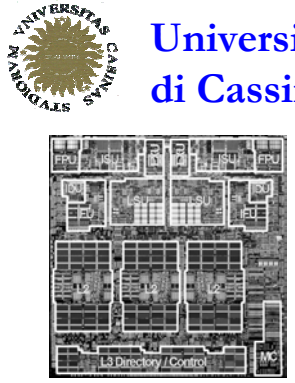


Università degli Studi di Cassino

Corso di Calcolatori Elettronici II

Unità Logico-Aritmetica

Anno Accademico 2004/2005

Francesco Tortorella

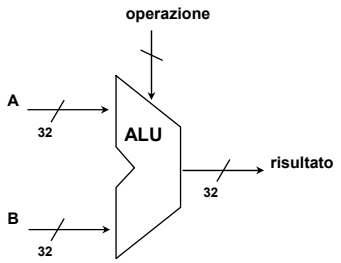
Progetto di una ALU

ALU *n. [**A**rthritic **L**ogic **U**nit or (rare) **A**rithmetic **L**ogic **U**nit]*
A random-number generator supplied as standard with all computer systems.

Stan Kelly-Bootle, *The Devil's DP Dictionary*

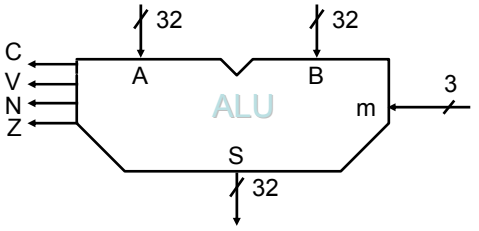
Compiti dell'ALU:

- esecuzione delle operazioni aritmetiche (addizioni, sottrazioni, moltiplicazioni,…)
- esecuzione delle operazioni logiche (AND, OR,…)
- produzione del risultato e modifica dei flag



Specifiche di progetto

- 2 ingressi da 32 bit
- 1 uscita da 32 bit
- almeno 6 operazioni possibili (add, sub, adc, and, or, not)
- gestione dei flag C, V, N, Z

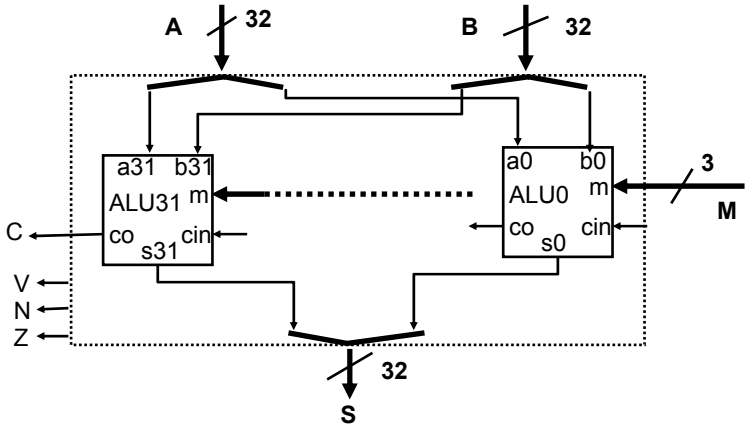


Quale approccio progettuale seguire ?

Forza bruta: rete combinatoria a 68 ingressi e 36 uscite.

Approccio top-down: decomposizione in sottoproblemi, impiego di componenti già noti.

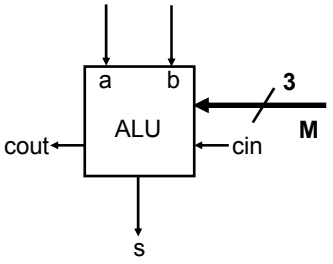
Decomponiamo l'ALU a 32 bit in 32 ALU da 1 bit (bit slice ALU)

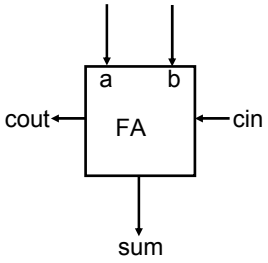


ALU da 1 bit

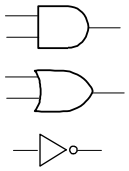
Come realizzarla ?

Quali tipi di componenti noti ci possono essere utili ?

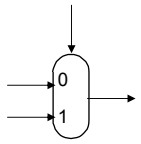




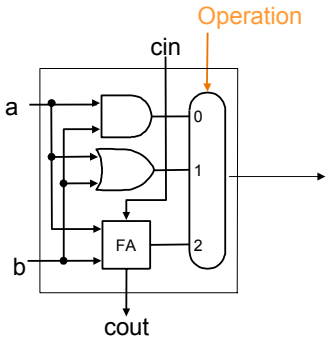
Full Adder

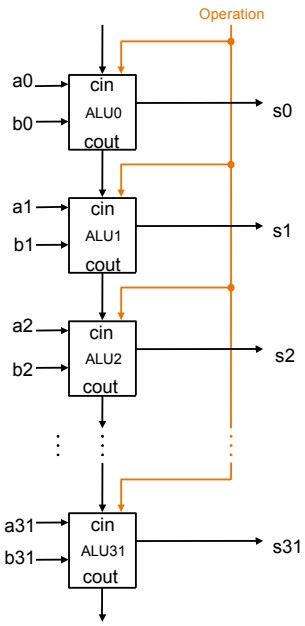


Porte logiche



Multiplexer



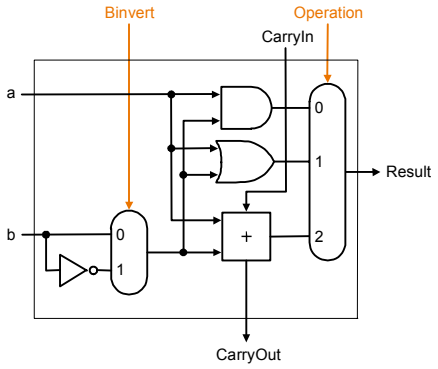


Operazioni possibili: somma, AND, OR.

Realizzazione della sottrazione

uso della rappresentazione per complementi

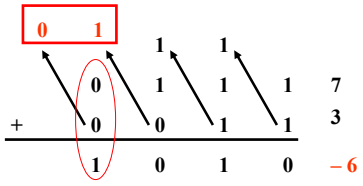
- A-B=A+$\overline{B}$
- $\overline{B}$ = !B+1

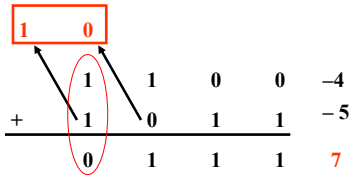


Gestione dell'overflow

- OVERFLOW: il risultato dell'operazione non è rappresentabile (> MAX o < MIN)
- Si verifica solo in presenza di operandi con lo stesso segno
- Criteri per la rilevazione:
  - ➔segno del risultato diverso dal segno degli operandi
  - ➔riporto entrante nel MSB diverso dal riporto uscente dal MSB

Esempio (a 4 bit):

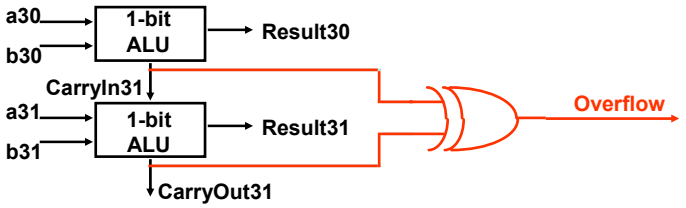




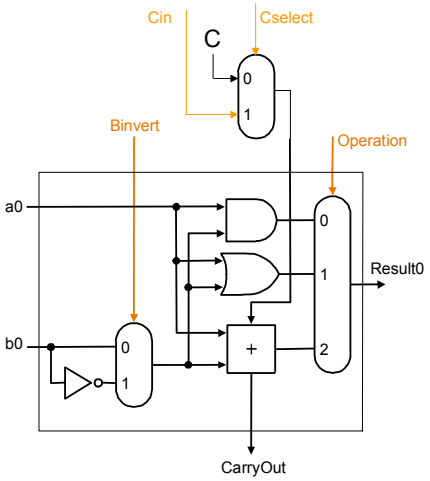
Gestione dell'overflow

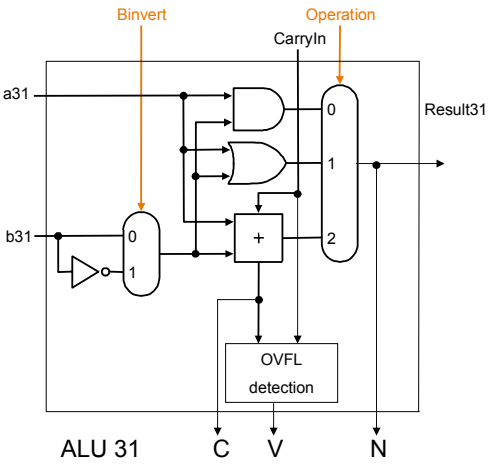
Overflow= CarryIn[31] XOR CarryOut[31]

Cin31	Cout31	Overflow
0	0	0
0	1	1
1	0	1
1	1	0



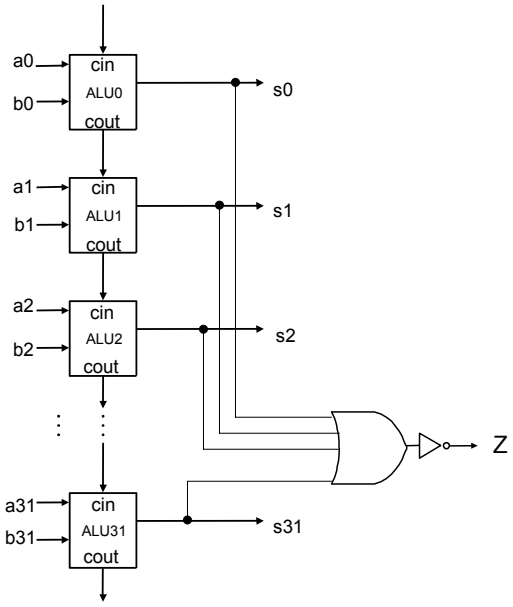
Modifiche da apportare alle ALU 0 e 31





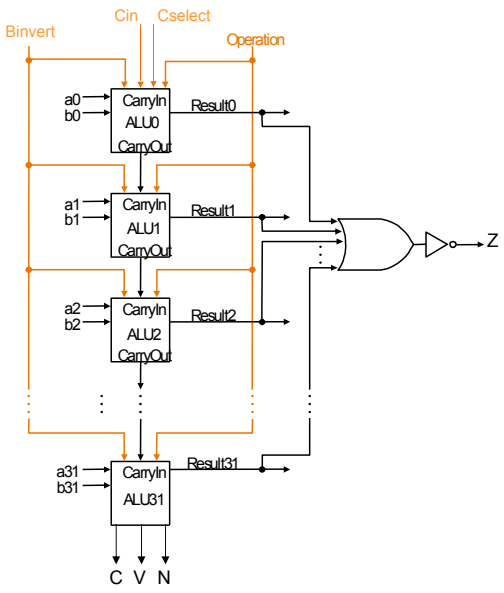
Gestione del flag Z

Il flag Z vale 1 quando il risultato è nullo!



Progetto complessivo

	Op	Binv	Cin	Csel
add	10	0	0	1
adc	10	0	X	0
sub	10	1	1	1
and	00	0	X	X
or	01	0	X	X
not	01	1	X	X

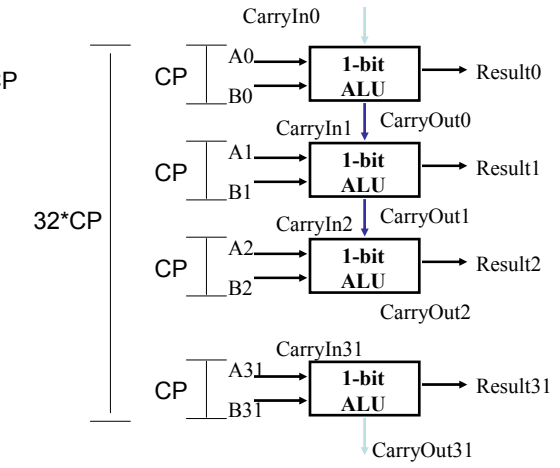


Valutazione del progetto

Quanto vale il percorso critico? n*CP

Problema:

il ripple carry adder è lento.



Valutazione del progetto

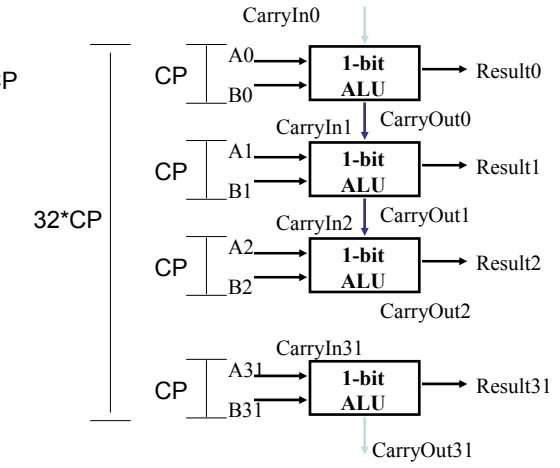
Quanto vale il percorso critico? n*CP

Problema:

il ripple carry adder è lento.

Esiste un'alternativa per la realizzazione dell'adder ?

Una Rete Combinatoria !



Due soluzioni estreme

Ripple carry adder
- modulare
- progetto semplice
- lento

Adder tramite rete combinatoria
- monolitico
- progetto complesso
- veloce

Esiste una terza via ?

Esaminiamo il problema del carry:

$$c_1 = b_0c_0 + a_0c_0 + a_0b_0$$
$$c_2 = b_1c_1 + a_1c_1 + a_1b_1$$
$$c_3 = b_2c_2 + a_2c_2 + a_2b_2$$
$$c_4 = b_3c_3 + a_3c_3 + a_3b_3$$

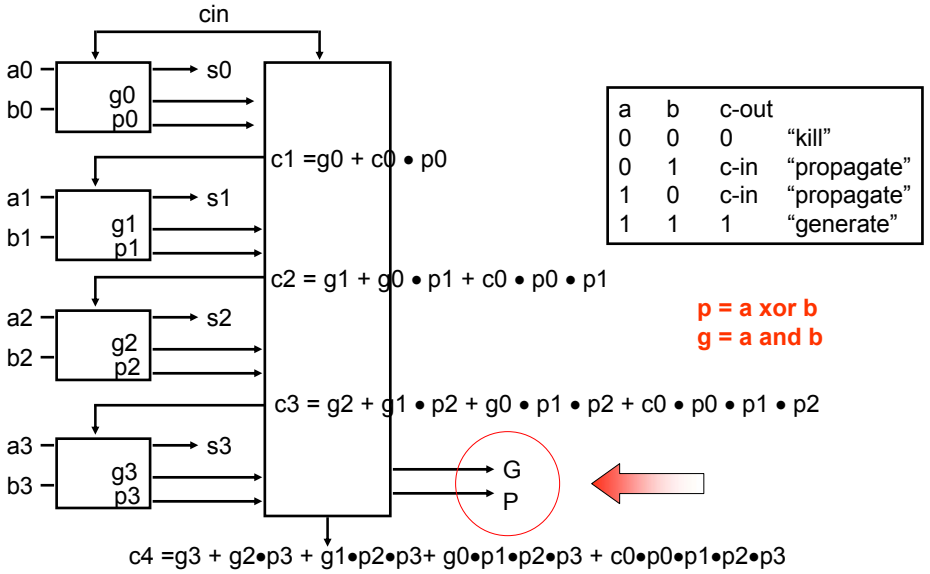
Quando viene generato un carry ?

$$g_i = a_i\, b_i$$

Quando viene propagato un carry ?

$$p_i = a_i + b_i$$

Adder con carry look-ahead a 4 bit



a	b	c-out	
0	0	0	"kill"
0	1	c-in	"propagate"
1	0	c-in	"propagate"
1	1	1	"generate"

p = a xor b
g = a and b

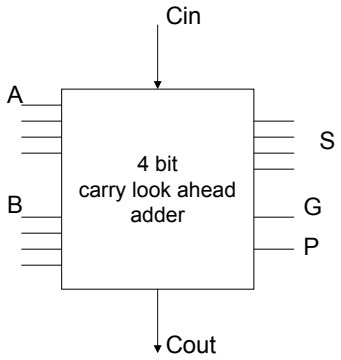
Adder con carry look ahead a 32 bit (1/3)

E' possibile generalizzare all'adder a 4 bit quanto fatto per l'adder ad 1 bit.

Cout =
g3 + g2•p3 + g1•p2•p3+ g0•p1•p2•p3 + Cin•p0•p1•p2•p3

G P

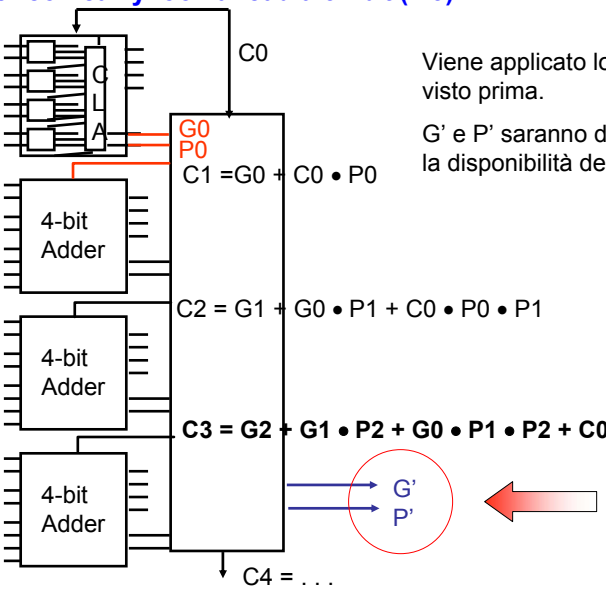
Cout=G + Cin • P



G e P sono disponibili dopo un intervallo pari a 2Tp (ritardo di porta) dall'istante in cui sono disponibili gli ingressi gi e pi.

E' possibile utilizzare la stessa struttura per un adder a 16 bit...

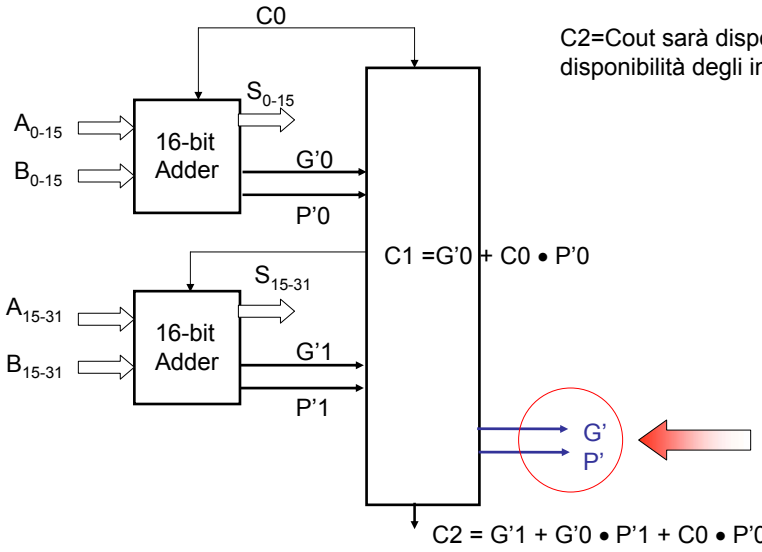
Adder con carry look ahead a 32 bit (2/3)



Viene applicato lo stesso modello visto prima.

G' e P' saranno disponibili 2Tp dopo la disponibilità degli ingressi Gi e Pi.

Adder con carry look ahead a 32 bit (3/3)



C2=Cout sarà disponibile 2Tp dopo la disponibilità degli ingressi G'i e P'i.

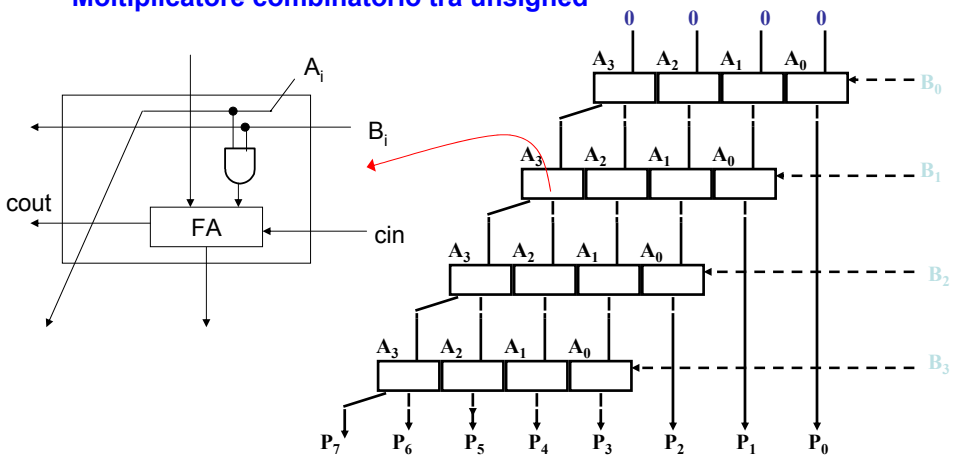
La moltiplicazione tra unsigned

- Operazione svolta a mano:

Moltiplicando	**1000**
Moltiplicatore	**1001**
	1000
	0000
	0000
	1000
Prodotto	**01001000**

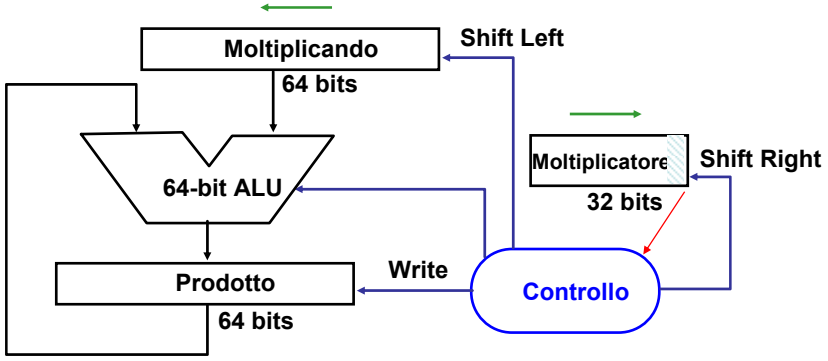
- m bits x n bits = prodotto a m+n bit
- In base 2 l'operazione è semplice:
  - 0 => aggiungi 0 (0 x moltiplicando)
  - 1 => aggiungi una copia (1 x moltiplicando)

Moltiplicatore combinatorio tra unsigned



Il livello i-mo somma A*2^i se Bi=1

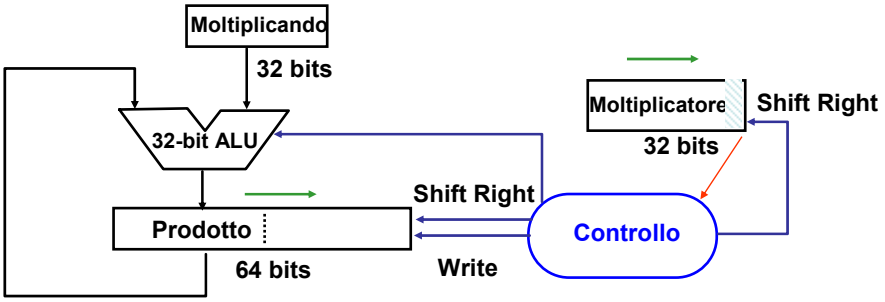
Moltiplicatore sequenziale tra unsigned (1/3)



- 1 colpo di clock per passo => 100 colpi di clock per moltiplicazione
- metà del moltiplicando è sempre 0 => l'adder da 64-bit è sprecato
- lo shift left del moltiplicando inserisce degli 0 in coda => i bit meno significativi del prodotto non cambiano, una volta calcolati

Shift right del prodotto, invece di shift left del moltiplicando?

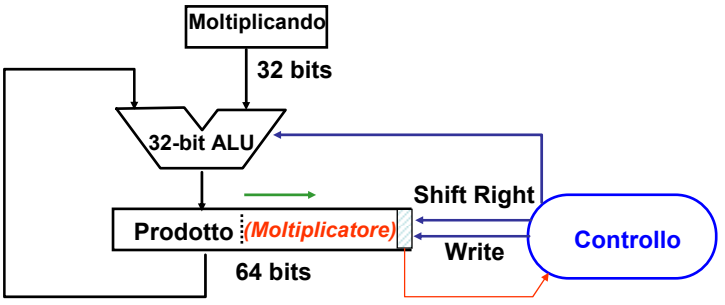
Moltiplicatore sequenziale tra unsigned (2/3)



- Registro del moltiplicando a 32 bit
- ALU a 32 bit
- Registro del prodotto a 64 bit

Il registro prodotto non utilizza parte del registro in cui potrebbe essere memorizzato il moltiplicatore

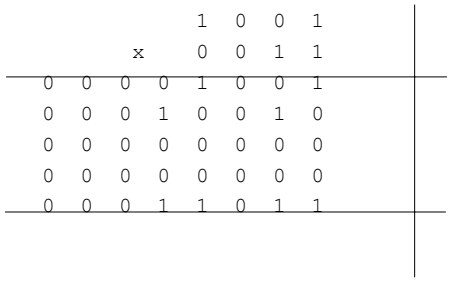
Moltiplicatore sequenziale tra unsigned (3/3)



E i numeri signed ?

Intero privo di segno

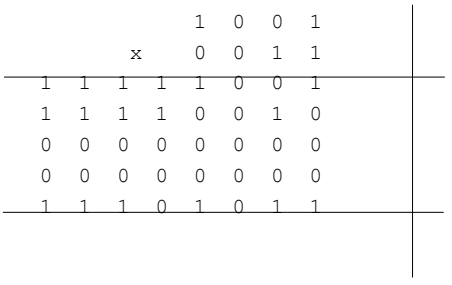
9 x 3 = 27



E i numeri signed ?

Intero con segno

-7 x 3 = -21



L'algoritmo è ancora corretto per moltiplicando con segno e moltiplicatore senza segno, a patto che si usi l'arithmetic shift.

La divisione tra interi

Operazione svolta a mano:

```
dividendo   1001010 | 1000  divisore
            1000    | 1001  quoziente
              1010
              1000
                10  resto
```

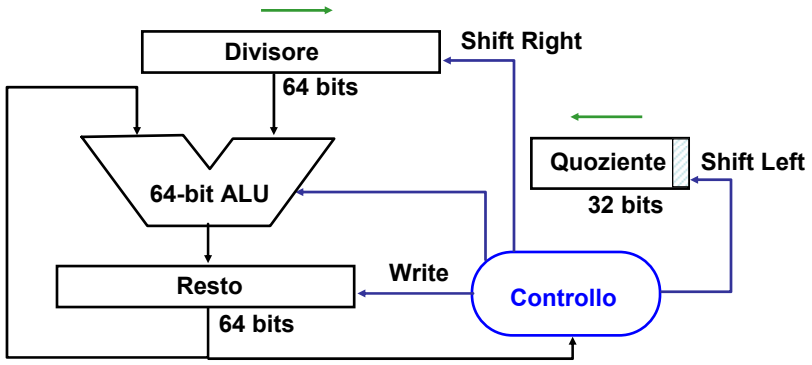
DD=DV*Q+R $Q=q_{n-1}b^{n-1}+q_{n-2}b^{n-2}+\ldots+q_1b^1+q_0$

L'algoritmo esegue n passi. All'i-mo passo:

$DD_i\text{-}DV*b^{n-i}$ { >=0 => $q_{n-i}=1$ $DD_{i+1}=DD_i\text{-}DV*b^{n-i}$; <0 => $q_{n-i}=0$ $DD_{i+1}=DD_i$

$DD_1=DD$ $R=DD_{n+1}$

Circuito divisore tra unsigned (1/3)



Registro divisore, reg. resto ed ALU a 64 bit

Registro quoziente a 32 bit

Metà del divisore è sempre a 0 => Si sprecano metà del registro divisore e dell'ALU.

Passi dell'algoritmo

(RR: registro resto RD: registro divisore RQ: registro quoziente)

0. dividendo -> RR, divisore -> RD_H, passi=0

1. [RR]-[RD] -> RR

2. If RR<0

[RR]+[RD]->RR (ripristino)

shl RQ

0 ->RQ_0

else

shl RQ

1 ->RQ_0

3. shr RD

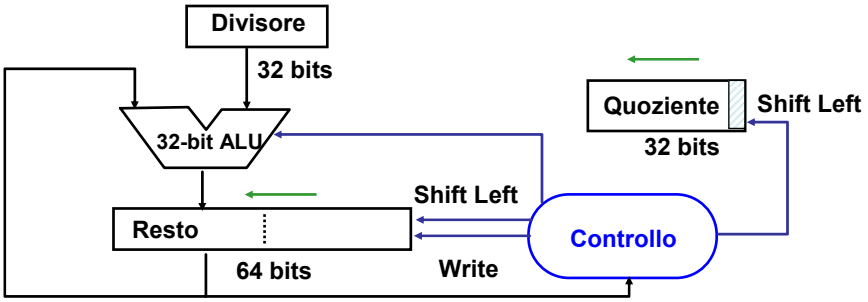
4. passi ++

5. If passi<33 goto 1.

Esempio: 7 / 2

	Passo	Quoziente	Divisore	Resto
0	Inizializzazione	0000	0010 0000	0000 0111
1	[RR]-[RD] -> RR	0000	0010 0000	1110 0111
	RR<0 => ripristino, shl RQ, 0 ->RQ0	0000	0010 0000	0000 0111
	shr RD	0000	0001 0000	0000 0111
2	[RR]-[RD] -> RR	0000	0001 0000	1111 0111
	RR<0 => ripristino, shl RQ, 0 ->RQ0	0000	0001 0000	0000 0111
	shr RD	0000	0000 1000	0000 0111
3	[RR]-[RD] -> RR	0000	0000 1000	1111 1111
	RR<0 => ripristino, shl RQ, 0 ->RQ0	0000	0000 1000	0000 0111
	shr RD	0000	0000 0100	0000 0111
4	[RR]-[RD] -> RR	0000	0000 0100	0000 0011
	RR>=0 => shl RQ, 1 ->RQ0	0001	0000 0100	0000 0011
	shr RD	0001	0000 0010	0000 0011
5	[RR]-[RD] -> RR	0001	0000 0010	0000 0001
	RR>=0 => shl RQ, 1 ->RQ0	0011	0000 0010	0000 0001
	shr RD	0011	0000 0001	0000 0001

Circuito divisore tra unsigned (2/3)



- Il resto viene traslato a sinistra, invece di fare lo shift right del divisore.
- In questo schema, il primo passo ([RR_H]-[RD]->RR_H) non può generare un 1 per il quoziente, altrimenti si avrebbe un Overflow ($DD/b^n - DV > 0$).
- Il registro resto non utilizza una parte che potrebbe ospitare il quoziente

Passi dell'algoritmo

(RR: registro resto RD: registro divisore RQ: registro quoziente)

0. dividendo -> RR, divisore -> RD, passi=0

1. shl RR

2. [RR_H]-[RD] -> RR_H

3. If RR<0

[RR_H]+[RD]->RR_H (ripristino)

shl RQ

0 ->RQ_0

else

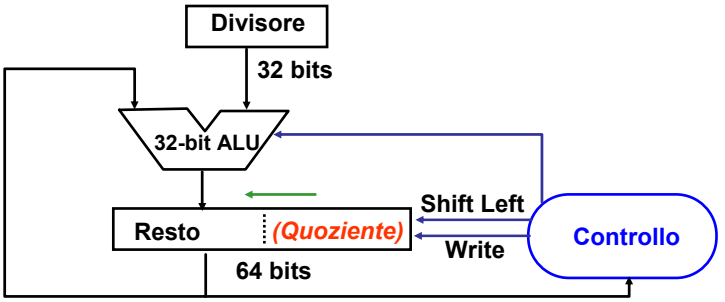
shl RQ

1 ->RQ_0

4. passi ++

5. If passi<32 goto 1.

Circuito divisore tra unsigned (3/3)



Passi dell'algoritmo

(RR: registro resto RD: registro divisore)

0. dividendo -> RR, divisore -> RD, passi=0
1. shl RR
2. $[RR_H]$-[RD] -> RR_H
3. If RR<0
   $[RR_H]$+[RD]->RR_H (ripristino)
   shl RR
   0 ->RR_0
   else
   shl RR
   1 ->RR_0
4. passi ++
5. If passi<32 goto 2.
6. shr RR_H

Esempio: 7 / 2

	Passo	Divisore	Resto
0	Inizializzazione	0010	0000 0111
	shl RR	0010	0000 1110
1	[RR]-[RD] -> RR	0010	1110 1110
	RR<0 => ripristino, shl RR, 0 ->RR0	0010	0001 1100
2	[RR]-[RD] -> RR	0010	1111 1100
	RR<0 => ripristino, shl RR, 0 ->RR0	0010	0011 1000
3	[RR]-[RD] -> RR	0010	0001 1000
	RR>=0 => shl RR, 1 ->RR0	0010	0011 0001
4	[RR]-[RD] -> RR	0010	0001 0001
	RR>=0 => shl RR, 1 ->RR0	0010	0010 0011
	shr RRH	0010	0001 0011

Divisione tra numeri interi signed

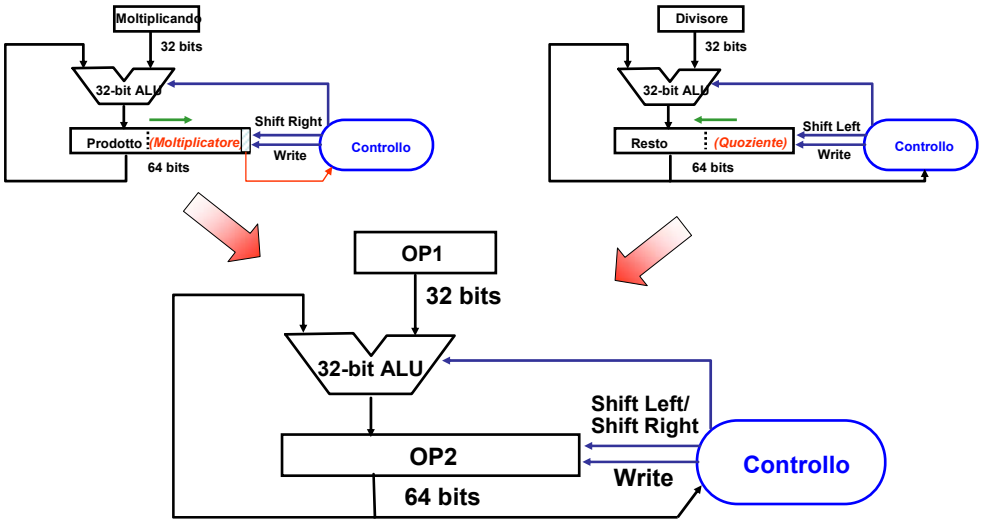
Non esistono algoritmi semplici per gestire la divisione tra interi signed come accade per la moltiplicazione.

La soluzione più semplice è di realizzare la divisione tra i valori assoluti, invertendo il segno del quoziente se i segni di dividendo e divisore sono discordi. Il segno del resto deve uguagliare quello del dividendo.

Esempio:

Dividendo	Divisore	Quoziente	Resto
+7	+2	+3	+1
+7	-2	-3	+1
-7	+2	-3	-1
-7	-2	+3	-1

Quanti circuiti occorrono ?



Operazioni aritmetiche con numeri reali

Due rappresentazioni possibili per i numeri reali:

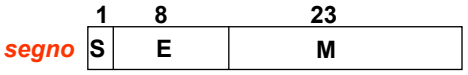
- **fixed point**

  si possono utilizzare i circuiti aritmetici per i numeri interi visti finora perché la rappresentazione interna è analoga a quella dei numeri interi

- **floating point**

  qualunque operazione richiede che siano considerati sia le mantisse che gli esponenti degli operandi. Di solito sono necessarie delle preelaborazioni (allineamento degli esponenti, denormalizzazione) e/o delle postelaborazioni (rinormalizzazione) per ottenere la rappresentazione corretta del risultato

Richiami sulla rappresentazione floating point (IEEE 754)



esponente:
intero con segno rappresentato in eccesso 127

mantissa:
reale in valore assoluto, rappresentato in fixed point. Normalizzata con bit nascosto per la parte intera: 1.M

$$N = (-1)^S \, 2^{E-127} \, (1.M) \qquad 0 < E < 255$$

L'esponente effettivo è:
e = E - 127

0 = 0 00000000 0 . . . 0 **-1.5 = 1 01111111 10 . . . 0**

Intervallo di rappresentazione in valore assoluto:

$$1.0 \times 2^{-126} \ldots (2-2^{-23}) \times 2^{+127}$$

all'incirca uguale a:

$$1.8 \times 10^{-38} \ldots 3.4 \times 10^{+38}$$

Addizione in floating point

Si debba realizzare l'operazione $9.999*10^{1}+1.610*10^{-1}$

Algoritmo:

1. allineamento del punto frazionario del numero a minore esponente al punto frazionario del numero ad esponente maggiore

   $1.610*10^{-1}$ -> $0.016*10^{1}$

2. somma tra le mantisse

   $9.999*10^{1}+0.016*10^{1} = 10.015*10^{1}$

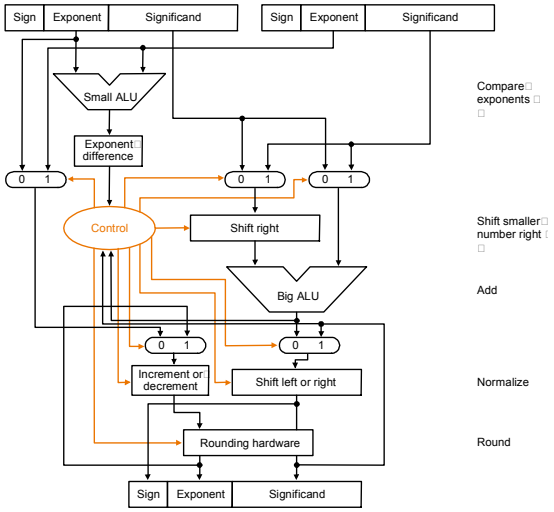
3. normalizzazione del risultato

   $10.015*10^{1}$ -> $1.0015*10^{2}$

4. arrotondamento del risultato

   $1.0015*10^{2}$ -> $1.002*10^{2}$

Schema di addizionatore floating point



Moltiplicazione in floating point

Si debba realizzare l'operazione $1.110*10^{10} * 9.200*10^{-5}$

Algoritmo:

1. calcolo dell'esponente risultante

   10-5 = 5

   Attenzione alla rappresentazione per eccessi

2. prodotto tra le mantisse

   1.110 * 9.200 = 10.212

3. normalizzazione del risultato

   $10.212 *10^{5}$ -> $1.0212 *10^{6}$

4. arrotondamento del risultato

   $1.0212 *10^{6}$ -> $1.021 *10^{6}$

Accuratezza del risultato

Nelle operazioni in floating point, l'accuratezza del risultato è limitata dal fatto che la rappresentazione è limitata: il risultato finale è affetto da un errore dovuto agli arrotondamenti effettuati sui risultati intermedi.

Bit di guardia
Bit aggiuntivi che permettono di valutare con maggiore precisione i risultati intermedi e giungere, dopo l'arrotondamento, ad un risultato finale più accurato. Lo standard IEEE 754 utilizza due bit di guardia (*guard* e *round*) per i risultati intermedi.

Esempio:

$2.56 \times 10^0 + 2.34 \times 10^2$

senza bit di guardia:

$2.56 \times 10^0 + 2.34 \times 10^2 \rightarrow 0.02 \times 10^2 + 2.34 \times 10^2 = 2.36 \times 10^2$

rounding

con i bit di guardia:

$2.56 \times 10^0 + 2.34 \times 10^2 \rightarrow 0.0256 \times 10^2 + 2.34 \times 10^2 = 2.3656 \times 10^2 = 2.37 \times 10^2$

guard round rounding