX-OR`

~&, ~|, ~^	bit-wise NAND, NOR, EX-NOR
	`c = a ~& b ; d = a ~| b ;`
	`e = a ~^ b ;`

Shift Operators and other Operators

Operator	Application
<<	`a << 1 ; // shift left a by 1-bit`
>>	`a >> 1 ; // shift right a by 1`
?:	`c = sel ? a : b ; /* if sel is true c = a, else c = b , ?: ternary operator */`
{}	`{co, sum } = a + b + ci ; /* add a, b, ci assign the overflow to co and the re-sult to sum: operator is called concatenation */`
{{}}	`b = {3{a}} /* replicate a 3 times, equivalent to {a, a, a} */`

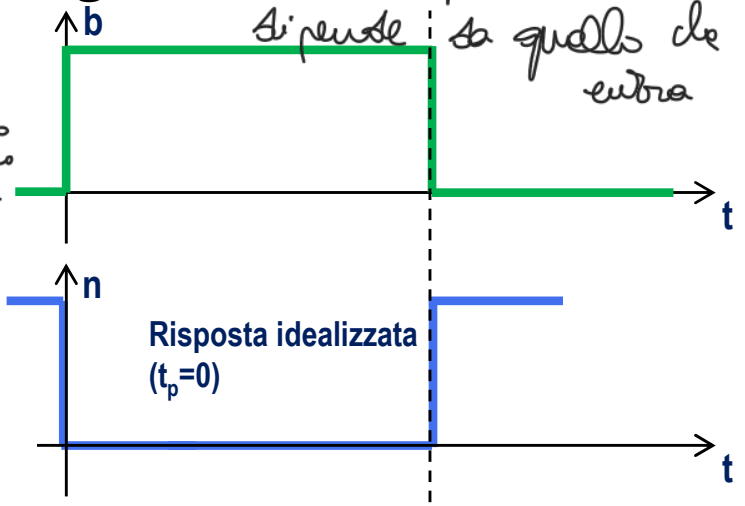
* Opera riducendo tutti i bit a uno solo:
in altri termini opera "orizzontalmente"

Verilog: Reti Combinatorie e Timing associato

- Una rete combinatoria si indica con «**assign**» seguita dall'equazione booleana:

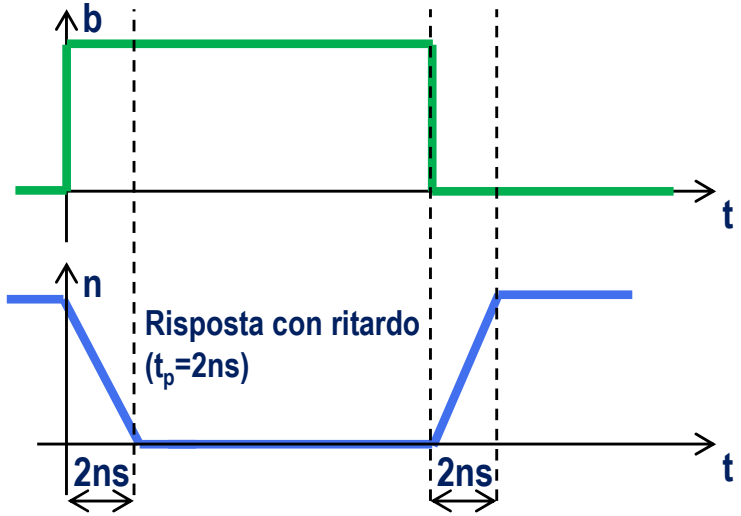
assign

n=~b



- Per indicare il tempo richiesto da tale operazione («tempo di propagazione») si aggiunge questa notazione:

assign #2 n=~b



  - '#2' indica un ritardo di 2 ns ('ns' è il default – ma si può cambiare la scala)

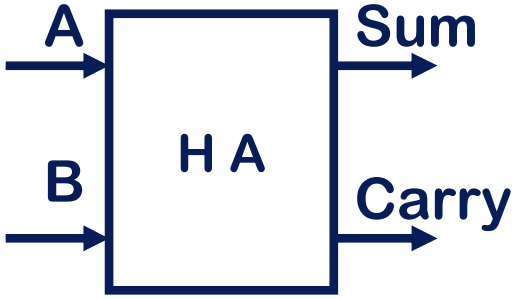
- Nota: "assign" indica quindi un **assegnamento continuo**
  - Ad ogni variazione delle variabili a destra, viene generata (col ritardo specificato) una variazione delle variabili a sinistra del segno di '='
  - Inoltre ciò avviene **in parallelo** ad ogni altro assign

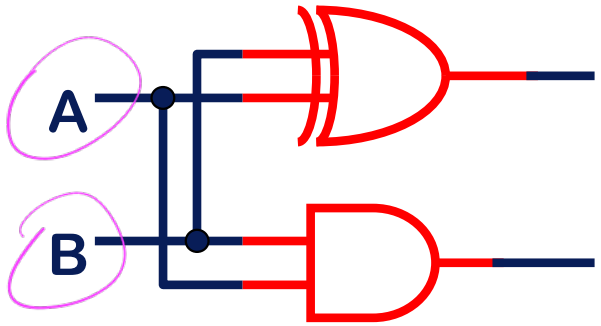
Esempio: Half Adder in Verilog

senza Carry IN

SOMMA = XOR dei due ingressi

- Dichiaro innanzitutto un modulo con determinati nomi per gli ingressi e le uscite:





ho 1 resto solo quando entrambi i bit in ingresso sono a 1

ingressi | uscite | comportamento | definizione di modulo

scatolina

argomenti

```
module half_adder(a,b, sum,carry);
    input  a,b;          //ingressi su 1-bit
    output sum,carry;    //uscite su 1-bit
    assign sum = a ^ b;  // a XOR b
    assign carry = a & b; // a AND b
endmodule
```

resto esempio half adder reti combinatorie ecc

ho descritto il modulo

ne descrivo il comportamento, 2 reti logiche

- Verilog keywords: input, output, module/endmodule

Assegnamento procedurale: initial e always

- Quando si assegna un valore ad un registro (keyword 'reg'), l'assegnamento DEVE essere fatto:
  - i) All'inizio della simulazione (caso tipico del segnale di reset) tramite la keyword 'initial', es.:

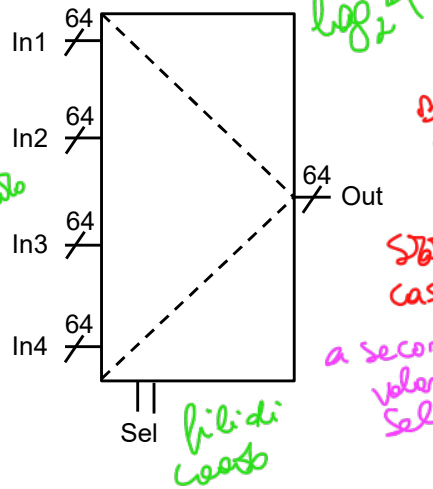
```
reg reset;
initial reset=0; //initial viene eseguito nel primo timestep di simulazione
```

  - ii) Al verificarsi di un evento (es. al variare di segnale) tramite la keyword 'always':

```
reg out; wire in1, in2;
always @(in1 or in2) out=1; //always → check ad ogni timestep in cui variano i segnali indicati nella lista di sensibilità @(...)
```

- Assegnamento non-bloccante ('<=') e bloccante ('=')
  - Negli assegnamenti procedurali posso avvalermi di due diverse sintassi:
    - Assegnamento NON-BLOCCANTE: es. `out1<=1; out2<=0; // i due registri vengono assegnati in parallelo`
    - Assegnamento BLOCCANTE: es. `out1=1; out2=0; // viene prima assegnato out1 e poi out2`
  - Nota: gli assegnamenti all'interno di uno stesso comando initial o always possono essere racchiusi in un cosiddetto «blocco begin-end»; es. `begin out1<=1; out2<=0; end`
- Modello di esecuzione: ad ogni timestep di simulazione
  - Vengono valutate tutte le espressioni bloccanti e assegnate (in successione se all'interno di begin-end)
  - Vengono valutate le parti a destra degli assegnamenti non-bloccanti e poi assegnate tutte in parallelo

Esempio: multiplexer a 4 ingressi di 64 bit




```
module Mult4to1(In1,In2,In3,In4,Sel, Out);
  input[63:0] In1,In2,In3,In4; //four 64-bit inputs
  input[1:0] Sel;              //selection signal
  output[63:0] Out; reg[63:0] Out; //64-bit output
  always @(In1 or In2 or In3 or In4 or Sel)
  casex(Sel)  //4->1 mux
     0: Out <= In1;
     1: Out <= In2;
     2: Out <= In3;
     default: Out <= In4;
  endcase
endmodule
```

Notazione per indicare più bit

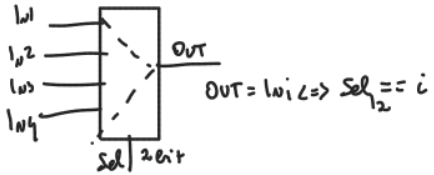
Notazione "reg":
memoria interna (registro)

Notazione «always @(...)» :
qualcosa accade quando vengono variati i seguenti segnali indicati in "@(.....)"

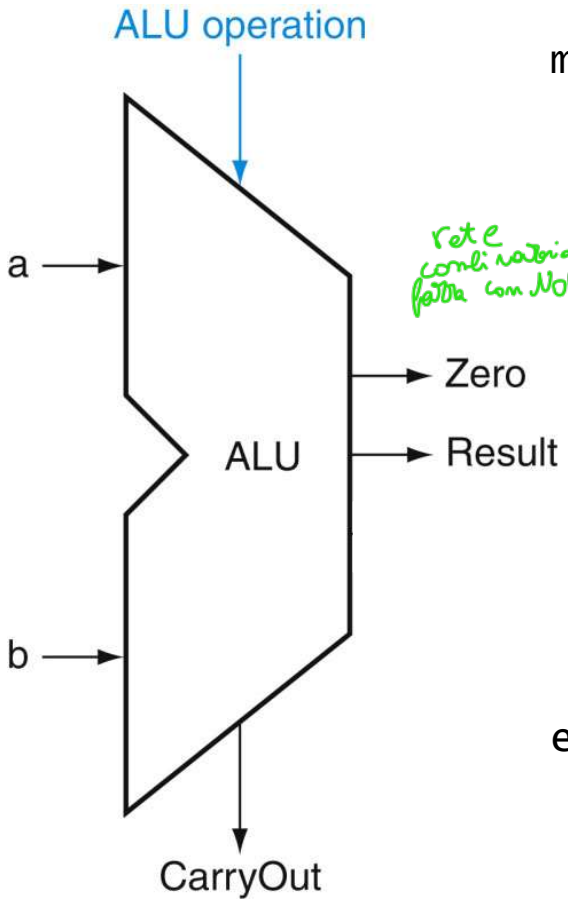
Notazione «casex/endcase»
funziona come il «switch» del linguaggio C

- **Verilog keywords:** reg, always @(...), casex/endcase

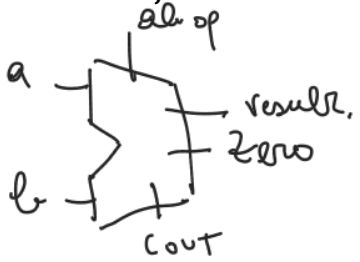
Nota1: il registro non c'era nella definizione iniziale del multiplexer; è stato messo per comodità di avere un valore stabile in uscita



Esempio: ALU




```
module alu(a,b,aluoperation,  result,zero, carryout);
   input[63:0] a, b; input[2:0] aluoperation;
   output[63:0] result; reg[63:0] result;
   output zero, carryout;
   assign zero = (result==0);
   always @(aluoperation or a or b) //re-evaluate if these change
      casex (aluoperation)
         0: result <= a & b;
         1: result <= a | b;
         2: result <= a + b;
         6: result <= a - b;
         7: result <= a<b ? 1:0;
         default: result<=0; //default to 0, should not happen
      endcase
endmodule
```



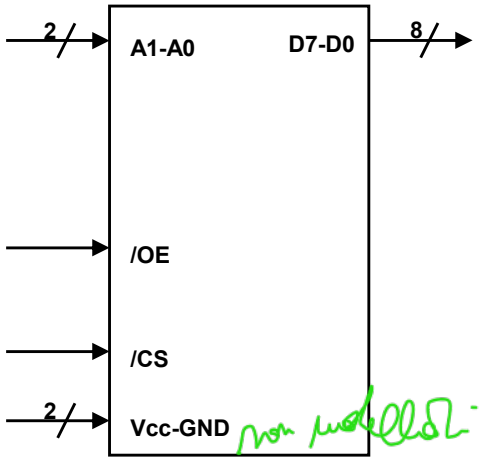
Nota1: sul Patterson viene fatto il caso con 4 bit di ALUctl: noi ne useremo 3 (il che implica che non vedremo il caso della 'nor' della fig. A.5.15).

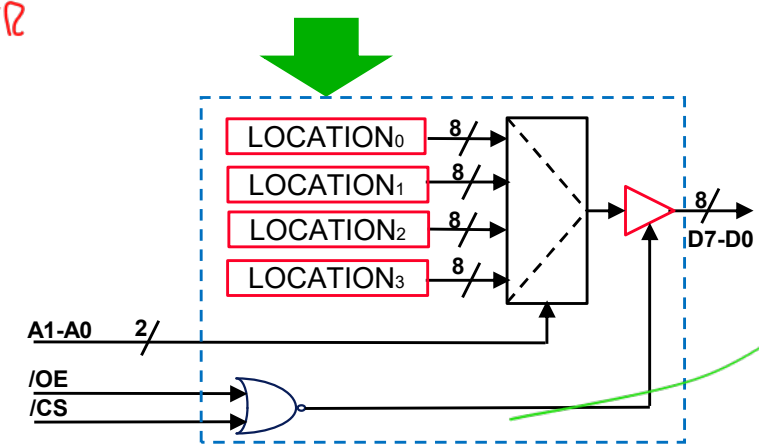
Esempio: ROM 4x8

```
module ROM(d7_d0, address, cs_, oe_);
  input  [1:0]  address;
  output [7:0]  d7_d0;
  input         cs_;
  input         oe_;

  wire alfa=~(cs_|oe_);
  assign d7_d0=
    (alfa==0)? 'BZZ : LOCATION(address);

  function [7:0] LOCATION;
   input [1:0] address;
   casex(address)
    0:       LOCATION = 0;
    1:       LOCATION = 1;
    2:       LOCATION = 2;
    3:       LOCATION = 3;
   endcase
  endfunction
endmodule
```

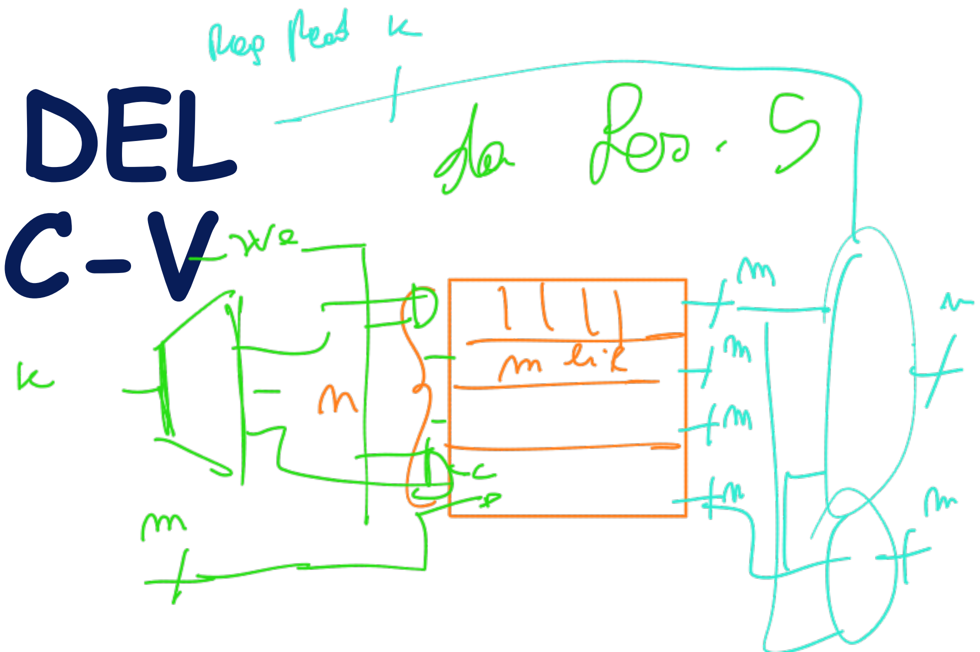




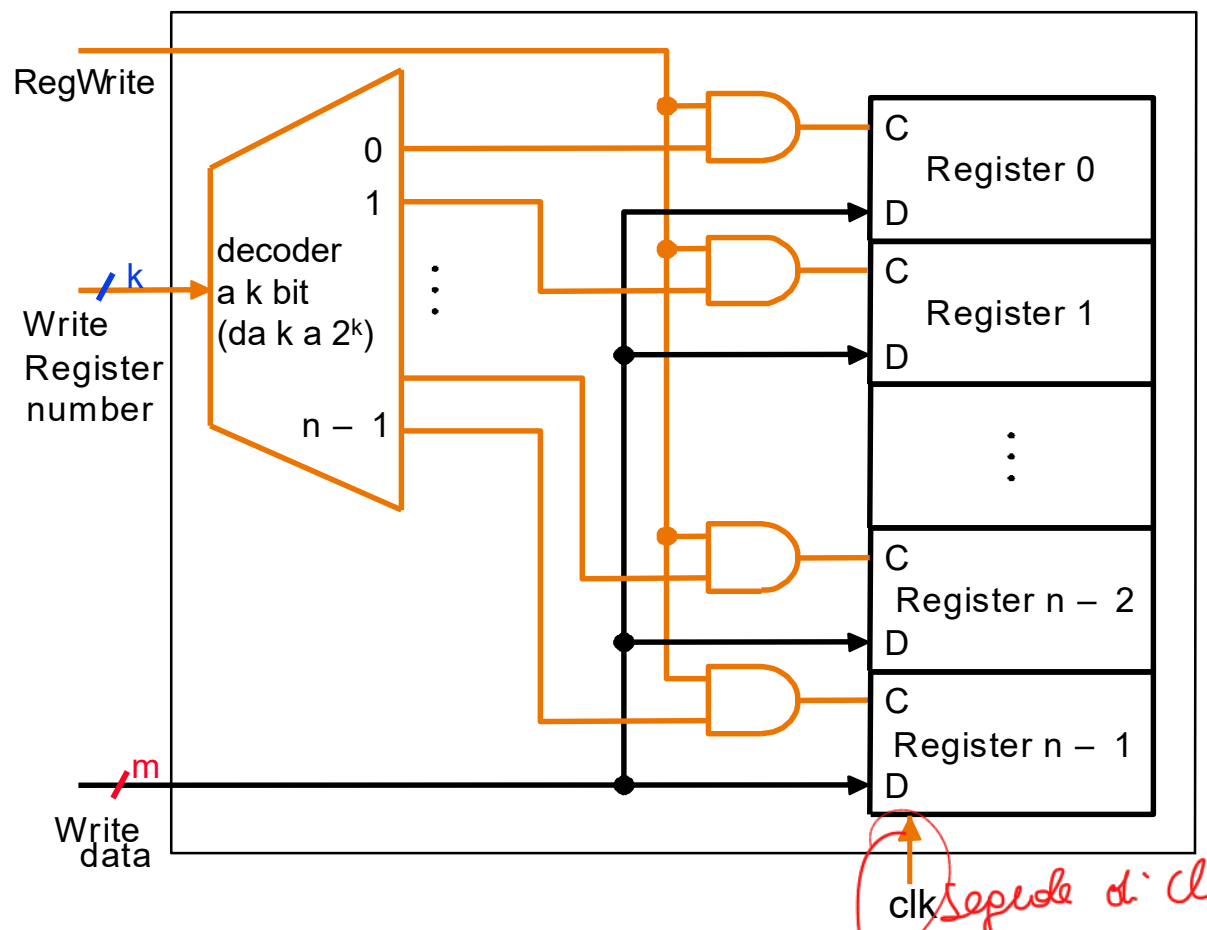
- Verilog keyword: function

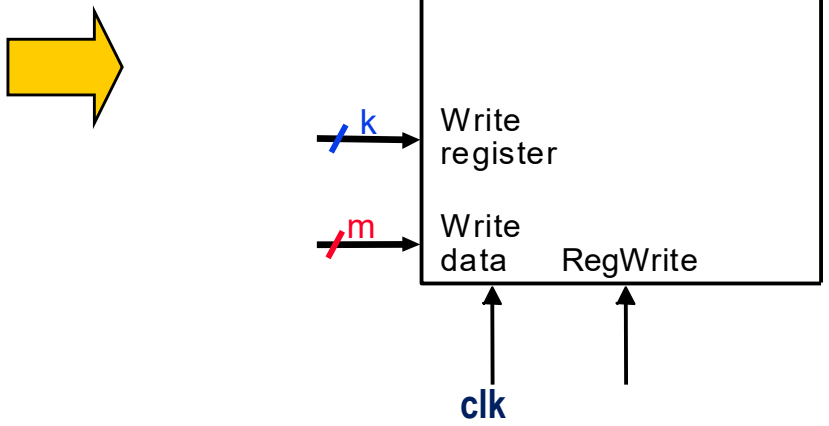
REALIZZAZIONE DEL PROCESSORE RISC-V

(V. PHRV1, APPENDICE A.8, CAP. 4.3)



Register File: logica di scrittura





n = numero di registri disponibili (e.g. 32)

k = $\log_2(n)$ (e.g. 5)

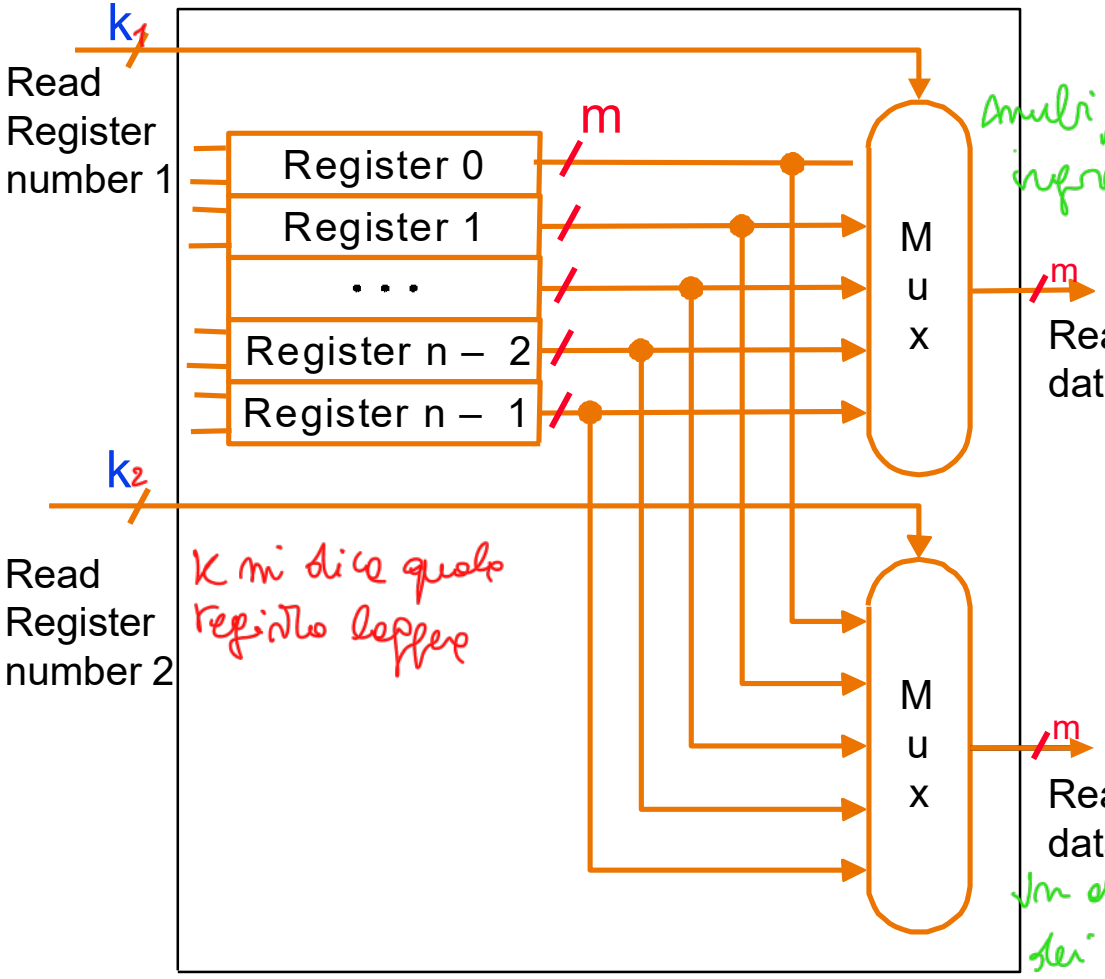
m = larghezza dei registri (e.g. 64)

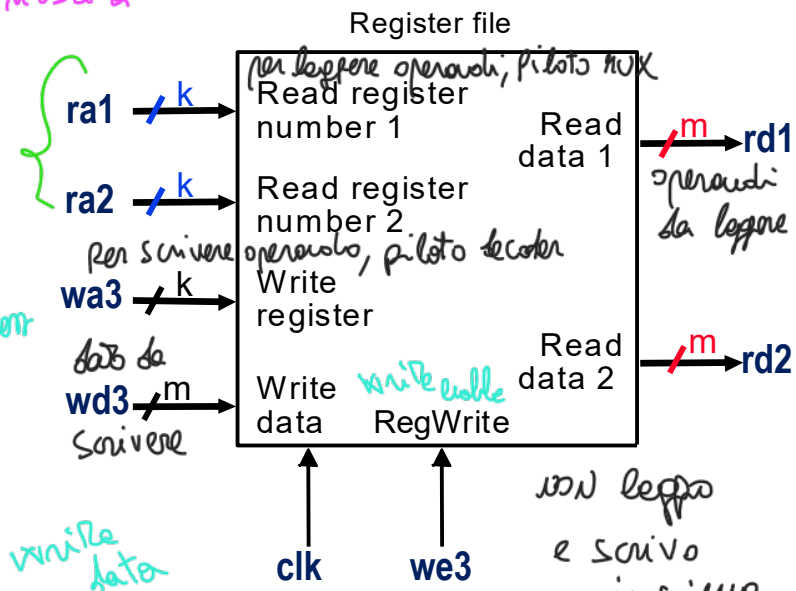
Nota: la struttura di un registro verrà esaminata in dettaglio in una lezione successiva

Nota: sul Patterson viene dato per implicito il segnale clk.

Register File: logica di lettura

- Si usano i flip-flop di tipo D





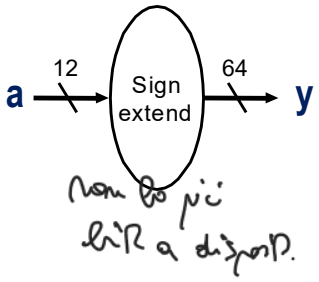

```
module regfile(clk, we3, ra1, ra2, wa3, wd3,   rd1, rd2);
   input clk,we3; input[4:0] ra1, ra2, wa3; input[63:0] wd3;
   output[63:0] rd1, rd2; reg[63:0] rd1, rd2;
   reg[63:0] rf[0:63];
   // three ported register file: read two ports
combinationally
   // write third port on falling edge of clk; register 0
hardwired to 0
   always @(negedge clk) if (we3) rf[wa3] <= wd3;
   always @(ra1) rd1 <= (ra1 != 0) ? rf[ra1] : 0;
   always @(ra2) rd2 <= (ra2 != 0) ? rf[ra2] : 0;
Endmodule
```

- Register File $\quad k = \log_2 n \quad n = \text{num di reg}$
- Ra_1 e Ra_2, ingressi a k bit per leggere operandi
- Un ingresso a k bit per scrivere dato Wa_3
- m fili per dato Wd_3 da scrivere We_3 per controllo
- uscita da m bit per operandi da leggere

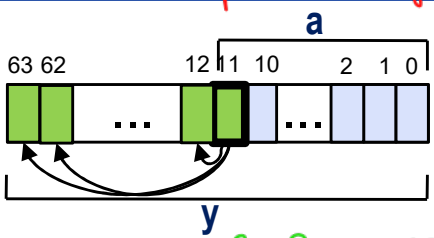
Altra logica sparsa e Memoria principale

- **Nel caso della beq, l'offset di salto può essere sia positivo che negativo e specificato su 12 bit: occorre una rete di estensione dell'operando con segno da 12 a 64 bit**
  - **Questo è anche il caso della load e della store**
  - **Inoltre: rete per moltiplicare per due (branch)**

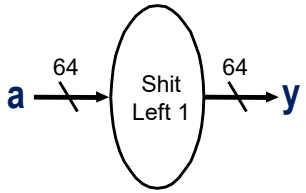
Gen. Immediate Unit



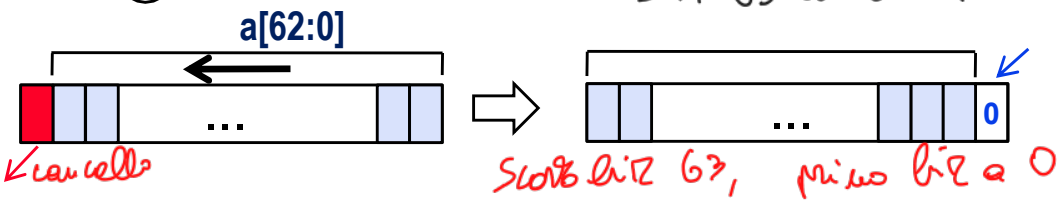

```
module signext(input [11:0] a,
               output [63:0] y);
  assign y = {{52{a[11]}}, a};
endmodule
```



Multiply-by-2 Unit

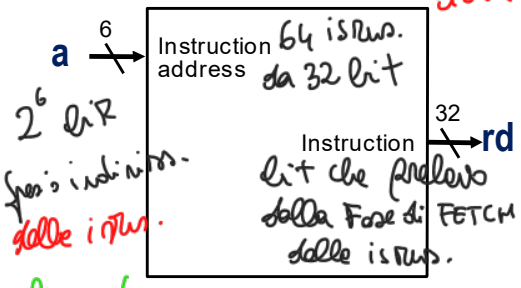



```
module sl1(input [63:0] a, output [63:0] y);
  // shift left by 1
  assign y = {a[62:0], 1'b0};
endmodule
```



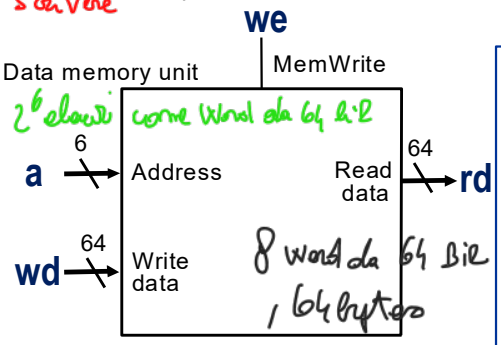
- **La memoria istruzioni e la memoria dati possono essere viste così:**

Instruction memory




```
module imem(input [5:0] a,
            output [31:0] rd);
  reg [31:0] ROM[0:63]; //256 Bytes
                        // (i.e., 64 words)
  assign rd = ROM[a];
endmodule
```

Data memory unit




```
module dmem(input clk, input 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)
  //a=byte aligned,RAM=dword alig.
  assign rd=RAM[a[5:3]];
  always @(posedge clk)
  if (we) RAM[a[5:3]] <= wd;
endmodule
```

Nota: sul Patterson la memoria dati ha anche il segnale MemRead (qui lo eliminiamo per semplicità)

nel Processore serve operando immediato

Serve immediato P. es. con

1) LOAD / WRITE 100 (X5) X5 + 100 indirizzo Base + OFFSET

100 ↑ OFFSET ↑ valore Base

2) Beq X5, X6, LABEL 100 el

PC + 100 è a program counter + 100

Branch equal, salto confronto 2 registri X_5 e X6 e poi salto di 100 elementi

3) ADDI $X_5 = X_5 + 100$ ADD immediato, aggiungo al reg. X_5, 100

Per usare l'immediato

- Incluso +100 nell'istruzione, non ho tutti i bit a disposizione, ne uso 12, ma il nostro processore è a 64 bit, devo estendere il 100 col segno

Estensione di bit = Estendo con Segno
- Se positivo replico tutti 0, altrimenti replico tutti 1

Mantengo così il segno

Immediato è su 12 bit, il mio processore è a 64 bit, serve della logica per estendere i 12 bit a 64

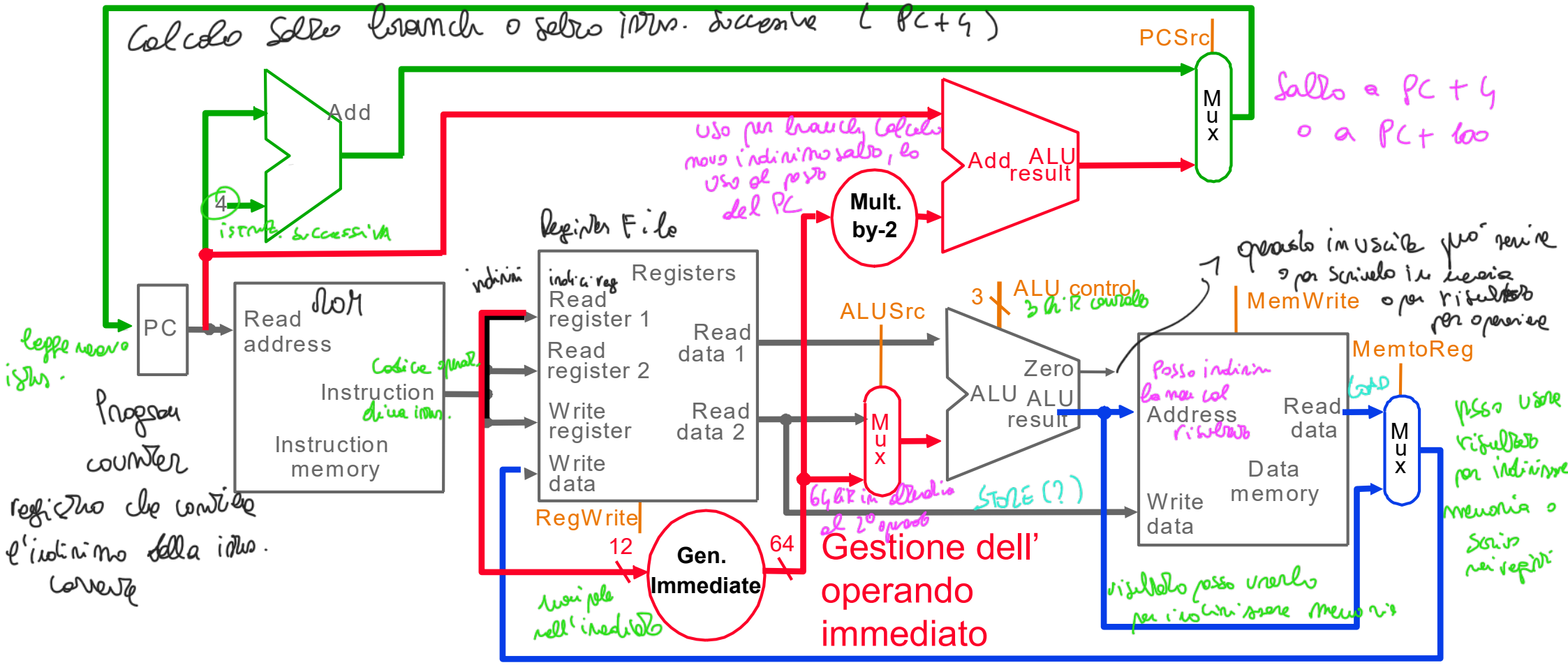
Le istruzioni sono sempre a multipli di 2, nella branch, fa comodo fare moltiplicaz. • 2, che faccio shiftando verso SX (moltiplic.)

Processore esegue ciclo di Fetch in esecuzione

Costruzione del Datapath del processore RISC-V

Gestione indirizzo istruzione successiva

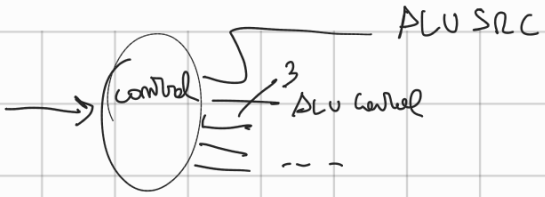
I segnali di controllo vengono pilotati in base a "op-code" e "funct" dell'istruzione:



Gestione del risultato della ALU o dato dalla memoria

- c.f. PHRV1 figura 4.11

*Nota: "Gen. Immediate" comprende estensione del segno e raggruppamento dei bits relativi all'operando immediato nei vari formati di istruzione (per ora vediamo solo la estensione del segno)

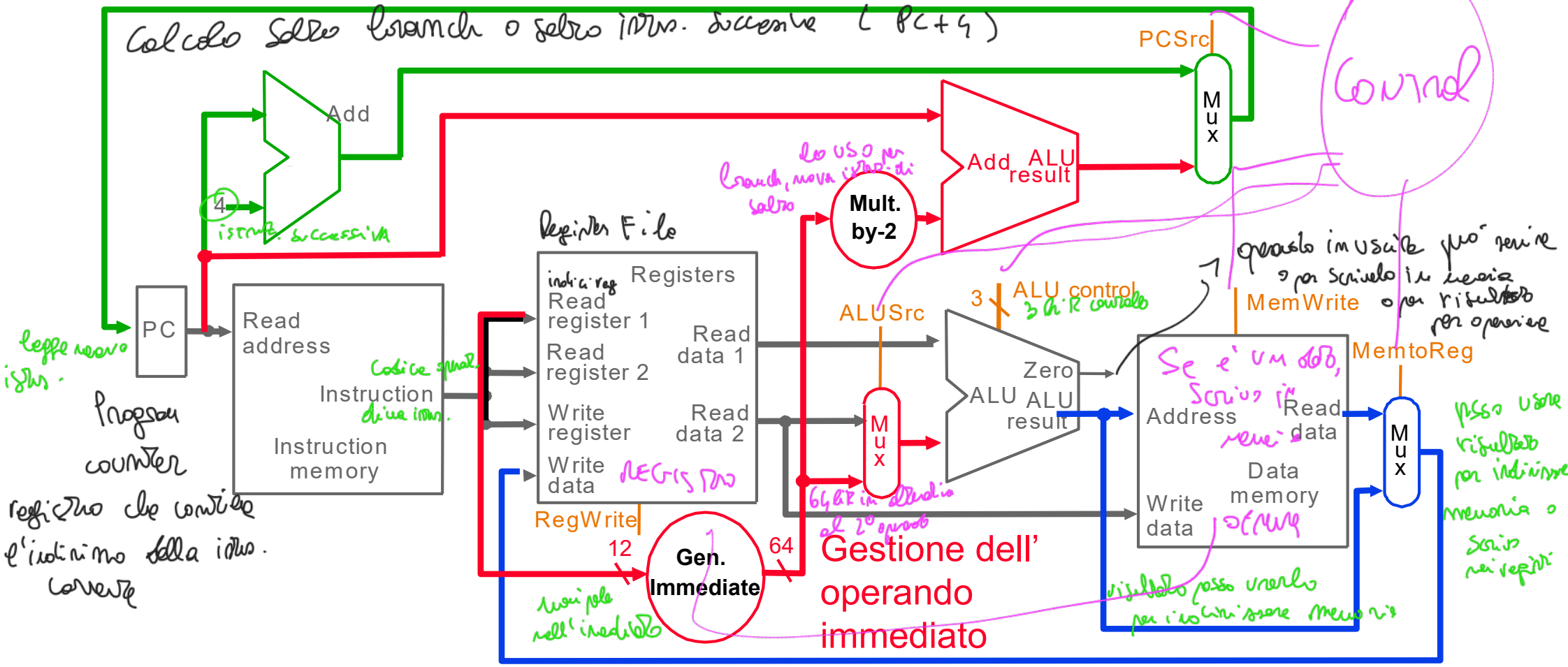


manca rete di controllo per generare controllo di lettura/scrittura, registri/memoria stati, pilotare multiplexer o le operazioni nell'alu

Costruzione del Datapath del processore RISC-V

Gestione indirizzo istruzione successiva

I segnali di controllo vengono pilotati in base a "op-code" e "funct" dell'istruzione:



Gestione del risultato della ALU o dato dalla memoria

- c.f. PHRV1 figura 4.11

*Nota: "Gen. Immediate" comprende estensione del segno e raggruppamento dei bits relativi all'operando immediato nei vari formati di istruzione (per ora vediamo solo la estensione del segno)