Circuiti combinatori
ALU

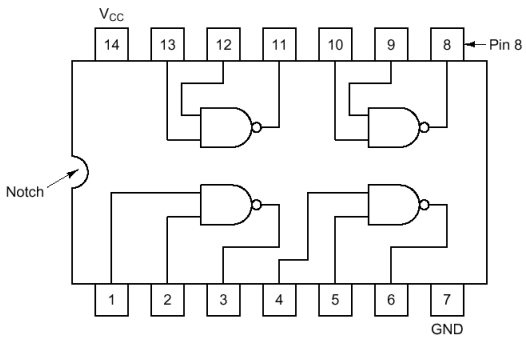
Salvatore Orlando

Circuiti integrati

- I circuiti logici sono realizzatati come IC (circuiti integrati)
  - realizzati su chip di silicio (piastrina)
  - gates e fili depositati su chip di silicio, inseriti in un package e collegati all'esterno con un certo insieme di pin (piedini)
  - gli IC si distinguono per grado di integrazione
    - da singole porte indipendenti, a circuiti più complessi
- Integrazione
  - SSI (Small Scale Integrated): 1-10 porte
  - MSI (Medium Scale Integrated): 10-100 porte
  - LSI (Large Scale Integrated): 100-100.000 porte
  - VLSI (Very Large Scale Integrated): > 100.000 porte
- Con tecnologia SSI, gli IC contenevano poche porte, direttamente collegati ai pin esterni
- Con tecnologia MSI, gli IC contenevano alcuni componenti base
  - circuiti comunemente incontrati nel progetto di un computer
- Con tecnologia VLSI, un IC può oggi contenere una CPU completa (o più)
  - microprocessore

Esempio di chip SSI

- *Texas Instruments 7400*
  - DIP (Dual Inline Package)
  - Tensione e terra condivisi da tutte le porte
  - Tacca per individuare l'orientamento del chip
- SSI
  - Rapporto pin/gate (piedini/porte) grande
  - Con l'aumento del del grado di integrazione
    - rapporto pin/gate diminuisce (molti gate rispetto ai pin)
    - chip più specializzati
      - es. chip che implementano particolari circuiti combinatori

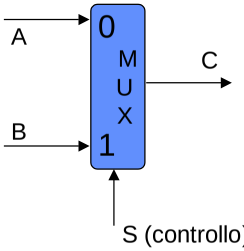


Circuiti combinatori

- Nella costruzione di un processore, o di un componente elettronico specializzato
  - necessario usare *componenti* realizzati con circuiti combinatori
    - l'output ottenuto solo combinando i valori in input con poter logiche
    - il circuito non ha memoria
- Circuiti combinatori usati quindi come *blocchi base* per costruire circuiti più complessi
  - spesso realizzati direttamente come componenti MSI
- Tratteremo:
  - Multiplexer e Demultiplexer
  - Decoder
  - ALU
- Per implementare circuiti combinatori useremo
  - PLA
  - ROM

Multiplexer (1)

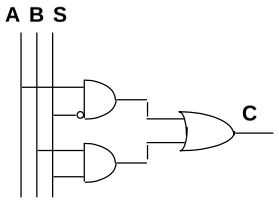
- *n input* ed *1 output*
- *$log_2 n$* segnali di *controllo* (da considerare ulteriori *input* del circuito)
- Il multiplexor, sulla base dei *segnali di controllo*, seleziona quale tra gli *n* input verrà presentato come output del circuito
- Caso semplice: Multiplexer 2:1, con un solo bit di controllo



se S=0: passa A
se S=1: passa B

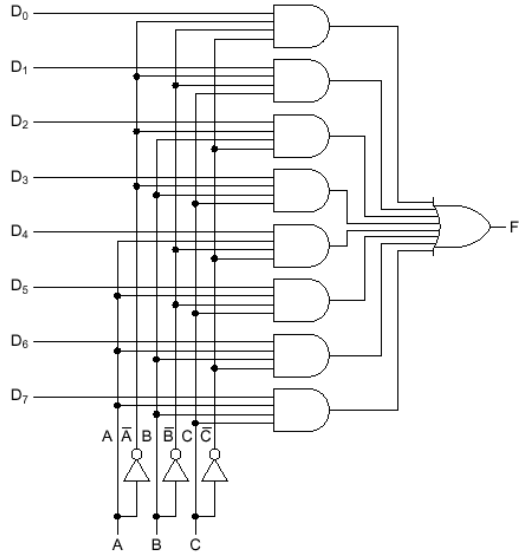
S \ AB	00	01	11	10
0			1	1
1		1	1	

C = A~S + BS

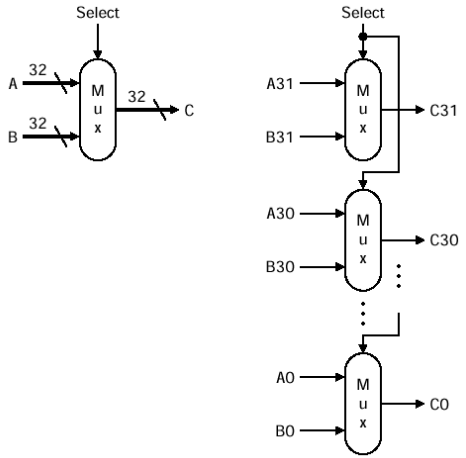


Multiplexer (2)

- Multiplexer 8:1
- $8=2^3$ input e 3 segnali di controllo
- $8=2^3$ porte AND e 1 porta OR
- In ogni porta AND entra una combinazione diversa dei segnali di controllo A,B,C
  - $8=2^3$ combinazioni possibili
- in dipendenza dei valori assunti da A, B e C, tutte le porte AND (eccetto una) avranno sicuramente output uguale a 0
- solo una delle porte produrrà *eventualmente* un valore 1
  - es. A~BC=1 => passa D_5



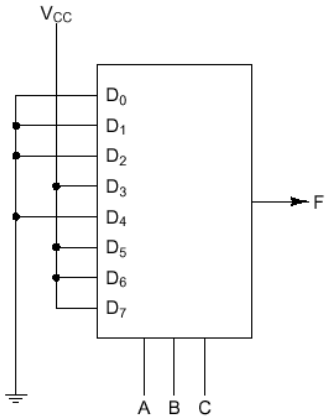
Multiplexer



- Multiplexer 2:1 a 32-bit (con *fili larghi* di 32 bit)
  - costruito usando 32 1-bit Multiplexer 2:1 con un segnale di controllo distribuito ai vari Multiplexer

Multiplexer e funzioni logiche arbitrarie

- Multiplexer n:1
  - possono essere usati per definire una qualsiasi funzione logica in $log_2 n$ variabili
    - funzione definita da una tabella di verità con n righe
    - le $log_2 n$ variabili in input della funzione logica *diventano* i segnali di controllo del multiplexer
  - ogni riga della tabella di verità
    - corrisponde ad uno degli n input del multiplexer, collegati ad un generatore di tensione (o alla terra)
      - se l'output, associato alla riga della tabella di verità, è 1 (o 0)
  - grande spreco di porte
    - circuito *fully encoded*
    - porte AND con arietà maggiore del necessario (+1)



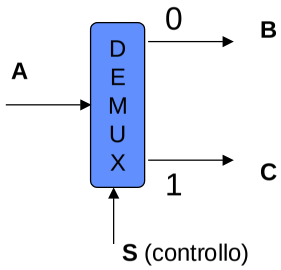
Componente MSI che realizza un multiplexer 8:1

Collegamento per ottenere la funzione:

F=ABC+ABC+ABC+ABC

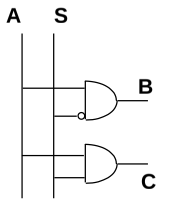
Demultiplexer

- Da 1 singola linea in input, a n linee in output
  - $\log_2 n$ segnali di controllo (S)
  - se la linea in input è uguale a 0
    - tutti gli output dovranno essere uguali a 0, indipendentemente da S
  - se la linea in input è uguale a 1
    - un solo output dovrà essere uguale a 1, tutti gli altri saranno 0
    - l'output da affermare dipende da S



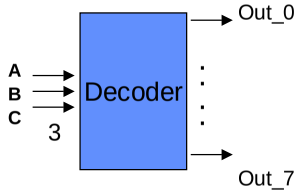
B=A~S

C=AS

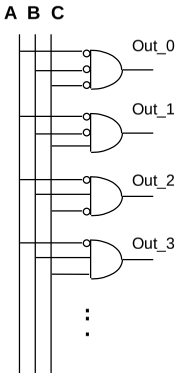


Decoder

- Componente con *n* inputs e 2^n output
  - gli n input sono interpretati come un numero unsigned
  - se questo numero rappresenta il numero i, allora
    - solo il bit in output di indice i (i=0,1,...,2^n-1) verrà posto ad 1
    - tutti gli altri verranno posti a 0

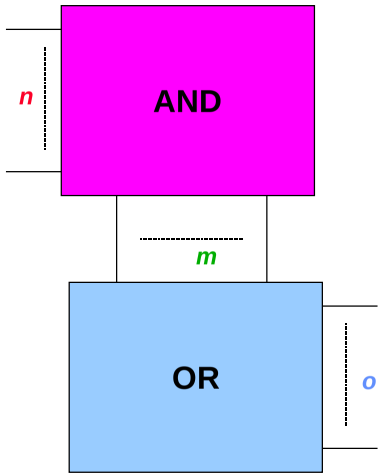


A	B	C	0	1	2	3	4	5	6	7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1



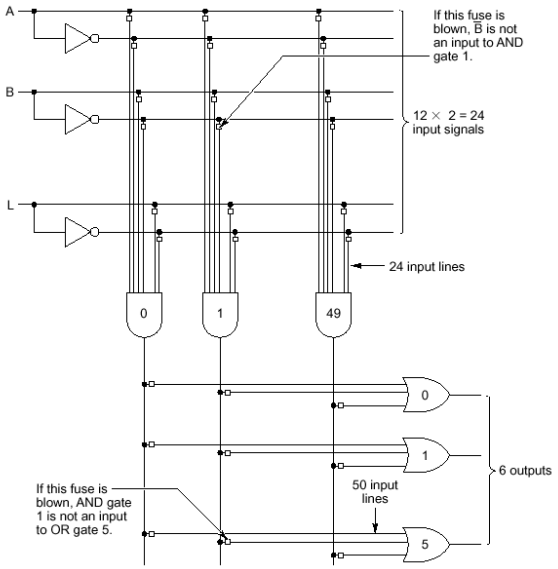
PLA

- Programming Logic Array (PLA)
- Componente per costruire funzioni logiche arbitrarie
  - permette di costruire funzioni in forma SP
    - porte AND al primo livello, e porte OR al secondo livello
  - *n* input e *o* output
    - *m* porte AND
    - o porte OR
    - *m* fissa un limite al numero di mintermini esprimibili
    - *o* fissa un limite al numero di funzioni differenti in forma canonica SP

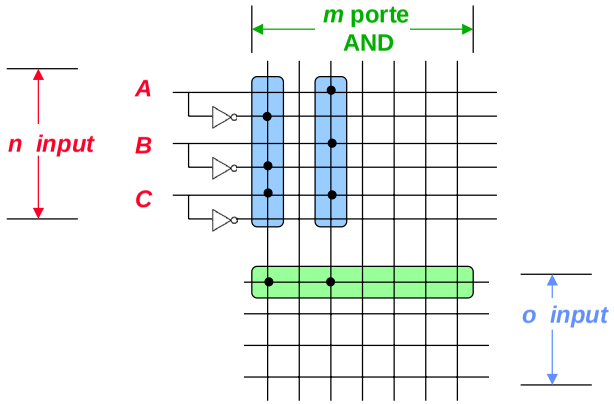


PLA

- Esempio:
  - n=12
  - o=6
  - m=50
- In ogni porta AND entrano 2n=24 input
  - normali e invertiti
- In ogni porta OR entrano m=50 input
- Fusibili da bruciare per decidere
  - quali sono gli input di ogni porta AND
  - quali sono gli input delle varie porte OR



PLA

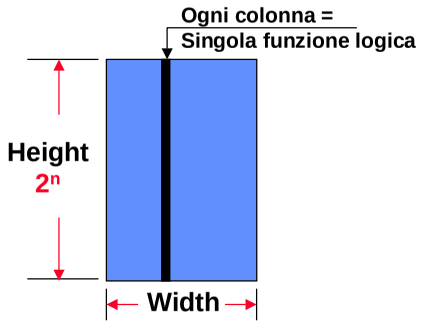


Esempio di funzione:

O = ~A ~BC + ABC

ROM

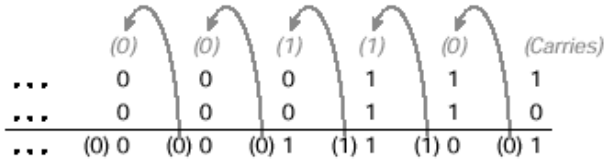
- **Memorie usabili anche per implementare, in maniera non minima, funzioni logiche arbitrarie**
  - **ROM (Read Only Memory)**
    - **PROM (Programmable ROM)**
      - scrivibili solo una volta
    - **EPROM (Erasable PROM)**
      - cancellabile con luce ultravioletta



- **In pratica**
  - **data una Tabella di Verità, le ROM sono usate per memorizzare direttamente le diverse funzioni logiche (corrispondenti a colonne distinte nella Tabella)**
  - **Indirizzo a *n* bit**
    - **individua una specifica combinazione delle *n* variabili logiche in input**
    - **individua una cella di Width bit della ROM**
- **Ogni funzione:**
  - **Singola colonna della ROM**
  - **Funzioni fully encoded**
  - **PLA più efficiente**

Addizionatori

- **Potremmo costruire un unico circuito combinatorio che implementa un addizionatore a *n* bit**
  - **dati 2*n input: $A_0 \ldots A_{n-1}$ $B_0 \ldots B_{n-1}$**
  - **n+1 diverse funzioni di output $C_0 \ldots C_{n-1}$ Rip**
  - **solo due livelli di logica, ma con porte AND e OR con *molti input***
    - ***fan-in* molto elevato (non ammissibile)**
- **Soluzione di compromesso, basata su serie di 1-bit adder collegati in sequenza**
  - **il segnale deve attraversare più livelli di logica**
  - **porte con *fan-in* limitato (ammissibile)**
  - **circuito che usa lo stesso metodo usato dall'algoritmo *carta e penna* a cui siamo abituati**



Addizionatore a singolo bit

- **La tabella di verità dell'*addizionatore a singolo bit***

A	B	Rip	Sum	Rip_out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Sum

Rip \ AB	00	01	11	10
0		1		1
1	1		1	

Rip_out

Rip \ AB	00	01	11	10
0			1	
1		1	1	1

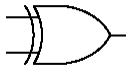
Sum = ~A ~B Rip + ~A B ~Rip + A B Rip + A ~B ~Rip

Rip_out = Rip B + A B + Rip A

Addizionatore a singolo bit

- La funzione Sum non può essere semplificata
  - ben 4 porte AND
- La costruzione di un 1-bit adder diventa più semplice impiegando porte XOR
  - funzione logica che vale 0 (F) se entrambi i bit in ingresso sono uguali, ovvero entrambi 0 (F) o entrambi 1 (T)
  - esempio di *or esclusivo* (XOR) nel linguaggio comune
    - o è bel tempo oppure prendo l'ombrello

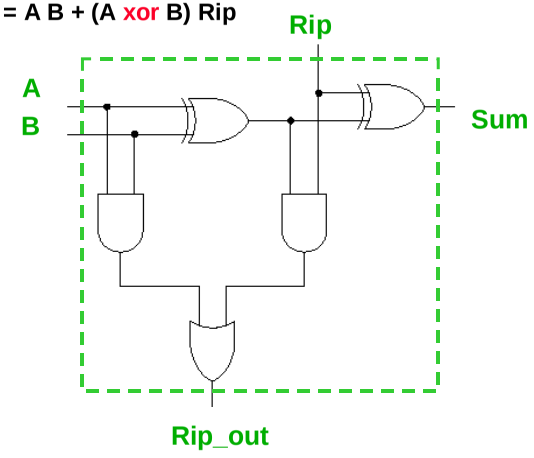
A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0



Porta XOR

1-bit adder usando porte XOR

- Funzione SUM
  - consideriamo che SUM tra una coppia di bit può essere ottenuta con
    A xor B
  - Sum = A xor B xor Rip
  - Rip_out = A B + (A xor B) Rip

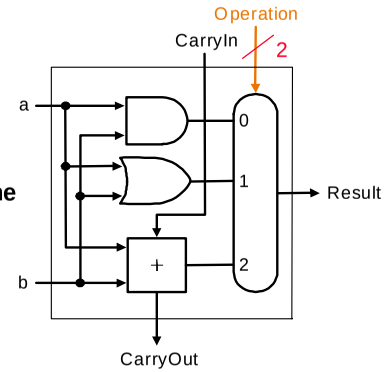


ALU & istruzioni aritmetiche/logiche

- ALU (Arithmetic Logic Unit)
  - circuito combinatorio usato per implementare le operazioni necessarie per l'esecuzione di diverse istruzioni macchina
- Studieremo l'ALU del processore MIPS
- Istruzioni a 3 operandi
  - operandi posti nei registri del processore (32 registri a 32 b = 4 B = 1 W)
- Istruzioni assembler di tipo aritmetico/logico
  cod_{op} Reg1, Reg2, Reg3
  - Semantica istruzioni
    Reg1 ← Reg2 op Reg3
- Istruzioni macchina (e relativi formati)
  istr. assembler *codificate* opportunamente su una Word di 32 b
- esempi di istruzioni aritmetico/logiche e di confronto
  - and $2, $3, $4 # $2 = $3 & $4
  - or $2, $3, $4 # $2 = $3 | $4
  - add $2, $3, $4 # $2 = $3 + $4
  - sub $2, $3, $4 # $2 = $3 - $4
  - slt $2, $3, $4 # if ($3<$4) $2=1 else $2=0

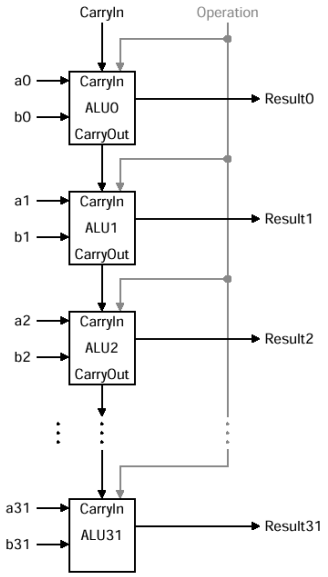
1-bit ALU

- 1-bit ALU usata per eseguire le istruzioni macchina seguenti:
  - and $2, $3, $4
  - or $2, $3, $4
  - add $2, $3, $4
- Operation è un segnale di controllo a 2 bit
  - determina il tipo di operazione che l'ALU deve eseguire
- l'ALU è la tipica componente che fa parte del *Datapath* (Parte operativa) del processore
- La Parte Controllo comanda l'esecuzione delle varie istruzioni
  - settando opportunamente i segnali di controllo dell'ALU (e delle altre componenti della Parte operativa)



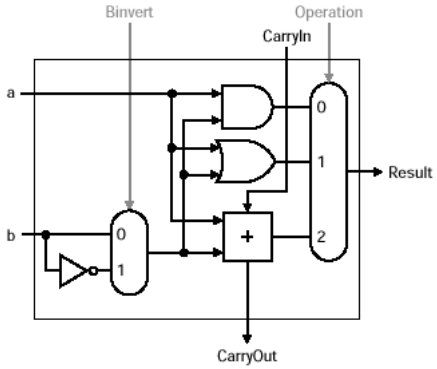
32-bit ALU

- 32-bit ALU
  - catena di 1-bit ALU con propagazione del Carry
  - segnali di controllo per determinare l'operazione che l'ALU deve eseguire
    - Operation :propagato a tutte le 1-bit ALU



Inversione e sottrazione

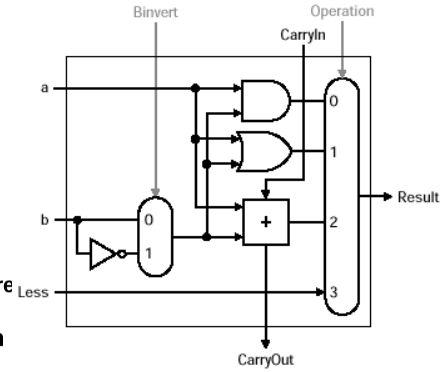
- L'1-bit ALU precedente può essere *resa più complessa* per poter eseguire:
  - sub $2, $3, $4
- Operazione di sottrazione:
  - $2 = $3 - $4 trasformata in: $2 = $3 + (-$4)
  - (-$4) significa che bisogna prima determinare il complemento a 2 del *numero signed* contenuto in $4
  - il complemento a 2 si ottiene
    - effettuando il l'inversione (complemento a 1 bit-a-bit)
    - sommando 1
  - l'ALU deve quindi possedere i circuiti predisposti per
    - invertire il secondo operando (b) e sommare 1
    - sommare 1 (che si ottiene ponendo semplicemente a 1 il *carry-in* dell'ALU)



Binvert	Operation	Istruzione
0	00	and
0	01	or
0	10	sum
1	10	sub

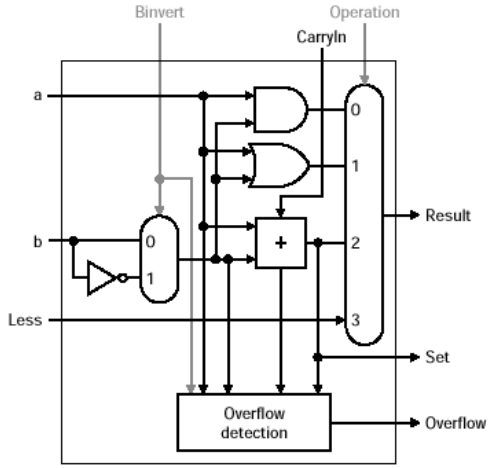
Istruzioni di confronto

- slt $2, $3, $4 (*set less than*)
  - $2=1 se è vero che $3 < $4
  - $2=0 altrimenti
- Se $3 < $4 allora $3-$4 < 0
- Quindi, per effettuare il confronto, possiamo semplicemente:
  - sottrarre e controllare il bit di segno
  - se non c'è overflow durante la sottrazione
    - il valore del bit di segno del risultato della sottrazione può essere semplicemente assegnato al bit meno significativo dei 32 bit in output
    - tutti gli altri bit in output devono essere posti a 0
- Tutte le 1-bit ALU devono quindi avere un ingresso in più
  - l'input Less, che verrà posto a 0 o a 1 sulla base del risultato dell'istruzione slt



Istruzioni di confronto

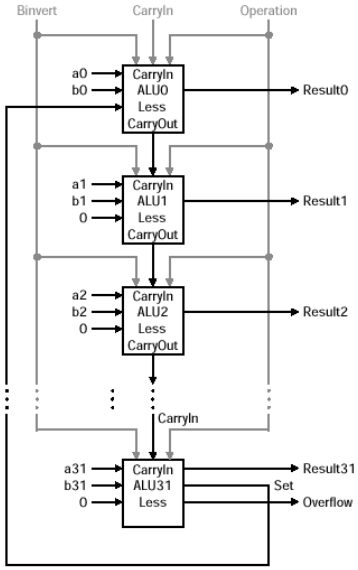
- L'ultima 1-bit ALU è più complessa poiché
  - deve controllare l'overflow
  - deve fornire, come ulteriore output, il bit di segno del risultato delle sottrazione (Set)
    - questo per permettere l'implementazione di slt
    - Set deve essere ridiretto verso la 1-bit ALU che fornirà in output il bit meno significativo del risultato
- Il blocco che controlla l'overflow lo fa sulla base
  - del tipo di operazione (sum o sub), identificata tramite Binvert
  - i segni degli operandi
  - il segno del risultato



Alu complessiva

- Output Set dell'ultima 1-bit ALU viene ridiretto sull'input Less della prima 1-bit ALU
- Tutti i bit Less delle varie 1-bit ALU (eccetto la prima) vengono posti a 0
- Segnali di controllo:
  - Binvert e Carryin vengono entrambi *asserted (affermati)* per sottrarre (sub e slt)
  - I bit di Operation sono posti a 11 per far passare in output l'ultimo bit in ingresso ai Multiplexer 4:1

Binvert	Carryin	Operation	Istruzione
0	0	00	and
0	0	01	or
0	0	10	sum
1	1	10	sub
1	1	11	slt



slt e overflow

- Il circuito proposto per implementare l'ultima 1-bit ALU della catena
  - potrebbe NON FUNZIONARE per il slt nel caso di overflow
  - non è ottimale per quanto riguarda l'overflow

- Caso di malfunzionamento relativo a slt
  - slt $2, $3, $4
  - se $3>0 e $4<0
    - potremmo concludere direttamente che è vero che $3>$4 => $2=0
    - se invece sottraiamo per implementare slt, finiamo per sommare due numeri positivi ($3 + (-$4))
      - potremmo avere overflow, ottenendo così un bit di segno (Set) non valido (uguale a 1, invece che uguale a 0)
  - un ragionamento analogo potrebbe essere fatto nel caso in cui $3<0 e $4>0

Circuito per slt

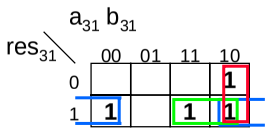
- Set deve essere determinato in modo da evitare il malfunzionamento precedente, relativo a un overflow non voluto
- Siano
  - $a = a_{31}....a_0$ e $b = b_{31}....b_0$ i due numeri da confrontare
  - $res = res_{31}....res_0$ il risultato degli 1-bit adder
  - $c = c_{31}....c_0$ il risultato della ALU, che nel caso di slt potrà solo essere
    - 0…….01 oppure 0…….00

- Se a>=0 e b<0, allora a>b, e possiamo porre direttamente Set = 0
- Se a<0 e b>=0, allora a<b, e possiamo porre direttamente Set = 1
  - nei 2 casi di sopra, anche se comunque all'ALU viene comandato di eseguire una sottrazione, l'eventuale *overflow* dovrà essere *ignorato*

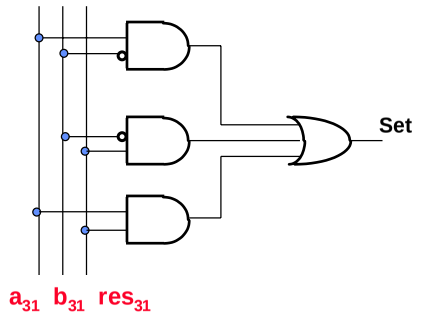
- Se a>0 e b>0, oppure se a<0 e b<0, allora
  - possiamo considerare il risultato della sottrazione, e possiamo porre Set = res_{31}
  - in questi casi non si può verificare OVERFLOW, per cui res_{31} conterrà correttamente il bit di segno corretto

Circuito corretto per slt

a_{31}	b_{31}	res_{31}	Set	
0	0	0	0	
0	0	1	1	
1	1	0	0	
1	1	1	1	
0	1	X	0	← a>=0 e b<0
1	0	X	1	← a<0 e b>=0

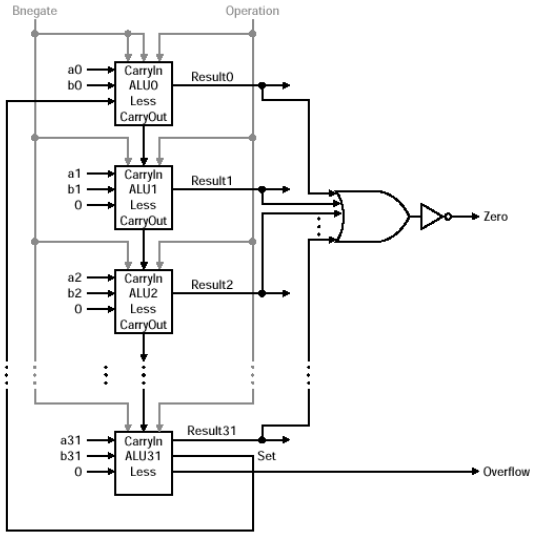


Set = a_{31} ~b_{31} + ~b_{31} res_{31} + a_{31} res_{31}

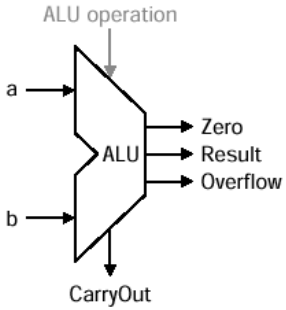


Alu finale

- Abbiamo risparmiato un bit di controllo
  - Bnegate al posto di: (Binvert, Carryin)
  - Nell'ALU precedente, infatti, (Binvert, Carryin) venivano sempre *asserted* o *deasserted* assieme
- Abbiamo ulteriormente specializzato l'ALU per l'esecuzione delle istruzioni di *branch condizionato*
  - beq e bne
  - devo controllare se
    - a==b oppure se a != b
  - posso *comandare* alla ALU di sottrarre, e controllare se
    - a-b=0 oppure se a-b != 0
  - Zero=1 ⇔ a-b==0 (a==b)



Componente combinatoria: ALU



- Simbolo usato per rappresentare la componente ALU nel progetto della CPU