

Progetto di una ALU

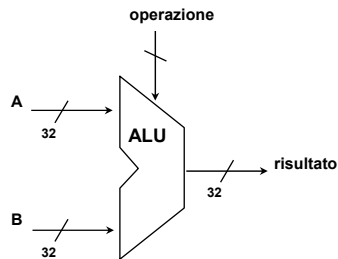
ALU n. [*Arthritic Logic Unit or (rare) Arithmetic Logic Unit*]

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



F. Tortorella

Corso di Calcolatori Elettronici

Università degli Studi
di Cassino

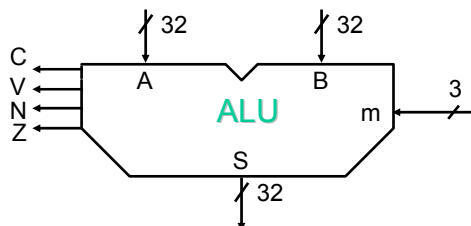
Specifiche di progetto

- 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.



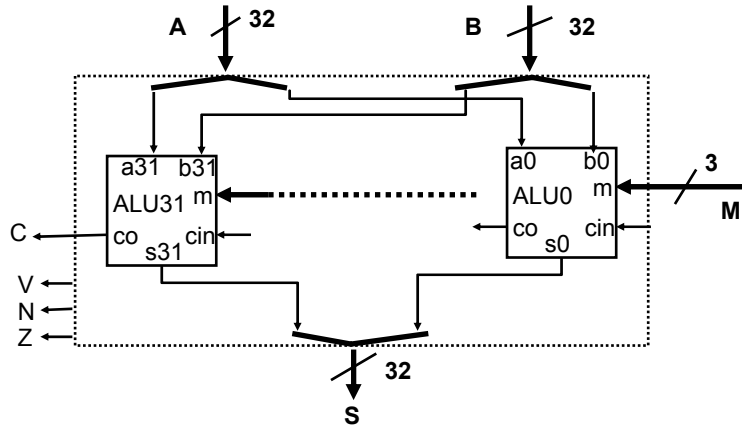
F. Tortorella

Corso di Calcolatori Elettronici

Università degli Studi
di Cassino



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



F. Tortorella

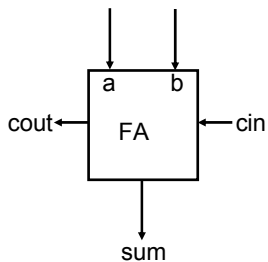
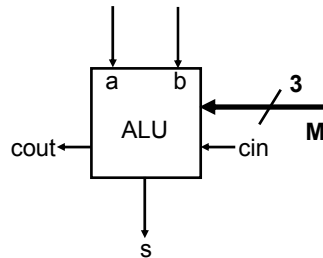
Corso di Calcolatori Elettronici

Università degli Studi
di Cassino

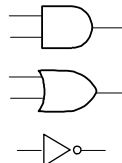
ALU da 1 bit

Come realizzarla ?

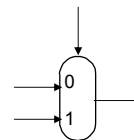
Quali tipi di componenti noti ci possono
essere utili ?



Full Adder



Porte logiche

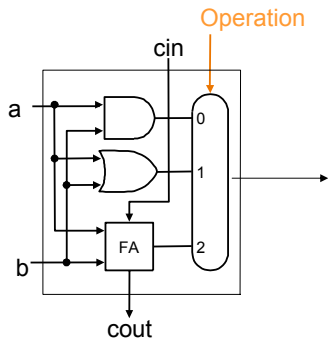


Multiplexer

F. Tortorella

Corso di Calcolatori Elettronici

Università degli Studi
di Cassino

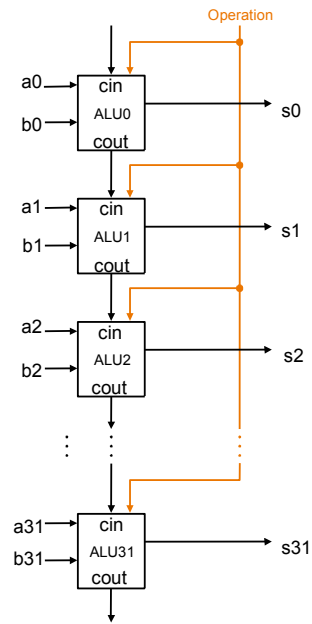


Operazioni possibili: somma, AND, OR.

F. Tortorella

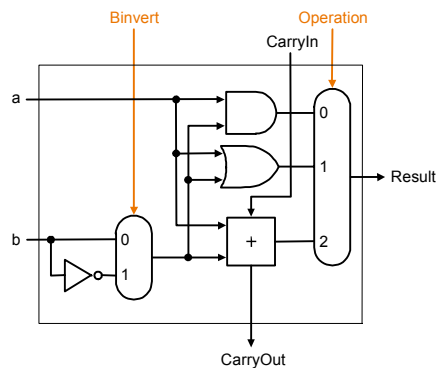
Corso di Calcolatori Elettronici

Università degli Studi di Cassino



Realizzazione della sottrazione

- uso della rappresentazione per complementi
- $A - B = A + \overline{B}$
- $\overline{\overline{B}} = B$



F. Tortorella

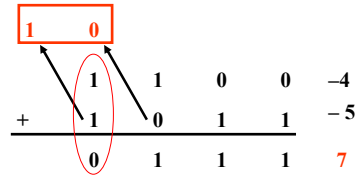
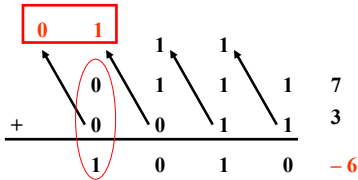
Corso di Calcolatori Elettronici

Università degli Studi di Cassino

Gestione dell'overflow

- OVERFLOW: il risultato dell'operazione non è rappresentabile ($> \text{MAX}$ o $< \text{MIN}$)
- 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):



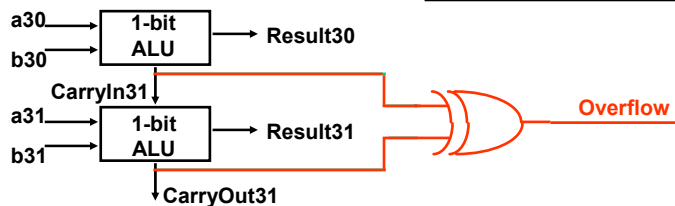
F. Tortorella

Corso di Calcolatori Elettronici

Università degli Studi
di Cassino

$$\text{Overflow} = \text{CarryIn}[31] \text{ XOR } \text{CarryOut}[31]$$

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

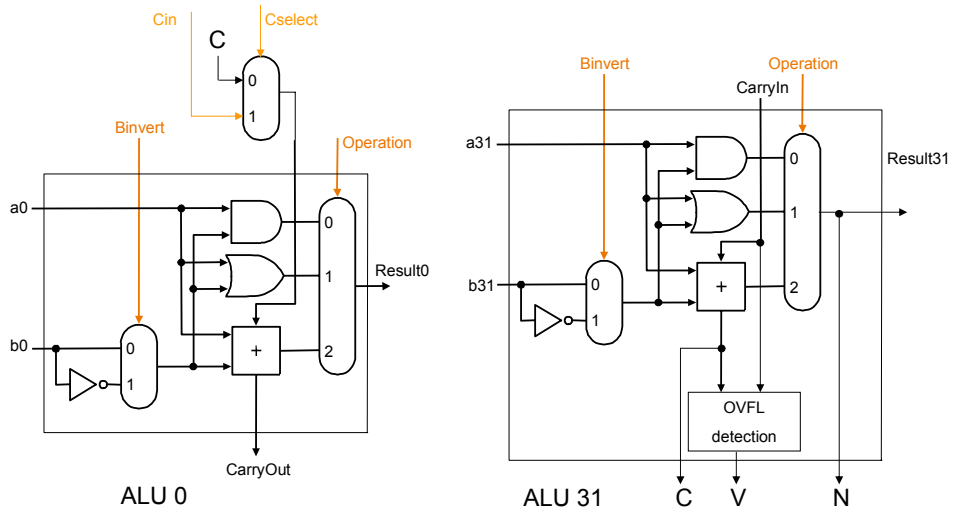


F. Tortorella

Corso di Calcolatori Elettronici

Università degli Studi
di Cassino

Modifiche da apportare alle ALU 0 e 31



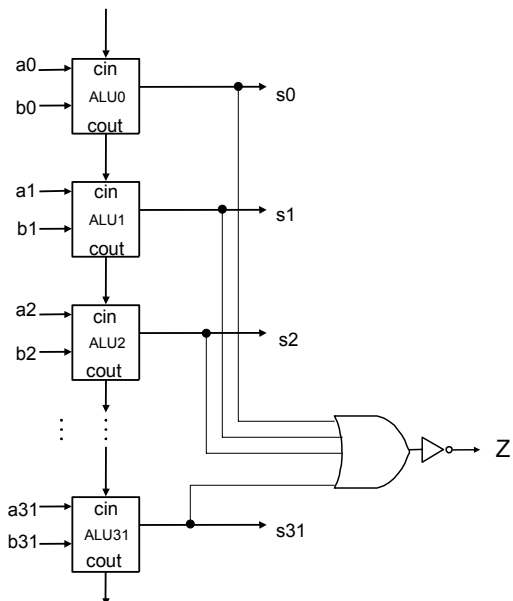
F. Tortorella

Corso di Calcolatori Elettronici

Università degli Studi di Cassino

Gestione del flag Z

Il flag Z vale 1 quando il risultato è nullo!



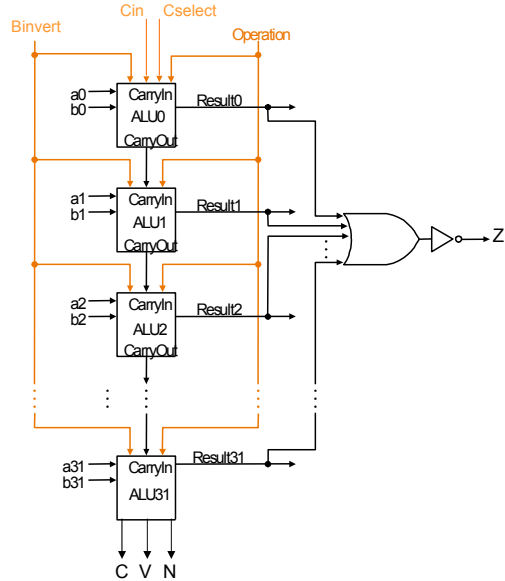
F. Tortorella

Corso di Calcolatori Elettronici

Università degli Studi di Cassino

Progetto complessivo

	Op	Binvert	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



F. Tortorella

Corso di Calcolatori Elettronici

Università degli Studi
di Cassino

Valutazione del progetto

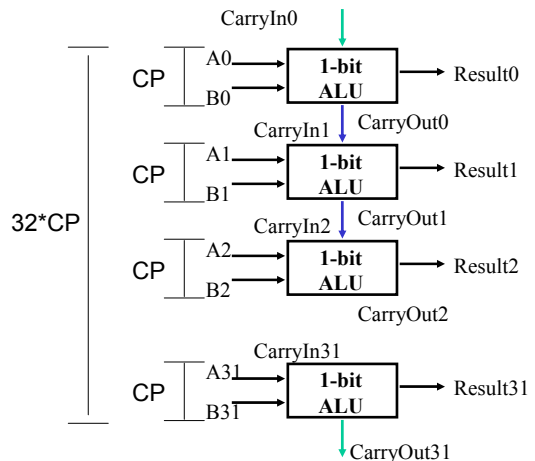
Quanto vale il percorso critico? $n \cdot CP$

Problema:

il ripple carry adder è lento.

Esiste un'alternativa per la
realizzazione dell'adder ?

Una Rete Combinatoria !



F. Tortorella

Corso di Calcolatori Elettronici

Università degli Studi
di Cassino

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 ?

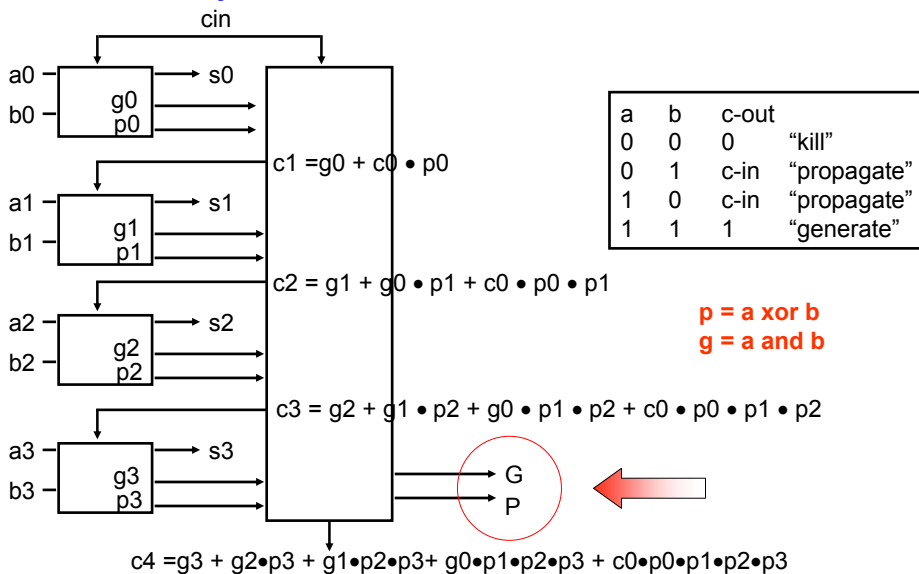
$$p_i = a_i + b_i$$

F. Tortorella

Corso di Calcolatori Elettronici

Università degli Studi
di Cassino

Adder con carry look-ahead a 4 bit



F. Tortorella

Corso di Calcolatori Elettronici

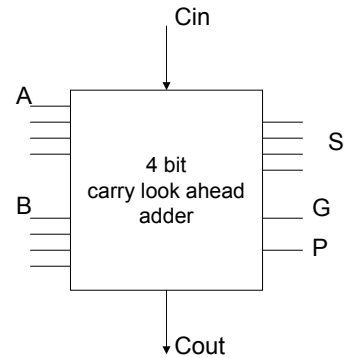
Università degli Studi
di Cassino

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

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

$$C_{out} = \underbrace{g_3 + g_2 \cdot p_3 + g_1 \cdot p_2 \cdot p_3 + g_0 \cdot p_1 \cdot p_2 \cdot p_3}_{G} + \underbrace{C_{in} \cdot p_0 \cdot p_1 \cdot p_2 \cdot p_3}_{P}$$

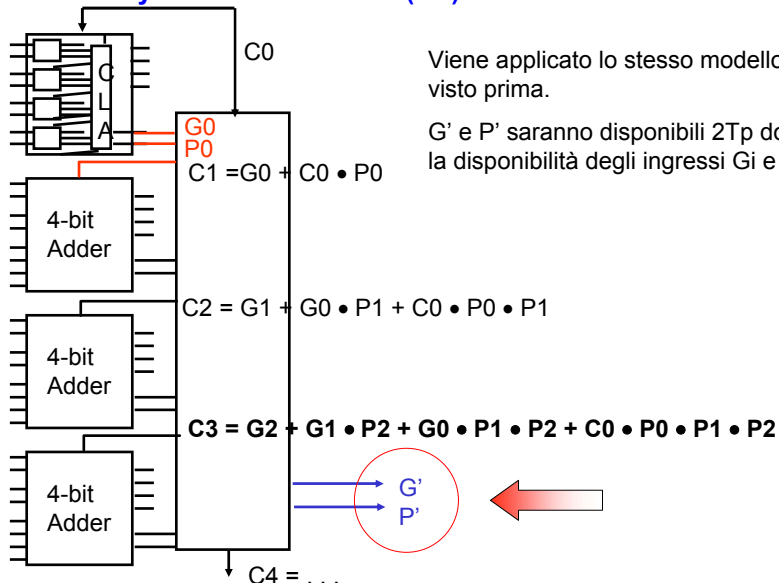
$$C_{out} = G + C_{in} \cdot P$$



G e P sono disponibili dopo un intervallo pari a $2T_p$ (ritardo di porta) dall'istante in cui sono disponibili gli ingressi g_i e p_i .

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

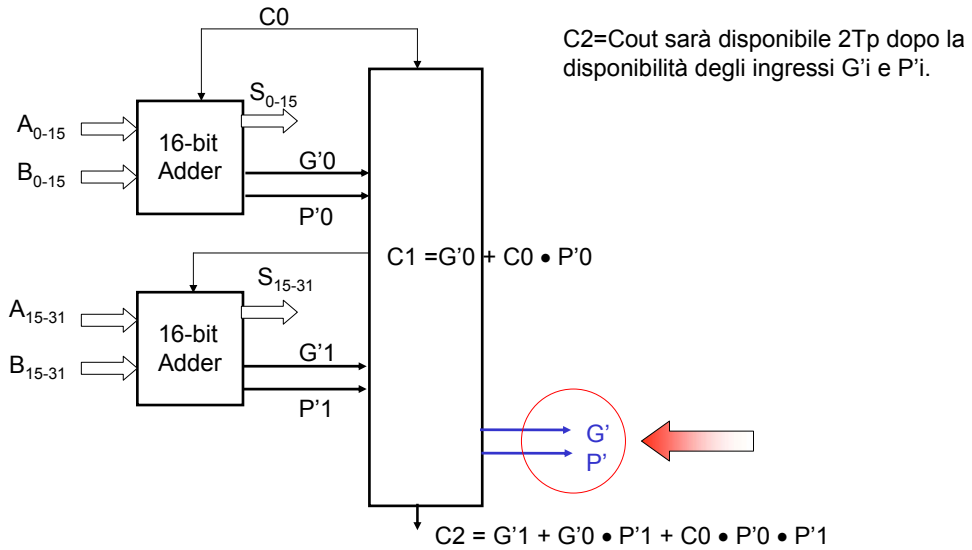
Adder con carry look ahead a 32 bit (2/3)



Viene applicato lo stesso modello visto prima.

G' e P' saranno disponibili $2T_p$ dopo la disponibilità degli ingressi G_i e P_i .

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



F. Tortorella

Corso di Calcolatori Elettronici

Università degli Studi di Cassino

La moltiplicazione tra unsigned

- Operazione svolta a mano:

Moltiplicando	1000	
Moltiplicatore	1001	
	1000	
	0000	
	0000	
	1000	
Prodotto	01001000	

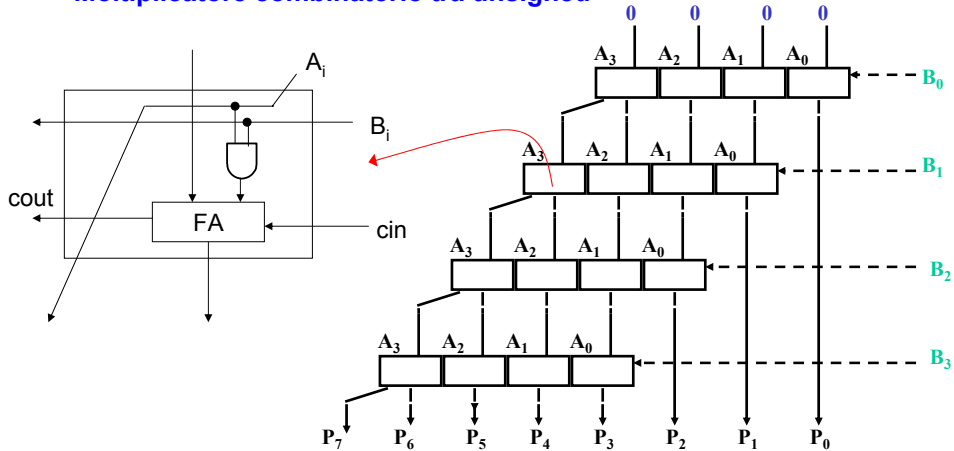
- m bits \times n bits = prodotto a $m+n$ bit
- In base 2 l'operazione è semplice:
 - $0 \Rightarrow$ aggiungi 0 (0 x moltiplicando)
 - $1 \Rightarrow$ aggiungi una copia (1 x moltiplicando)

F. Tortorella

Corso di Calcolatori Elettronici

Università degli Studi di Cassino

Moltiplicatore combinatorio tra unsigned



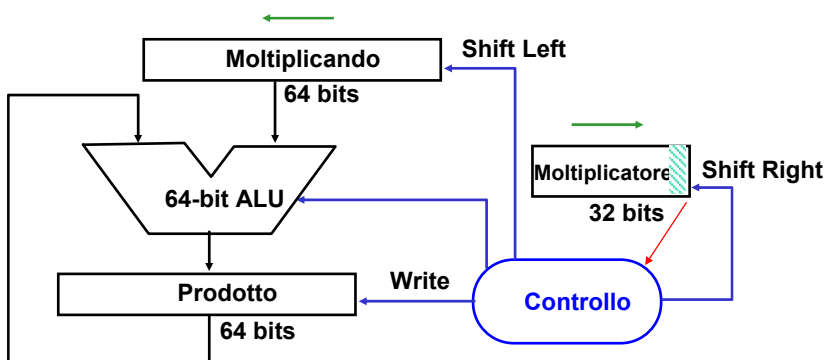
Il livello i-mo somma $A \cdot 2^i$ se $B_i=1$

F. Tortorella

Corso di Calcolatori Elettronici

Università degli Studi
di Cassino

Moltiplicatore sequenziale tra unsigned (1/3)



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

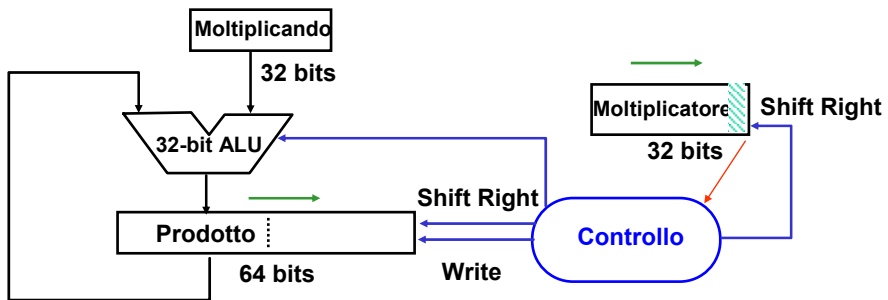
Shift right del prodotto, invece di shift left del moltiplicando?

F. Tortorella

Corso di Calcolatori Elettronici

Università degli Studi
di Cassino

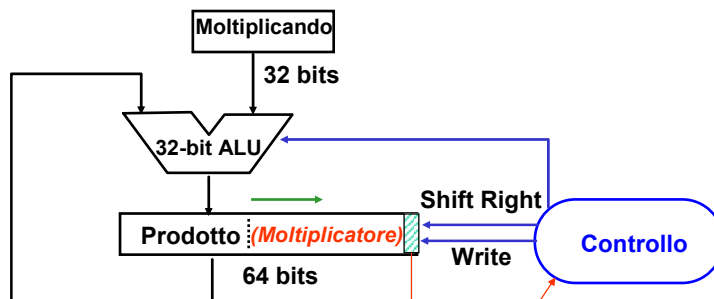
Moltiplicatore sequenziale tra unsigned (2/3)



- 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

$$9 \times 3 = 27$$

					1	0	0	1			
				x	0	0	1	1			
				0	0	0	0	1	0		
				0	0	0	1	0	0		
				0	0	0	0	0	0		
				0	0	0	0	0	0		
				0	0	0	1	1	0	1	

F. Tortorella

Corso di Calcolatori Elettronici

Università degli Studi
di Cassino

E i numeri signed ?

Intero con segno

$$-7 \times 3 = -21$$

					1	0	0	1				
				x	0	0	1	1				
				1	1	1	1	1	0	0	1	
				1	1	1	1	0	0	1	0	
				0	0	0	0	0	0	0	0	
				0	0	0	0	0	0	0	0	
				1	1	1	0	1	0	1	1	

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

F. Tortorella

Corso di Calcolatori Elettronici

Università degli Studi
di Cassino

L'algoritmo di Booth (1/2)

Esempio

$2 \times 6 = 0010 \times 0110$:

```

      0010
      0110
    +-----
+   0000    shift
+   0010    add
+   0010    add
+   0000    shift
    -----
   00001100
  
```

Possiamo considerare che $6 = -2 + 8$ $0110 = -00010 + 01000 = 11110 + 01000$

```

      0010
      0110
    +-----
+   0000    shift
-   0010    sub
+   0000    shift
+   0010    add
    -----
   00001100
  
```

```

      0010
      0110
    +-----
00000000
11111100
00000000
00010000
    -----
   00001100
  
```

F. Tortorella

Corso di Calcolatori Elettronici

Università degli Studi
di Cassino

L'algoritmo di Booth (2/2)

Bit corrente	Bit precedente	Situazione	Esempio	Op
1	0	inizio stringa di 1	000111 <u>1</u> 000	sub
1	1	dentro stringa di 1	00011 <u>1</u> 1000	nop
0	1	fine stringa di 1	00 <u>0</u> 1111000	add
0	0	dentro stringa di 0	<u>0</u> 01111000	nop

01101
x 11010



```

      0 1 1 0 1 |
      x 0 -1 +1 -1 0
    -----
0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 0 0 1 1
0 0 0 0 1 1 0 1
1 1 1 0 0 1 1
0 0 0 0 0 0
    -----
1 1 1 0 1 1 0 0 1 0
  
```

F. Tortorella

Corso di Calcolatori Elettronici

Università degli Studi
di Cassino

La divisione tra interi

Operazione svolta a mano:

$$\begin{array}{r}
 \text{dividendo} \quad 1001010 \quad | \quad 1000 \quad \text{divisore} \\
 \underline{1000} \quad \quad \quad | \quad 1001 \quad \text{quoziente} \\
 1010 \\
 \underline{1000} \\
 10 \quad \text{resto}
 \end{array}$$

$$DD = DV * Q + R$$

$$Q = q_{n-1}b^{n-1} + q_{n-2}b^{n-2} + \dots + q_1b^1 + q_0$$

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

$$DD_i - DV * b^{n-i} \begin{cases} >= 0 \Rightarrow q_{n-i} = 1 \quad DD_{i+1} = DD_i - DV * b^{n-i} \\ < 0 \Rightarrow q_{n-i} = 0 \quad DD_{i+1} = DD_i \end{cases}$$

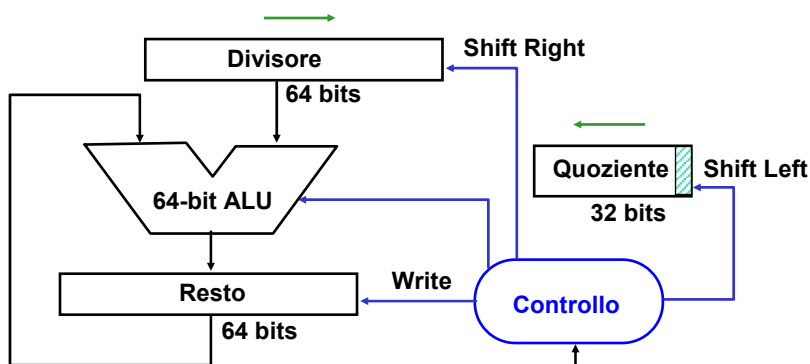
$$DD_1 = DD \quad R = DD_{n+1}$$

F. Tortorella

Corso di Calcolatori Elettronici

Università degli Studi
di Cassino

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.

F. Tortorella

Corso di Calcolatori Elettronici

Università degli Studi
di Cassino

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 ->RQ₀

else

shl RQ

1 ->RQ₀

3. shr RD

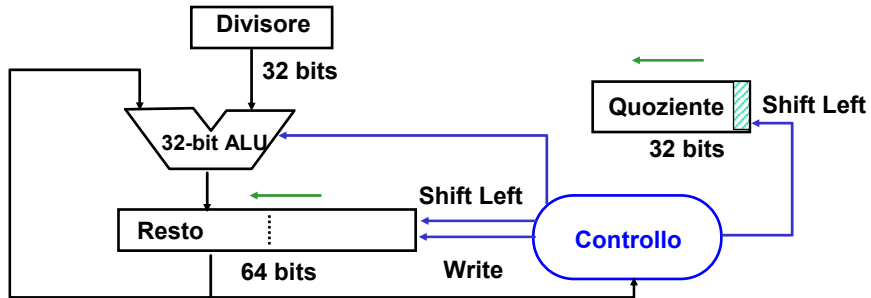
4. passi ++

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 ($[RR_H]-[RD] \rightarrow RR_H$) 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 \rightarrow RR, divisore \rightarrow RD, passi=0

1. shl RR

2. $[RR_H]-[RD] \rightarrow RR_H$

3. If $RR < 0$

$[RR_H]+[RD] \rightarrow RR_H$ (ripristino)

shl RQ

0 \rightarrow RQ₀

else

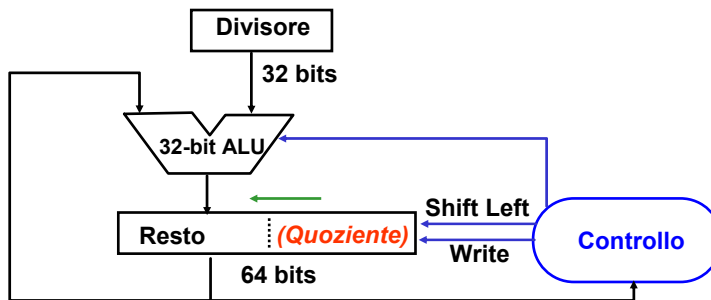
shl RQ

1 \rightarrow RQ₀

4. passi ++

5. If passi < 32 goto 1.

Circuito divisore tra unsigned (3/3)



F. Tortorella

Corso di Calcolatori Elettronici

Università degli Studi
di Cassino

Passi dell'algoritmo

(RR: registro resto RD: registro divisore)

0. dividendo \rightarrow RR, divisore \rightarrow RD, passi=0

1. shl RR

2. $[RR_H] - [RD] \rightarrow RR_H$

3. If $RR < 0$

$[RR_H] + [RD] \rightarrow RR_H$ (ripristino)

shl RR

0 \rightarrow RR_0

else

shl RR

1 \rightarrow RR_0

4. passi ++

5. If passi < 32 goto 2.

6. shr RR_H

F. Tortorella

Corso di Calcolatori Elettronici

Università degli Studi
di Cassino

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

F. Tortorella

Corso di Calcolatori Elettronici

Università degli Studi
di Cassino

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:

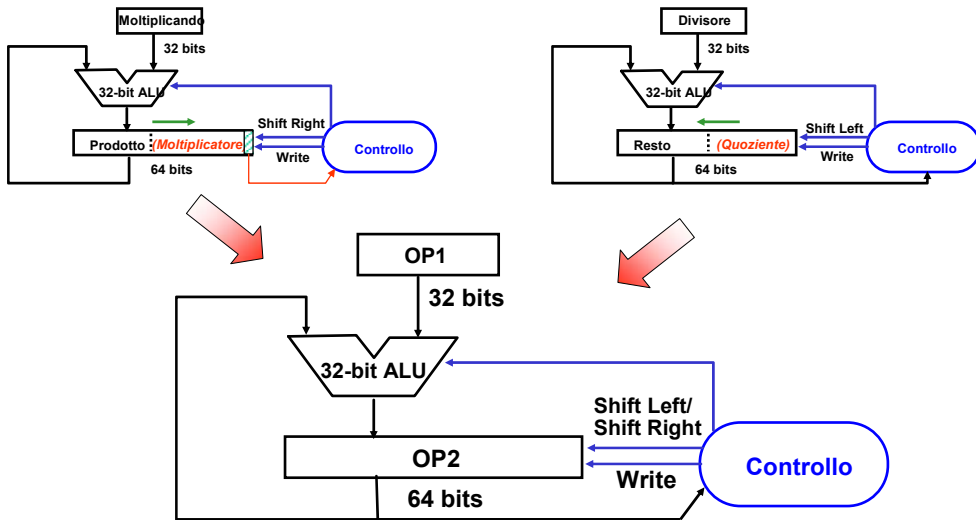
Dividendo	Divisore	Quoziente	Resto
+7	+2	+3	+1
+7	-2	-3	+1
-7	+2	-3	-1
-7	-2	+3	-1

F. Tortorella

Corso di Calcolatori Elettronici

Università degli Studi
di Cassino

Quanti circuiti occorrono ?



F. Tortorella

Corso di Calcolatori Elettronici

Università degli Studi
di Cassino

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**

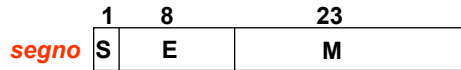
qualunque operazione richiede che siano considerati sia le mantisse che gli esponenti degli operandi. Di solito sono necessarie delle preelaborazioni (allineamento degli esponenti, denormalizzazione) e/o delle postelaborazioni (rinormalizzazione) per ottenere la rappresentazione corretta del risultato

F. Tortorella

Corso di Calcolatori Elettronici

Università degli Studi
di Cassino

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

$$N = (-1)^S 2^{E-127} (1.M)$$

$0 < E < 255$

L'esponente effettivo è:
 $e = E - 127$

$$0 = 0 \text{ 00000000 } 0 \dots 0 \quad -1.5 = 1 \text{ 01111111 } 10 \dots 0$$

Intervallo di rappresentazione in valore assoluto:

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

all'incirca uguale a:

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

Addizione in floating point

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

Algoritmo:

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

$$1.610 \times 10^{-1} \rightarrow 0.016 \times 10^1$$

2. somma tra le mantisse

$$9.999 \times 10^1 + 0.016 \times 10^1 = 10.015 \times 10^1$$

3. normalizzazione del risultato

$$10.015 \times 10^1 \rightarrow 1.0015 \times 10^2$$

4. arrotondamento del risultato

$$1.0015 \times 10^2 \rightarrow 1.002 \times 10^2$$

Moltiplicazione in floating point

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

Algoritmo:

1. calcolo dell'esponente risultante

$$10 - 5 = 5$$

Attenzione alla rappresentazione per eccessi

2. prodotto tra le mantisse

$$1.110 \cdot 9.200 = 10.212$$

3. normalizzazione del risultato

$$10.212 \cdot 10^5 \rightarrow 1.0212 \cdot 10^6$$

4. arrotondamento del risultato

$$1.0212 \cdot 10^6 \rightarrow 1.021 \cdot 10^6$$

F. Tortorella

Corso di Calcolatori Elettronici

Università degli Studi
di Cassino

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:

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

senza bit di guardia:

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

con i bit di guardia:

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

guard

round

rounding

rounding

F. Tortorella

Corso di Calcolatori Elettronici

Università degli Studi
di Cassino