



# Single Instruction Multiple Data (SIMD)

Sistemi Digitali M

A.A. 2024/2025

Stefano Mattoccia

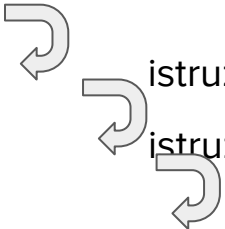
Università di Bologna

# Introduzione

## Paradigma di elaborazione di un microprocessore single core, scalare

- Prevede l'esecuzione sequenziale di una singola istruzione e l'elaborazione dei dati previsti dalla medesima istruzione

<b>LW R2,0x8000(R10)</b>	;	istruzione 1	
<b>ADD R1,R2,R3</b>	;		2
<b>XOR R5,R1,R3</b>	;		3
....			



- Possiamo definire questo paradigma Single Instruction Single Data; le istruzioni (1,2,3,etc) sono eseguite in sequenza elaborando ciascuna i dati necessari
- Esistono anche microprocessori (definiti superscalari) che leggono ed eseguono contemporaneamente più istruzioni (eg, due o più) aumentando le unità funzionali di elaborazione

# Strategie per velocizzare l'esecuzione: pipelining 1/2

## Pipelining

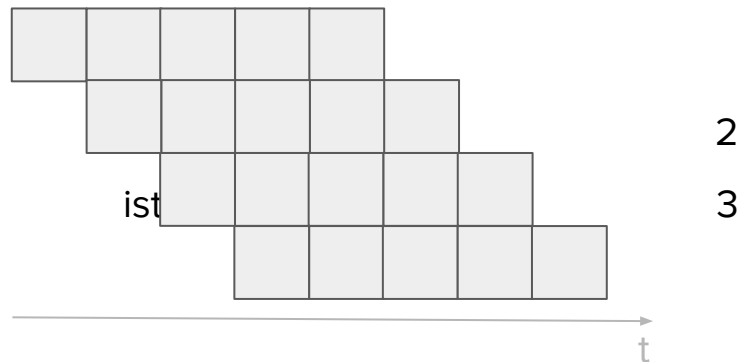
- La strategia principale per aumentare il throughput consiste nell'adottare il principio del pipelining
- Aumenta il throughput mantenendo inalterata la latenza
- Sono contemporaneamente in esecuzione più istruzioni in differenti fasi di elaborazione in base alla lunghezza della pipeline (eg, 5 stadi come nell'esempio)

**LW R2,0x8000(R10)** ; istruzione 1

**ADD R1,R2,R3** ;

**XOR R5,R1,R3** ;

....



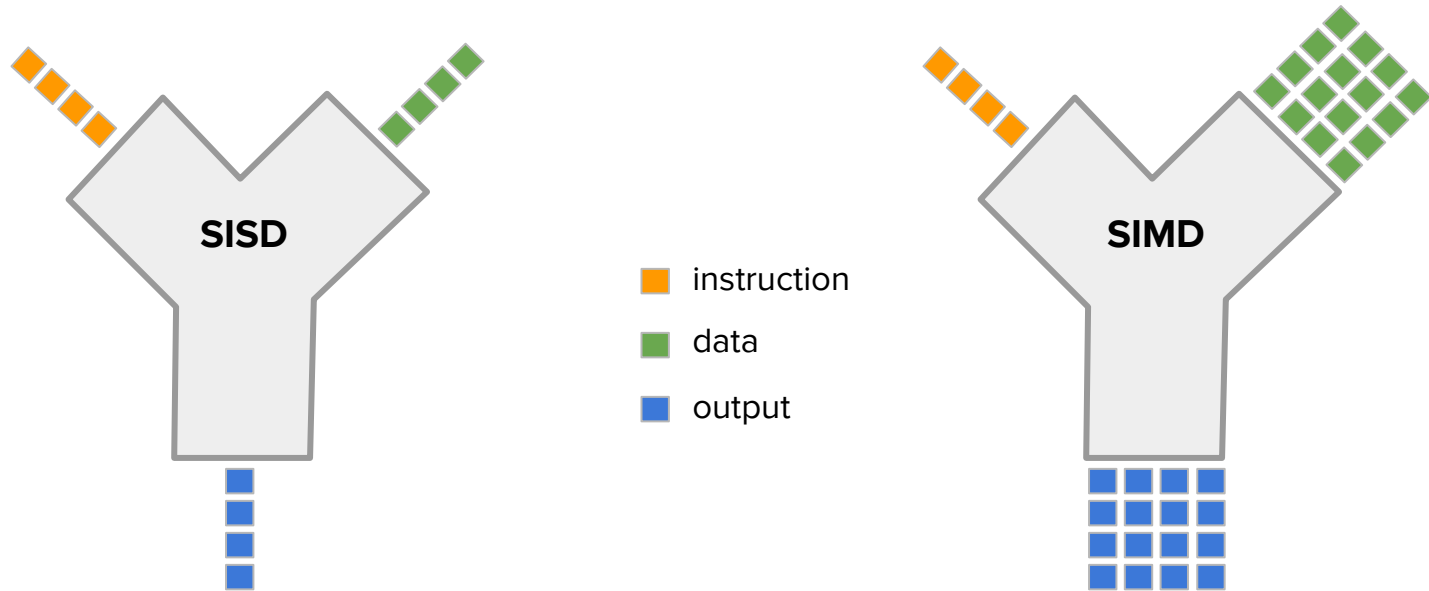
# Strategie per velocizzare l'esecuzione: pipelining 2/2

## Effetti collaterali

- L'esecuzione contemporanea di più istruzioni genera problemi che tipicamente non si riscontrano nel caso di elaborazione sequenziale
- Tuttavia, molti di questi problemi, come dipendenze tra i dati elaborati da diverse istruzioni in esecuzione, possono essere gestiti (eg, applicando il *forwarding*)
- Ovviamente, a supporto del pipelining è necessario accedere ai dati rapidamente, mediante cache, per non mandare in stallo la pipeline
- Sebbene il pipelining sia essenziale in tutti i microprocessori moderni, esistono anche altre strategie aggiuntive per aumentare il throughput come l'esecuzione fuori ordine (Out Of Order Execution)
- In seguito sarà considerata una strategia alternativa per velocizzare i calcoli

# Paradigma di elaborazione SIMD 1/4

- Una singola istruzione SIMD elabora multipli dati (eseguendo sugli stessi dati la medesima operazione) memorizzati in registri speciali/estesi della CPU



# Paradigma di elaborazione SIMD 2/4

## Introduzione

- I registri della CPU sono di taglia estesa (eg, 64, 128, 256, 512) rispetto ai registri dell'architettura base e contengono più dati (eg, 2, 4, 8, 16 bit) compattati



Register



XMM (extended register)

**ADD R1, R2, R3**

$R1 = R2 + R3$

**XADD XR1, XR2, XR3**

$XR1[0] = XR2[0] + XR3[0]$

$XR1[1] = XR2[1] + XR3[1]$

$XR1[2] = XR2[2] + XR3[2]$

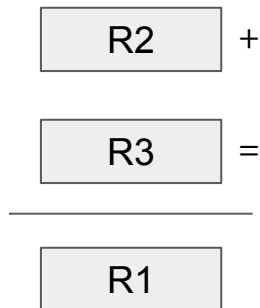
$XR1[3] = XR2[3] + XR3[3]$

# Paradigma di elaborazione SIMD 3/4

## SISD

**ADD R1, R2, R3**

$R1 = R2 + R3$



## SIMD

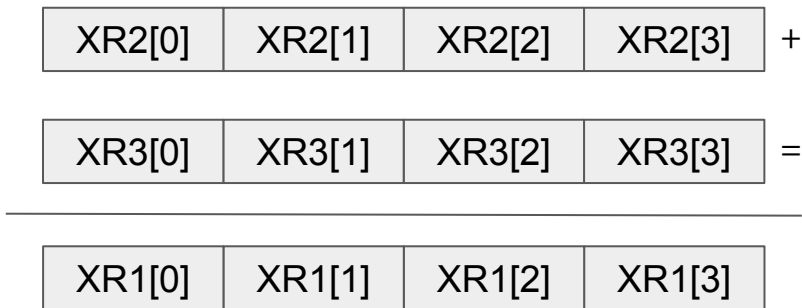
**XADD XR1, XR2, XR3**

$XR1[0] = XR2[0] + XR3[0]$

$XR1[1] = XR2[1] + XR3[1]$

$XR1[2] = XR2[2] + XR3[2]$

$XR1[3] = XR2[3] + XR3[3]$



# Paradigma di elaborazione SIMD 4/4

- Incrementa il parallelismo agendo a livello dei dati
- Può essere utilizzata con tutte le strategie menzionate (eg, pipelining)
- Richiede un numero di modifiche hardware limitato vs SISD come l'integrazione di ALU addizionali con impatto modesto in termini maggiori di risorse utilizzate
- Supportato da quasi tutte le ISA più diffuse (eg, x86, ARM e RISC-V)
- Tuttavia, ISA diverse hanno set di istruzioni SIMD differenti sebbene spesso con funzionalità simili
- Anche all'interno della stessa ISA, possono esserci diversi set di istruzioni SIMD anche se la tendenza è quella di supportare tutte le estensioni SIMD precedenti, principalmente per ragioni di compatibilità software
- In alcuni casi (eg, x86) le estensioni SIMD iniziali (eg, MMX) sono deprecate



# Evoluzione x86 1/4

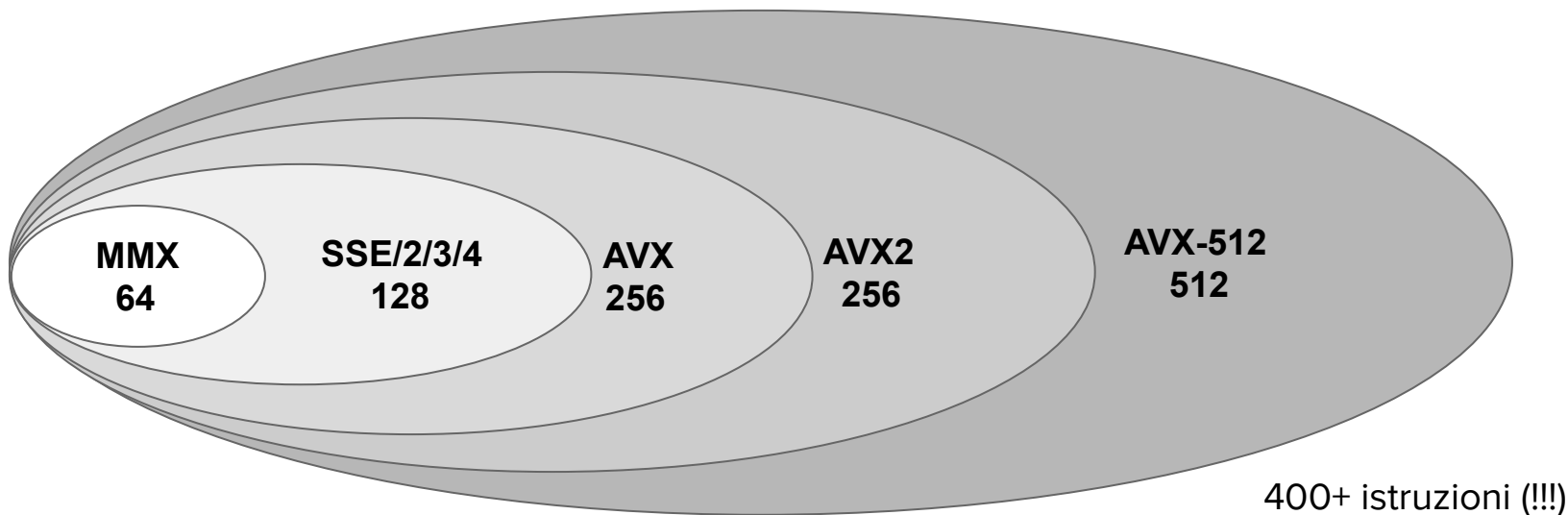
- Il paradigma SIMD è stato proposto per la prima volta negli anni '60 e utilizzato principalmente per realizzare sistemi per l'elaborazione parallela
- L'ampia diffusione mediante CPU general-purpose si è avuta però solo nei primi anni 90 con l'uso massiccio di dati multimediali (audio, immagini) e operazioni intensive (encoding, real-time processing, etc) su tali dati
- Sebbene non il primo set di istruzioni SIMD, l'estensione MMX (MultiMedia eXtension, 1996) proposta da Intel per le proprie CPU ha avuto larga diffusione
- **MMX** consiste in 57 istruzioni, prevalentemente per l'elaborazione di dati interi, che estendono l'ISA x86 utilizzando registri multimediali a 64 bit ricavati dai registri floating della CPU

## Evoluzione x86 2/4

- 1988 - AMD **3DNow!**, ora deprecata, supporta, tra le varie cose, per la prima volta operazioni con dati floating-point
- 1999 - Intel **SSE** (Streaming SIMD Extension): 70 nuove istruzioni, registri a 128 bit, supporto per operazioni con dati floating-point
- 2000 - Intel **SSE2** (Streaming SIMD Extension 2): 144 nuove istruzioni, registri a 128 bit, supporto per operazioni con dati floating-point
- 2004 - Intel **SSE3** (Streaming SIMD Extension 3): 13 nuove istruzioni, registri a 128 bit, supporto per operazioni con dati floating-point, supporto per operazioni *orizzontali* su dati all'interno dello stesso registro
- 2007 - Intel **SSE4\*** (Streaming SIMD Extension 4): 54 nuove istruzioni, registri a 128 bit, supporto per operazioni con dati floating-point

## Evoluzione x86 3/4

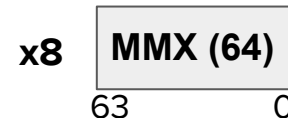
- 2008 - Intel **AVX** (Advanced Vector Extensions): registri a 256 bit, floating-point
- 2013 - Intel **AVX2** (Advanced Vector Extensions 2): registri a 256 bit, anche interi
- 2016 - Intel **AVX-512** (Advanced Vector Extensions-512): registri a 512 bit



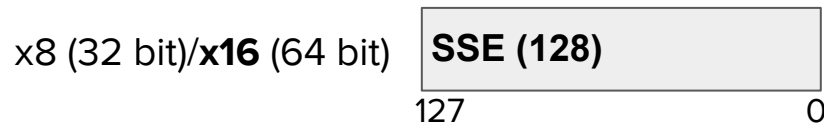
# Evoluzione x86 4/4

La dimensione dei registri multimediali (o estesi varia) in funzione della specifica implementazione: MMX x8, SSE x16 (64 bit mode), AVX x16

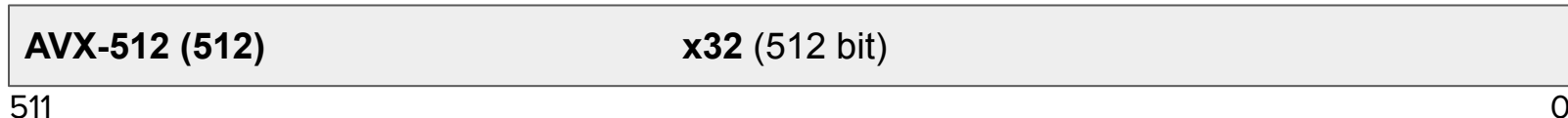
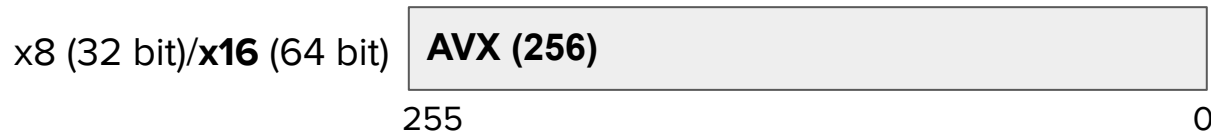
$\frac{1}{8}$  vs AVX-512



$\frac{1}{4}$  vs AVX-512



$\frac{1}{2}$  vs AVX-512



# Evoluzione SIMD per CPU x86



MMX: Pentium, 1996



3DNow!: AMD K6, 1998



SSE: Pentium III, 2000



SSE2/3: Pentium 4, 2004



SSE4: Intel Core, 2007



AVX: Sandy Bridge, 2008



AVX2: Haswell, 2013



AVX-512: Skylake, 2016

## x86: tipi di dati supportati

- I tipi di dati possono essere byte (8 bit), interi a 16 bit, interi a 32 bit, single precision floating-point (32 bit), double precision floating-point (64 bit)
- Per i tipi byte e interi sono supportate operazioni con dati signed e unsigned
- Non tutte le istruzioni o estensioni supportano tutti i tipi di dato (eg, MMX solo interi 8, 16, 32, 64 bit)
- In generale non c'è molta *ortogonalità* nell'ISA x86
- Estensione MMX è attualmente considerata deprecata/obsoleta

<https://www.intel.com/content/www/us/en/support/articles/000005779/processors.html>

<https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-10/intrinsics.html>

# ARM

- Anche i microprocessori ARM supportano istruzioni SIMD mediante il set di istruzioni NEON
- Inizialmente la dimensione dei registri era di 32 o 64 bit
- Attualmente, registri di dimensione fino a 128 bit
- Tipi di dato: interi a 8 bit, 16 bit, 32 bit e 64 bit signed e unsigned e floating-point a 16 e 32 bit
- Estensione per HPC (SVE ed SVE2) con dati di dimensioni variabili da 128 a 2048 bit con step di 128 bit

<https://developer.arm.com/documentation/100076/0100/Instruction-Set-Overview/Overview-of-the-Arm-Architecture/Advanced-SIMD?lang=en>

# RISC-V

- L'ISA prevede un'estensione SIMD, denominata RISC-V Vector Extension (RVV)
- RVV definisce 32 registri di dimensione pari a una potenza di 2 non prefissata ma configurabile dai progettisti della CPU
- I dati possono essere di taglia pari a una potenza di 2, con valore maggiore o uguale a 8
- Maggiori informazioni e dettagli nel seminario del Prof. Giuseppe Tagliavini

<https://eupilot.eu/wp-content/uploads/2022/11/RISC-V-VectorExtension-1-1.pdf>



# Operazioni SIMD: introduzione

- Ogni estensioni/produttore prevede proprie istruzioni SIMD e non esiste una uniformità tra le varie implementazioni
- Tuttavia, le operazioni supportate dalle varie estensioni sono in qualche modo simili come:
  - rendere agevole e veloce la lettura/scrittura, il packing/unpacking e interleaving/de-interleaving di dati
  - eseguire operazioni comuni nell'ambito del image/signal processing come somme, sottrazioni, moltiplicazioni, etc
  - individuare massimi e minimi, replicare elementi all'interno dei registri estesi, consentire operazioni con saturazione e operazioni molto comuni come SAD (Sum of Absolute Differences)

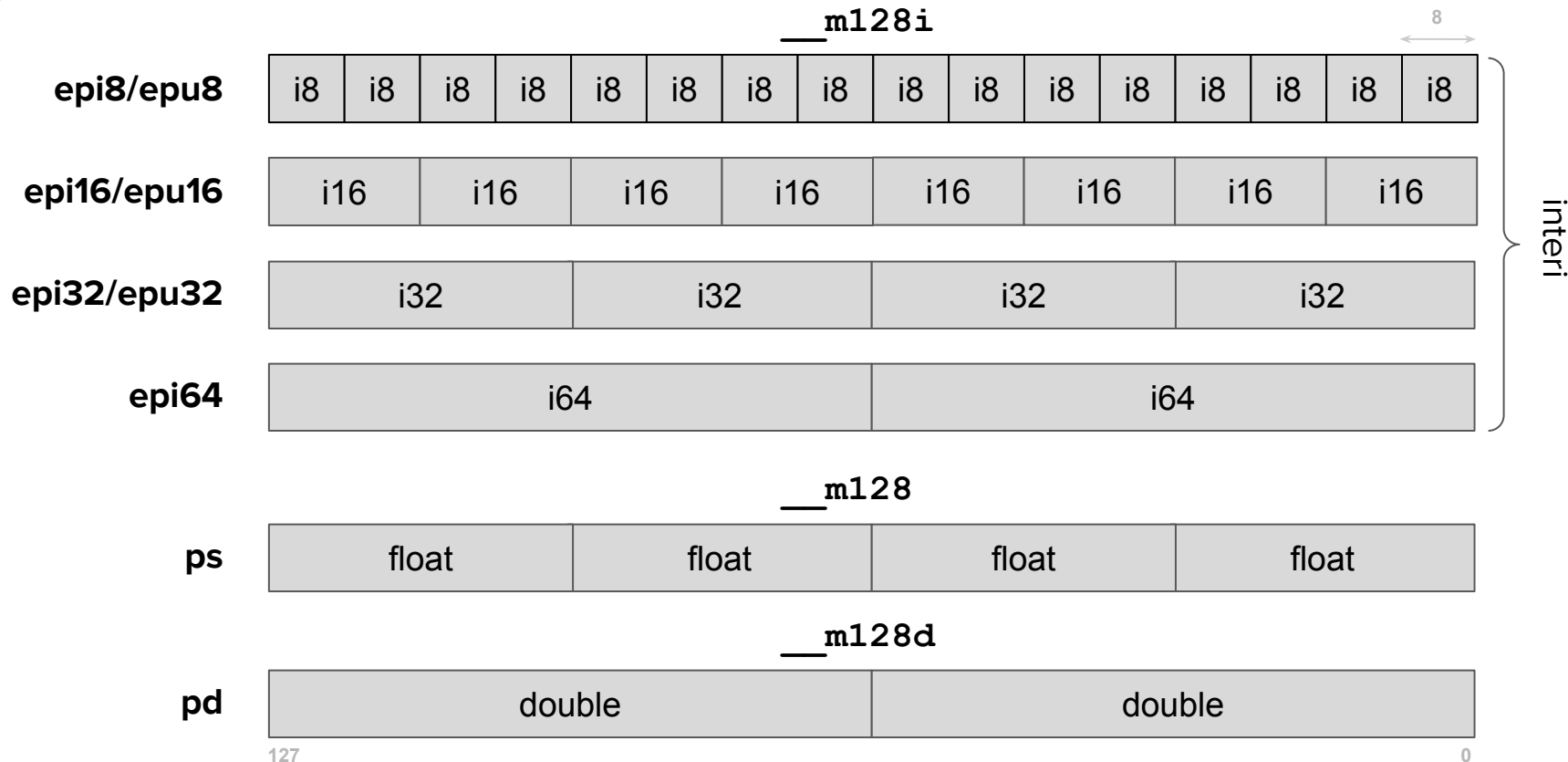
# Istruzioni Assembly vs Intrinsics

- Le operazioni SIMD sono disponibili come istruzioni in linguaggio assembly
- E' possibile scrivere codice in linguaggi ad alto livello inserendo, per le porzioni desiderate, codice assembly
- Tuttavia, una soluzione meno complicata per il programmatore consiste nell'utilizzare "*Intrinsics*"
- Gli *intrinsics* non sono altro che funzioni online che corrispondono a singole o sequenze di istruzioni SIMD in linguaggio assembly
- Il vantaggio principale è che il codice è più leggibile e semplice da scrivere
- Lo svantaggio è in alcune circostanze non si ha il controllo che si avrebbe con istruzioni assembly (eg, non è possibile specificare quali sono esattamente i registri della CPU utilizzati in una determinata istruzione)

# Tipi di dati presenti in x86 SSE 1/2

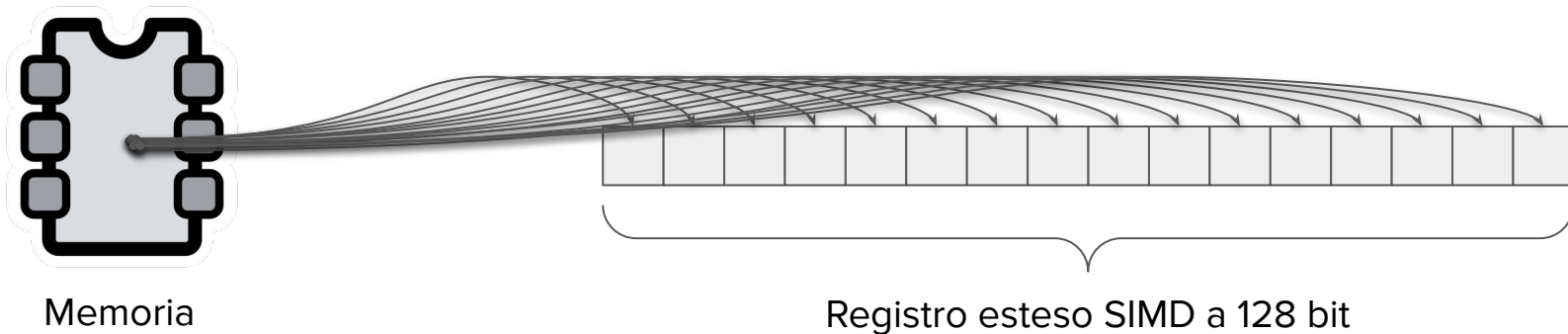
- I registri SSE a 128 bit possono essere utilizzati con i seguenti tipi di dato:
  - `__m128i` (interi)
    - 16x8 bit – **epi8** (signed), **epu8** (unsigned)
    - 8x16 bit – **epi16** (signed), **epu16** (unsigned)
    - 4x32 bit – **epi32** (signed), **epu32** (unsigned)
    - 2x64 bit – **epi64** (signed)
  - `__m128` (float)
    - 4x32 bit – tipicamente denominati **ps** (floating-point a singola precisione)
  - `__m128d` (double)
    - 2x64 bit – tipicamente denominati **pd** (floating-point a doppia precisione)

# Tipi di dati presenti in x86 SSE 2/2



## Lettura dalla memoria 1/2

- Operazioni strettamente necessarie riguardano la lettura di blocchi di dati contigui dalla memoria verso i registri estesi SIMD
- Esempio, nell'ipotesi di avere registri estesi a 128 bit: dato un array di byte in memoria, è possibile trasferire 16 byte in un registro con una singola istruzione



## Lettura dalla memoria 2/2

- L'istruzione seguente (SSE) carica dalla memoria 128 bit a partire dall'indirizzo **mem\_addr** che deve essere allineato con la taglia del registro (16 byte)
- I dati letti nel registro **\_\_mm128i** possono essere byte (16 elementi), interi a 16 bit (8 elementi), interi a 32 bit (4 elementi)

```
__m128i _mm_load_si128 (__m128i const* mem_addr)
```

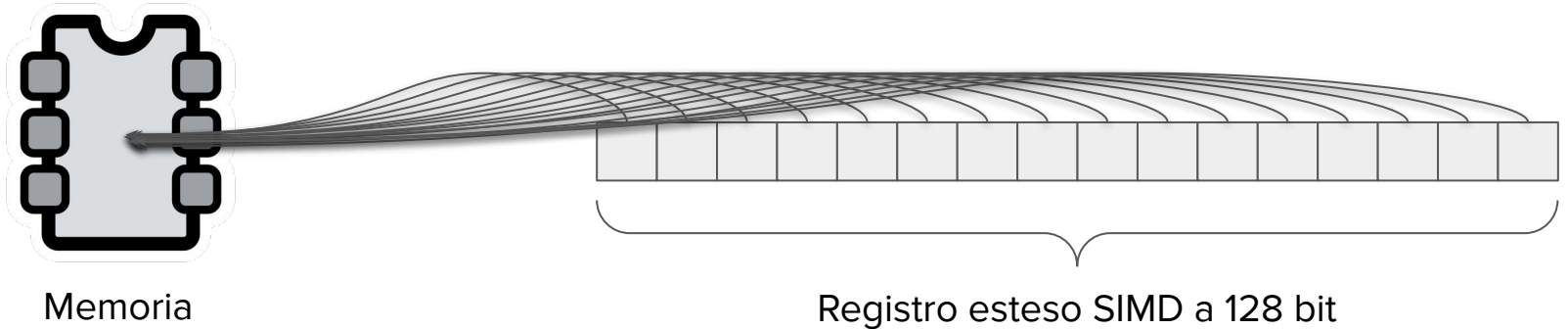
- Esistono anche istruzioni (SSE) per leggere dati floating-point a singola (32bit, 4 elementi in **\_\_m128**) e a doppia precisione (64 bit, 2 elementi in **\_\_mm128d**)

```
__m128 _mm_load_ps (float const* mem_addr)
```

```
__m128d _mm_load_pd (double const* mem_addr)
```

# Scrittura in memoria 1/2

- La scrittura di un registro esteso SIMD in posizioni contigue della memoria è un'altra operazione spesso necessaria
- Esempio, nell'ipotesi di avere registri estesi a 128 bit: dato un registro contenente 16 byte è possibile scriverlo in memoria con una singola istruzione SIMD



## Scrittura in memoria 2/2

- L'istruzione seguente (SSE) scrive in memoria 128 bit di interi a partire dall'indirizzo **mem\_addr** allineato con la taglia del registro (16 byte)
- I dati scritti in memoria possono essere byte (16 elementi), interi a 16 bit (8 elementi), interi a 32 bit (4 elementi)

```
void _mm_store_si128 (__m128i* mem_addr, __m128i a)
```

- Esistono anche istruzioni (SSE) per scrivere dati floating-point a singola (32bit, 4 elementi in **\_\_m128**) e a doppia precisione (64 bit, 2 elementi in **\_\_mm128d**)

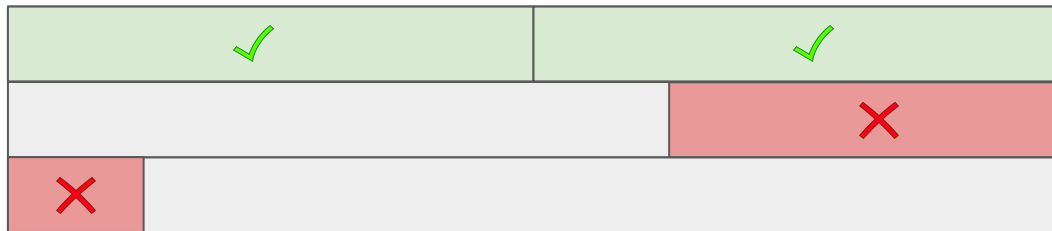
```
void _mm_store_ps (float* mem_addr, __m128 a)
```

```
void _mm_store_pd (double* mem_addr, __m128d a)
```



# Operazioni SIMD: letture/scrittura e allineamento 1/3

- In realtà, il trasferimento dati tra la memoria e i registri estesi avviene attraverso delle memorie *cache*
- Le cache organizzano in propri dati per linee che contengono porzioni della memoria (pari a una potenza di 2) a partire da un indirizzo allineato con la taglia delle linee
- Per questa ragione è fondamentale leggere e scrivere a indirizzi allineati con la taglia dei registri per non incorrere in penalità o (in alcuni casi) fault



linee cache

## Operazioni SIMD: letture/scrittura e allineamento 2/3

- Il problema è facilmente risolvibile memorizzando i dati in memoria a indirizzi allineati con la taglia dei registri estesi
- Anche l'accesso ai dati dovrebbe avvenire con lo stesso vincolo e questo potrebbe richiedere qualche accorgimento in più nella scrittura del codice
- Alcune estensioni SIMD non consentono accessi disallineati: tuttavia, anche in questi casi si potrebbe incorrere in penalizzazioni (maggior tempo di esecuzione)
- I linguaggi di programmazione consentono di forzare l'allineamento dei dati
- Esempio, in C/C++ un array allineato a un indirizzo multiplo di 16 può essere dichiarato come segue:

```
int A[8] __attribute__((aligned(16))); // 16 byte (128 bit) aligned
```

# Operazioni SIMD: letture/scrittura e allineamento 3/3

- E' sempre bene memorizzare ed accedere ai dati in modo allineato
- Tuttavia, questo in alcuni casi questo potrebbe non essere possibile
- Nonostante questo, è sempre possibile accedere ai dati in modo allineato e poi combinare le informazioni con istruzioni predisposte per questa finalità
- Tale approccio potrebbe introdurre un certo grado di inefficienza dovuto alle operazioni di manipolazione dei dati ma è comunque percorribile
- In alcuni casi (eg, x86 SSE) esistono alcune (poche) istruzioni per accedere a dati non allineati

```
__m128i _mm_loadu_si128 (__m128i const* mem_addr)
```

```
void _mm_storeu_si128 (__m128i* mem_addr, __m128i a)
```

- Nonostante questo, l'accesso a dati allineati è sempre la strada preferibile

# Allocazione dinamica della memoria

- Considerando le problematiche inerenti l'allineamento dei dati, SSE dispone di funzioni per allocare e deallocare dinamicamente memoria
- E' possibile specificare il tipo di allineamento desiderato (parametro size\_t align)

```
void* _mm_malloc (size_t size, size_t align)
```

```
void _mm_free (void * mem_addr)
```

- Esempio, allocazione di un array di 1024 byte signed con allineamento 16

```
char *SIMD_array = (char *) _mm_malloc(1024, 16);  
...  
...  
_mm_free(SIMD_array);
```

# Gestione cache

- E' possibile invalidare in tutti i livelli di una cache la linea che contiene un determinato indirizzo di memoria

```
void _mm_clflush (void const* p)
```

- L'utilizzo di questa istruzione andrebbe attentamente valutata perchè può avere un impatto molto negativo sulle prestazioni
- E' anche possibile suggerire (non essendo una vera istruzione) al compilatore di eseguire un prefetch di dati nella cache in accordo a 4 strategie (parametro i=0, 1, 2, 3)

```
void _mm_prefetch (char const* p, int i)
```

## Lecture/scrittura: istruzioni *stream* (x86)

- Nel caso x86 (SSE) sono disponibili delle istruzioni (*stream*) che eseguono operazioni con la memoria senza modificare il contenuto delle cache
- Questo per evitare di inserire nella cache dati che non saranno utilizzati in seguito a discrezione del programmatore
- Un esempio è la seguente istruzione

```
__m128i _mm_stream_load_si128 (void* mem_addr)
```

che carica 128 bit di dati (interi) senza modificare il contenuto della cache, a meno che il dato non sia già presente (in tal caso aggiorna il contenuto della cache)

- L'istruzione richiede che i dati siano allineati con la taglia del registro (16 byte) altrimenti viene generata un'eccezione

# Esempio: copia di un vettore 1/3

Problema: copiare gli elementi di un vettore A in un vettore B

```
#pragma GCC target("sse4.2")    // Needed with OneCompiler
#include <stdio.h>
#include <immintrin.h>

#define VECTOR_LENGTH 32
#define SSE_DATA_LANE 16
#define DATA_SIZE 1

void print_output(char *A, char *B, int length)
{
    for (int i=0; i<VECTOR_LENGTH; i++)
    {
        printf("A[%d]=%d, B[%d]=%d\n",i,A[i],i,B[i]);
    }
}
```

## Esempio: copia di un vettore 2/3

```
int main()
{
    char A[VECTOR_LENGTH] __attribute__((aligned(SSE_DATA_LANE))); // 16 byte (128 bit) aligned
    char B[VECTOR_LENGTH] __attribute__((aligned(SSE_DATA_LANE))); // 16 byte (128 bit) aligned

    __m128i *p_A = (__m128i*) A;
    __m128i *p_B = (__m128i*) B;

    __m128i XMM_SSE_REG;

    for (int i=0; i<VECTOR_LENGTH; i++)
    {
        A[i]=i;
        B[i]=0;
    }
```



## Esempio: copia di un vettore 3/3

```
printf("\nInput data:\n");  
print_output(A,B,VECTOR_LENGTH);  
  
for (int i=0; i<VECTOR_LENGTH/SSE_DATA_LANE/DATA_SIZE; i++)  
{  
    XMM_SSE_REG = _mm_load_si128 (p_A+i);  
    _mm_store_si128 (p_B+i, XMM_SSE_REG);  
}  
  
printf("\nOutput data:\n");  
print_output(A,B,VECTOR_LENGTH);  
  
}
```

# Ambienti di sviluppo e compilazione

- Compilazione: `g++ -msse4 copy_array.cpp -o copy_array`  
(-msse4 opzionale in questo caso)
- La soluzione ottimale consiste nell'utilizzare un ambiente di sviluppo nativo x86 Linux, Windows o Mac (solo x86 con SSE)
- In alternativa (eg, utenti Mac con processori M) è possibile utilizzare compilatori online come OneCompiler (<https://onecompiler.com/>) o (meglio) Compiler Explorer (<https://godbolt.org/>) che consente funzionalità più avanzate
- In ogni caso, in un sistema multi-tasking, le misure dei tempi di elaborazione sono problematiche perché influenzate dalla gestione dei processi da parte del sistema operativo

# Performance counters 1/2

- Sebbene il tempo di elaborazione di un processo sia difficilmente ripetibile in un sistema multi-tasking, le CPU moderne offrono alcuni utili strumenti
- In particolare, dispongono di contatori che sono aggiornati con il clock della CPU
- Tali contatori possono essere letti per avere indicazioni temporali, conoscendo la frequenza del processore, o in termini di cicli di clock di una determinata porzione di codice
- Per esempio, nei sistemi x86 sono presenti dei contatori a 64 bit (free-running) che possono essere letti, per ottenere un *timestamp*, mediante istruzioni come:

```
unsigned __int64 __rdtsc();
```

## Performance counters 2/2

- La rilevazione del numero di cicli di clock associati all'esecuzione di una porzione di codice può essere calcolato come segue (`performance_counters.cpp`):

```
#include <stdio.h>
#include <immintrin.h>

#ifdef _WIN32
#include <intrin.h>
#else
#include <x86intrin.h>
#endif

int main() {
    u_int64_t clock_counter_start = __rdtsc();
    // porzione codice da valutare
    u_int64_t clock_counter_end = __rdtsc();
    printf("\nElapsed clocks: %lu\n", clock_counter_end-clock_counter_start);
}
```

# Autovettorizzazione del codice 1/2

- I compilatori moderni possono generare un codice eseguibile che sfrutta il paradigma SIMD e altre forme di ottimizzazione identificando flussi di elaborazione parallela:
  - `g++ -O3 -msse2 -ftree-vectorize code.cpp -o code_ex`
  - i compilatori possono essere molto efficaci ma è difficile pensare di sostituire l'intervento umano, soprattutto per elaborazioni complesse
- In generale, in gcc/g++, il grado di ottimizzazione può essere impostato mediante il flag `-Ox`:
  - `-O0` nessuna ottimizzazione
  - `.....`
  - `-O3` massima ottimizzazione (`-ftree-vectorize` inutile, autovettrizzazione default)

## Autovettorizzazione del codice 2/2

- Numero di cicli di clock codice scalare vs SIMD (file performance\_counters.cpp)  
con varie ottimizzazioni e vettori di 1024 elementi):
  - -O0 41706 vs 881
  - -O1 6002 vs 350
  - -O2 11132 vs 185
  - -O3 382 vs 67 (\*)
  - -O3 -msse2 -ftree-vectorize -ftree-loop-linear 374 vs 71 (\*)

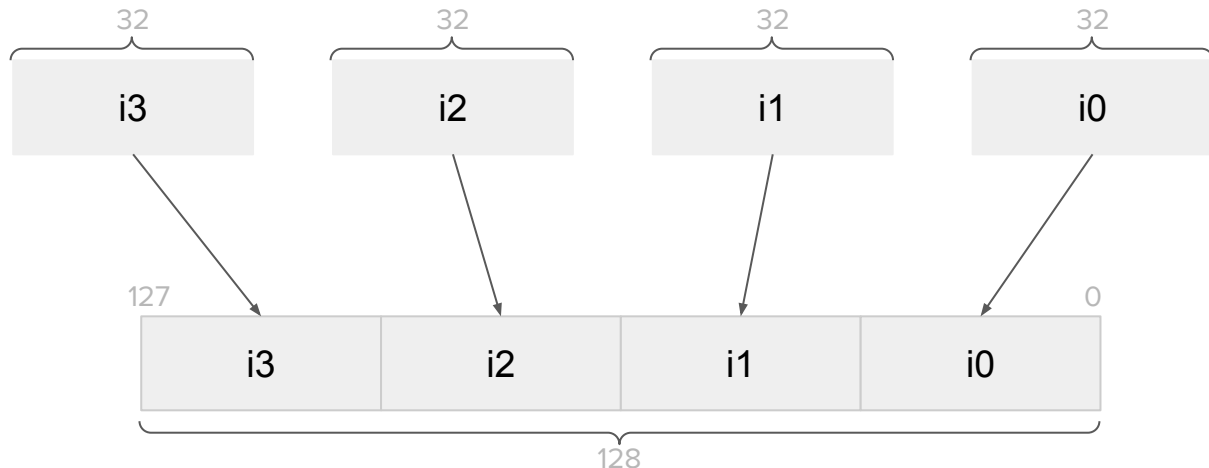
# Predisposizione (non efficiente) dei dati per calcolo vettoriale 1/2

- I dati da elaborare non sono in origine sempre memorizzati in modo tale da rendere efficiente l'utilizzo di operazioni parallele tra registri estesi
- Per questa ragione, tutte le estensioni SIMD prevedono la possibilità di manipolare i dati in modo da renderli elaborabili parallelamente
- Il modo più semplice, ma poco efficiente, consiste nell'inserire in un registro esteso una serie di elementi indicati nell'istruzione stessa; ad esempio (SSE), l'istruzione seguente (in realtà una sequenza di istruzioni) inserisce nel registro destinazione a 128 bit i quattro valori interi a 32 bit e0, e1, e2, e3:

```
__m128i _mm_set_epi32 (int e0, int e1, int e2, int e3)
```

## Predisposizione (non efficiente) dei dati per calcolo vettoriale 2/2

`__m128i _mm_set_epi32 (int i0, int i1, int i2, int i3)`



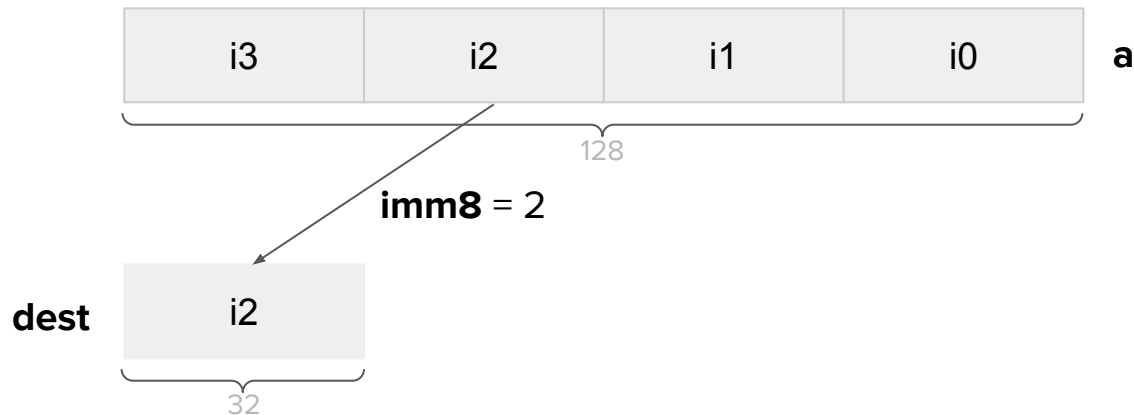
- Istruzioni (in realtà, una sequenze di istruzioni) analoghe esistono per gli altri tipi di dato supportati dall'ISA x86
- Attenzione all'ordine tra lista parametri e collocazione nel registro esteso (vedi sopra)



# Estrazione di dati dai registri estesi

- E' possibile estrarre un singolo valore da un registro esteso
- Il valore è selezionato in base a un valore immediato presente nell'istruzione
- L'istruzione esiste per vari formati di dati (eg, interi a 8 bit, 16 bit, 32 bit)

```
int _mm_extract_epi32 (__m128i a, const int imm8)
```



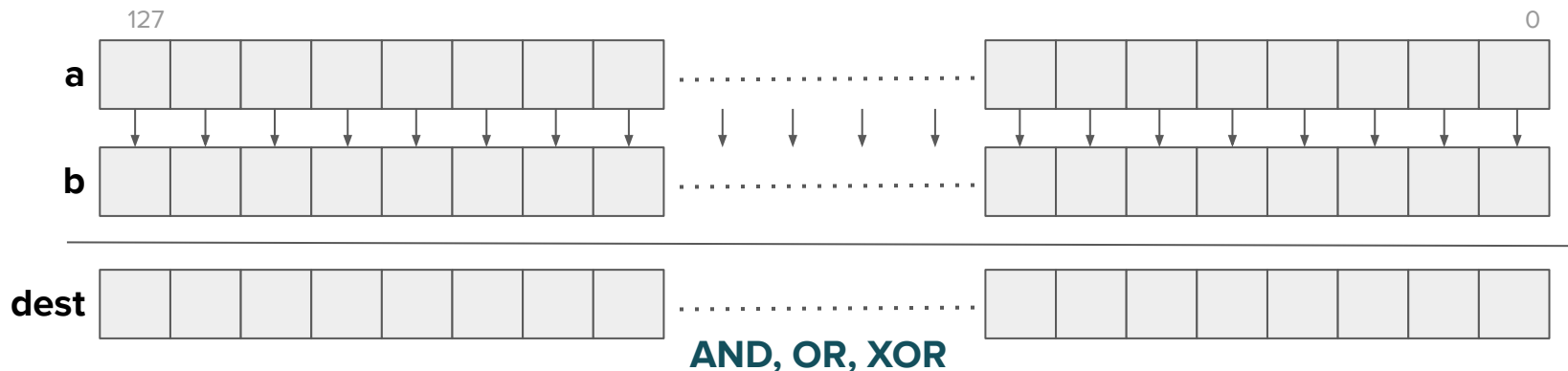
# Istruzioni logiche: AND, OR, XOR

- Le istruzioni SIMD supportano operazione logiche bit a bit (*bitwise*) AND, OR, XOR

`__m128i _mm_and_si128 (__m128i a, __m128i b)`

`__m128i _mm_or_si128 (__m128i a, __m128i b)`

`__m128i _mm_xor_si128 (__m128i a, __m128i b)`

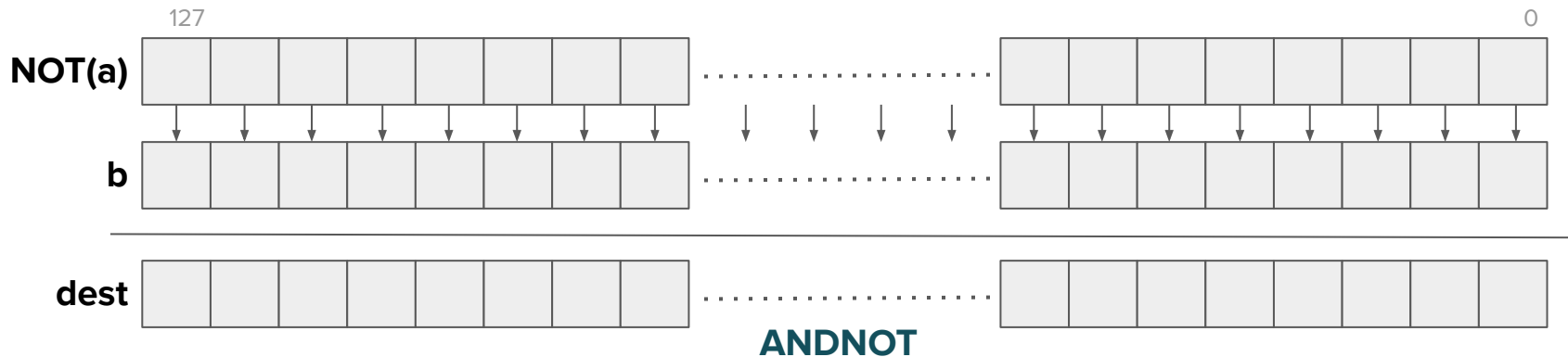


# Istruzione logica: ANDNOT

- La seguente istruzione, prima esegue NOT(a) e successivamente l'AND con b
- Ovvero,  $\text{dest} = \text{NOT}(a) \text{ AND } b$  eseguito in modo *bitwise*

**\_\_m128i \_\_mm\_andnot\_si128 (\_\_m128i a, \_\_m128i b)**

- Può essere utile, così come altre istruzioni logiche, per *mascherare/selezionare* dei bit in funzione di un risultato precedente



## Istruzioni di shift 1/2

- Le operazioni di shift – Left Logic (SLL), Right Logic (SRL) Right Arithmetic (SRA) – sono eseguite, con gli stessi parametri, su tutti gli elementi del registro
- L'entità dello shift è indicata attraverso un immediato o un registro esteso utilizzando i bit meno significativi necessari per lo specifico tipo di dato utilizzato
- I tipi di dati supportati sono interi a 16, 32, 64
- L'istruzione seguente esegue uno shift a sinistra di ciascun elemento del vettore di interi a 16 bit **a** di un numero di bit indicato da **imm8**

**\_\_m128i \_mm\_slli\_epi16 (\_\_m128i a, int imm8)**

<b>A</b>	7	6	5	4	3	2	1	0
----------	---	---	---	---	---	---	---	---

<b>dest</b>	14	12	10	8	6	4	2	0	<b>imm8 = 1</b>
-------------	----	----	----	---	---	---	---	---	-----------------

## Istruzioni di shift 2/2

- L'istruzione seguente esegue uno shift a sinistra di ciascun elemento del vettore di interi a 16 bit **a** di un numero di bit indicato dagli ultimi bit di **count**

**\_\_m128i \_mm\_sll\_epi16 (\_\_m128i a, \_\_m128i count)**

**A**

7	6	5	4	3	2	1	0
---	---	---	---	---	---	---	---

**count**

2
---

**dest**

28	24	20	16	12	8	4	0
----	----	----	----	----	---	---	---

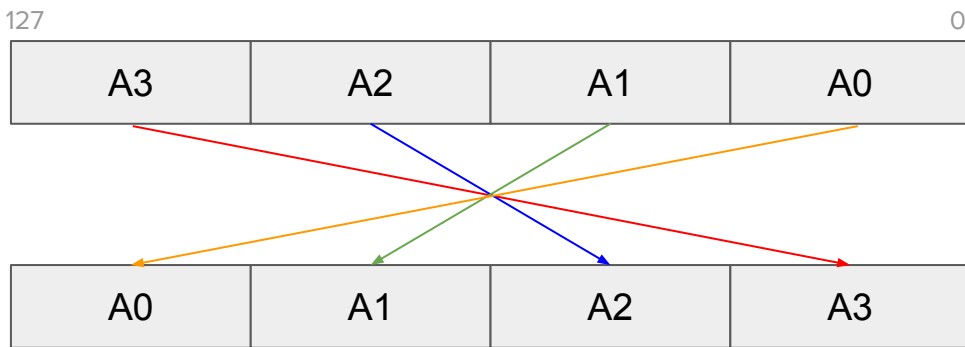
- Esiste anche la possibilità di eseguire lo shift dell'intero registro a 128 bit di una quantità di **BYTE** pari al valore dell'immediato **count**

**\_\_m128i \_mm\_slli\_si128 (\_\_m128i a, int imm8)**

# Istruzioni di shuffling

- Sono istruzioni che consentono di *smistare* i dati di un registro sorgente in posizioni differenti nel registro destinazione in funzione di una maschera
- Utili quando si desiderano disporre in un determinato ordine i valori presenti in un registro esteso

**`__m128i _mm_shuffle_epi32 (__m128i a, int imm8)`**



$\text{imm8} = 27_{10} = \text{00011011}_2$

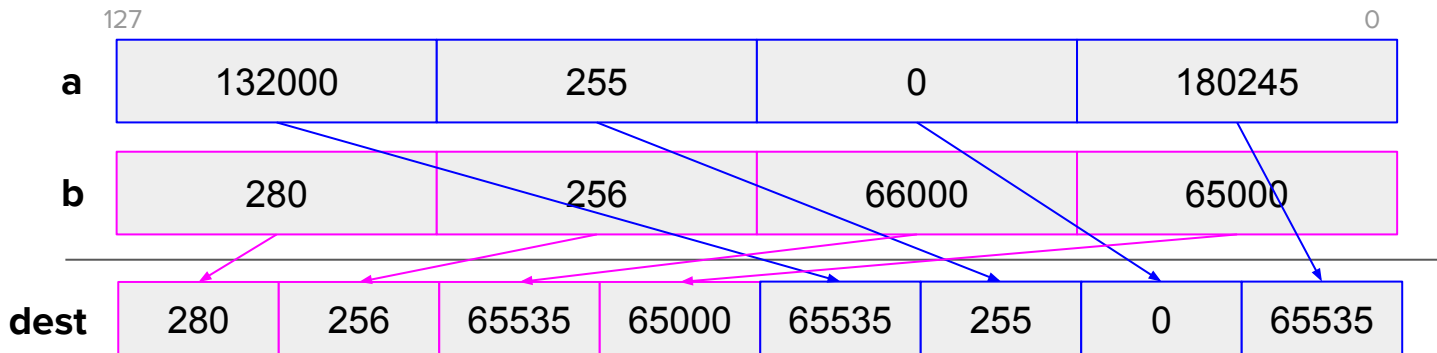
# Istruzioni di packing unsigned

- Istruzioni che *impacchettano* in un registro il contenuto di due registri effettuando il casting (con saturazione) a un tipo di dato di dimensione inferiore (32→16, 16→8)

**\_\_m128i \_mm\_packus\_epi16 (\_\_m128i a, \_\_m128i b)**

**\_\_m128i \_mm\_packus\_epi32 (\_\_m128i a, \_\_m128i b)**

- Esempio: `dest = _mm_packus_epi32(a, b);`



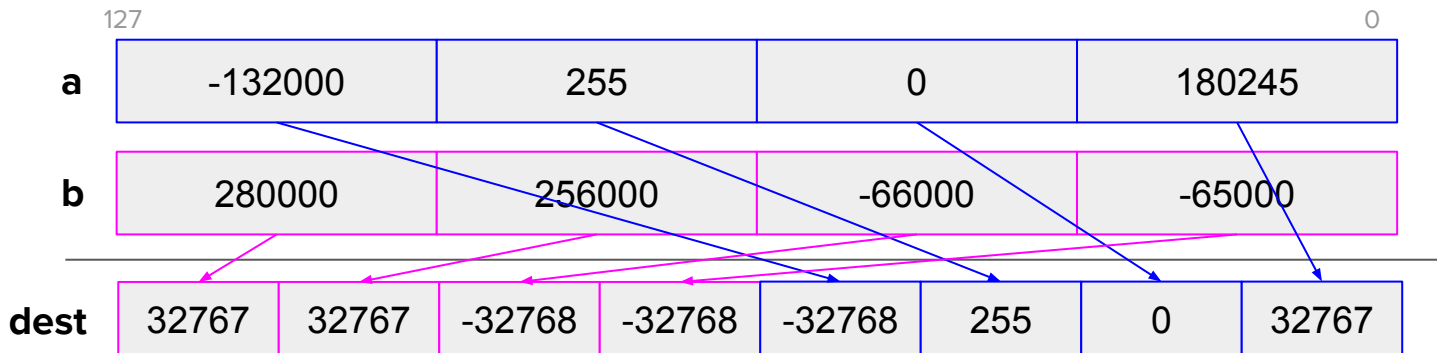
# Istruzioni di packing signed

- Istruzioni di *impacchettamento* (32→16, 16→8) con saturazione analoghe alle precedenti ma per dati signed

**\_\_m128i \_mm\_packs\_epi16 (\_\_m128i a, \_\_m128i b)**

**\_\_m128i \_mm\_packs\_epi32 (\_\_m128i a, \_\_m128i b)**

- Esempio: `dest = _mm_packs_epi32(a, b);`



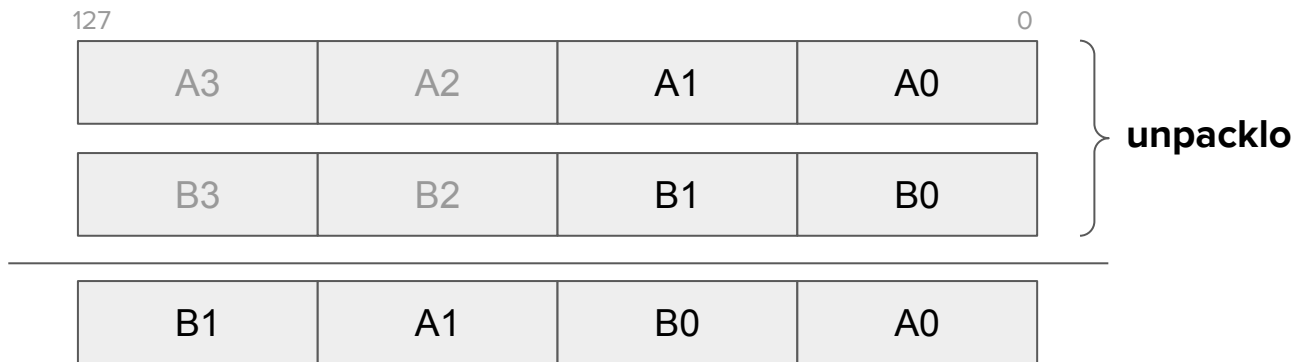


# Istruzioni di unpacking

- Questo gruppo di istruzioni consentono di *combinare* il contenuto di due registri attraverso *interleaving*
- Disponibili per tutti i tipi di dato intero (8, 16, 32, 64 bit)
- Nel caso di dati a 32 bit:

**\_\_m128i \_mm\_unpacklo\_epi32 (\_\_m128i a, \_\_m128i b)**

**\_\_m128i \_mm\_unpackhi\_epi32 (\_\_m128i a, \_\_m128i b)**



# Conversione tra tipi di dati: istruzioni cvt

- Talvolta è necessario passare da una rappresentazione ad un'altra (eg, int8 → int16)
- Già vista la compattazione per ridurre la dimensione dei dati mediante *packing*
- Il passaggio da un tipo di dato a uno, tipicamente di dimensione superiore, è gestito attraverso la classe di istruzioni cvt

Esempio, la conversione da valori a 8 bit signed a 16 bit signed può essere eseguita mediante l'istruzione seguente (che agisce sugli 8 byte meno significativi):

**\_\_m128i \_mm\_cvtepi8\_epi16 (\_\_m128i a)**

<b>a</b>	-20	64	50	4	-45	127	-77	6	-87	7	68	-95	0	15	-127	3
----------	-----	----	----	---	-----	-----	-----	---	-----	---	----	-----	---	----	------	---

<b>dest</b>	-87	7	68	-95	0	15	-127	3
-------------	-----	---	----	-----	---	----	------	---

## Istruzioni di blending 1/2

- Questo gruppo di istruzioni consentono di *combinare* contenuto di due registri in funzione di un valore immediato (`blend`) o un registro (`blendv`)
- Disponibili per i tipi di dati interi a 8 (`blendv`) e 16 bit (`blend`)
- Nel caso dell'istruzione `blend` (solo per `int16`):

**`__m128i _mm_blend_epi16 (__m128i a, __m128i b, const int imm8)`**

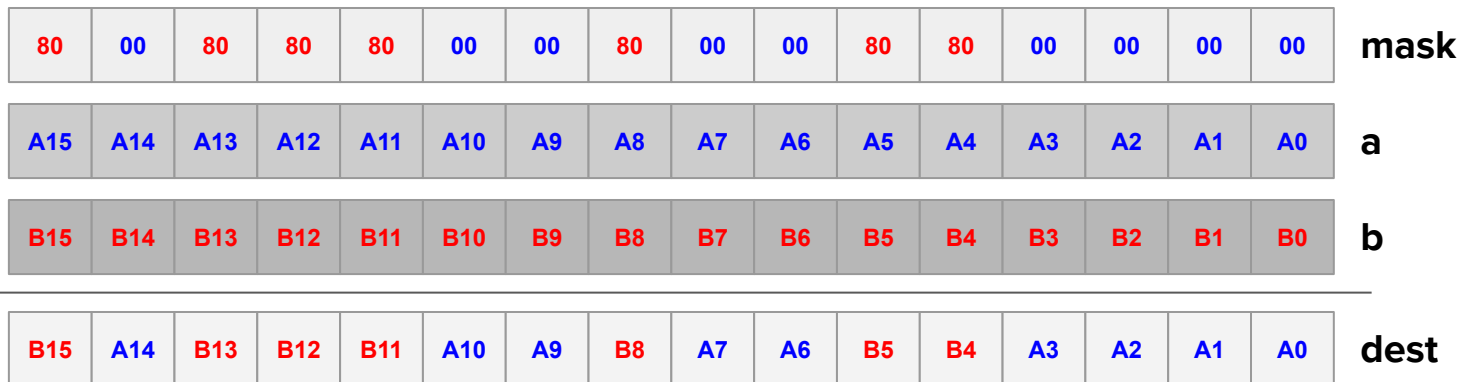
**`imm8 = 11001010`**

A7	A6	A5	A4	A3	A2	A1	A0	a
B7	B6	B5	B4	B3	B2	B1	B0	b
B7	B6	A5	A4	B3	A2	B1	A0	dest

## Istruzioni di blending 2/2

- Questo gruppo di istruzioni consentono di *combinare* contenuto di due registri in funzione di un valore immediato (`blend`) o un registro (`blendv`)
- Disponibili per i tipi di dati interi a 8 (`blendv`) e 16 bit (`blend`)
- Nel caso dell'istruzione `blendv` (solo per byte):

**`__m128i _mm_blendv_epi8 (__m128i a, __m128i b, __m128i mask)`**



# Copia con inserimento di un valore

Questa istruzione consente di copiare un registro in un altro modificando un singolo elemento del registro sorgente

**\_\_m128i \_mm\_insert\_epi16 (\_\_m128i a, int i, const int imm8)**

Copia il registro sorgente **a** nel registro destinazione inserendo il valore **i** nella posizione specificata dal valore dell'immediato **imm8**; l'intero i può rappresentare un valore a 8, 16, 32 o 64 bit

Esempio: **dest** = \_mm\_insert\_epi16 (**a**, 68, 3)

								16
<b>A</b>	87	7	1245	95	-42	-15	127	3
<b>dest</b>	87	7	1245	95	<b>68</b>	-15	127	3

## Istruzioni di confronto 1/3

- Le istruzioni di confronto sono fondamentali nell'esecuzione scalare di un qualsiasi codice al fine di poter compiere operazioni differenti sui dati in base all'esito del confronto
- Nel caso SIMD non è possibile avere flussi di elaborazioni diverse in funzione dei multipli confronti effettuati sui dati e si sfruttano delle maschere per poter compiere o meno delle operazione sui dati
- A tal fine, sono confrontati tra loro elementi posti nelle stesse posizioni all'interno dei registri estesi generando una maschera
- Se il confronto è stato soddisfatto la maschera corrisponde a una serie di 1 per quell'elemento mentre una serie di 0 in caso contrario
- Le maschere ottenute possono essere utilizzate per filtrare e/o elaborare i dati in funzione dell'esito del confronto

## Istruzioni di confronto 2/3

- Sono previste tre tipologie di confronti: uguale (EQ), maggiore (GT) o minore (LT)
  - EQ: per valori interi a 8, 16, 32, 64 bit
  - GT: per valori interi signed a 8, 16, 32, 64 bit
  - LT: per valori interi signed a 8, 16, 32 bit

**`__m128i _mm_cmpeq_epi8 (__m128i a, __m128i b)`**

**`__m128i _mm_cmpgt_epi8 (__m128i a, __m128i b)`**

**`__m128i _mm_cmplt_epi8 (__m128i a, __m128i b)`**

## Istruzioni di confronto 3/3

Esempio: `dest = _mm_cmpgt_epi16(A,B);`

Cluster	Number of Genes
A	42
B	-765
C	68
D	870
E	0
F	1556
G	-7234
H	3030

<b>B</b>	87	7	87	95	0	-15	127	3
----------	----	---	----	----	---	-----	-----	---

<b>dest</b>	0000	0000	0000	65535	0000	65535	0000	65535
-------------	------	------	------	-------	------	-------	------	-------

I due valori 0000h e FFFFh ( $65535_{10}$ ) generati dalle istruzioni di confronto si prestano ad essere utilizzati, ma non necessariamente, con istruzioni di tipo logico



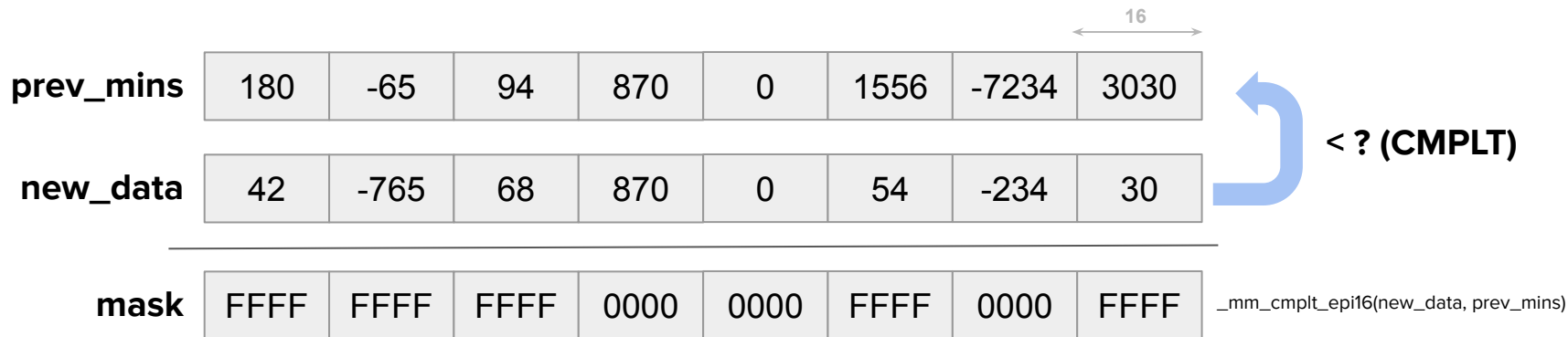
# Istruzioni di confronto senza branch/salti?

- Come appena visto, esistono istruzioni per eseguire confronti ma non istruzioni di salto condizionato (*branch*)
- Questo era prevedibile essendo possibile un solo flusso di esecuzione delle istruzioni
- Per eseguire operazioni diverse sui dati all'interno di un unico flusso di esecuzione si utilizzano delle *maschere* (eg, l'output di una istruzione di confronto)
- Mediante le maschere è possibile filtrare dati che soddisfano (o non soddisfano) determinate condizioni

**Esercizio:** si consideri l'esempio della slide precedente, come è possibile inserire in un registro solo i valori minimi per ogni linea tra i due registri estesi **A** (chiamato in seguito **prev\_mins**) e **B** (chiamato in seguito **new\_data**) contenenti ciascuno 8 valori interi signed a 16 bit?

# Utilizzo di maschere per assegnamenti condizionati 1/4

- Con riferimento all'esercizio proposto nella slide precedente, una strategia per selezionare i valori minimi tra due vettori (**prev\_mins** e **new\_data**) potrebbe essere la seguente, partendo da una istruzione di confronto (`__m128i _mm_cmplt_epi16`)



- La maschera indica con 0000 le linee che non soddisfano la condizione CMPLT mentre con FFFF il caso contrario (attenzione al mix, voluto, tra valori decimali ed esadecimali)

## Utilizzo di maschere per assegnamenti condizionati 2/4

- Una volta ottenuta la maschera è possibile azzerare in **new\_data** i valori che non concorrono all'aggiornamento dei valori contenuti in **prev\_mins** e mantenere inalterati gli altri valori; per questa operazione è sufficiente un AND tra **mask** e **new\_data**

									16	
									←	→
<b>mask</b>	FFFF	FFFF	FFFF	0000	0000	FFFF	0000	FFFF		
									<b>AND</b>	
<b>new_data</b>	42	-765	68	870	0	54	-234	30		
<b>temp_1</b>	42	-765	68	0000	0000	54	0000	30		_mm_and_si128

- I valori che non sono 0000 in **temp\_1** saranno quelli che, successivamente, sostituiranno i corrispondenti valori per quella linea nel registro **prev\_mins**

## Utilizzo di maschere per assegnamenti condizionati 3/4

- Analogamente è possibile azzerare in **prev\_mins** i valori che non risultano più minimi e mantenere inalterati gli altri valori; per questa operazione è sufficiente un ANDNOT tra **mask** e **prev\_mins**; si ricorda che ANDNOT prima nega il primo operando (**mask**) e poi esegue AND del risultato con **prev\_mins**

	<div>← 16 →</div>								
<b>not(mask)</b>	0000	0000	0000	FFFF	FFFF	0000	FFFF	0000	<b>ANDNOT</b>
<b>prev_mins</b>	180	-65	94	870	0	1556	-7234	3030	
<hr/>									
<b>temp_2</b>	0000	0000	0000	870	0	0000	-7234	0000	<small>_mm_pandnot_si128</small>

- I valori non nulli in **temp\_2** sono quelli che, successivamente, non saranno sostituiti dai corrispondenti valori per quella linea nel registro **prev\_mins**

## Utilizzo di maschere per assegnamenti condizionati 4/4

- L'ultimo passo consiste nell'aggiornare il valore di **prev\_mins** *fondendo* i valori *sopravvissuti* in **temp\_1** e **temp\_2** con un semplice OR per ottenere, in **prev\_mins**, i valori minimi per ogni linea tra i due registri (**prev\_mins** e **new\_data**)

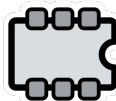
								16
								←
temp_1	42	-765	68	0000	0000	54	0000	30
temp_2	0000	0000	0000	870	0	0000	-7234	0000
OR								
prev_mins	42	-765	68	870	0	54	-7234	30
_mm_por_si128								

- Sarebbe stato possibile usare altre istruzioni (eg, istruzioni di *blending*)?  
In questo caso no, non è disponibile l'istruzione *blendv* per interi a 16 bit

# Utilizzo di maschere per assegnamenti condizionati in memoria

- Solo nel caso di **byte** è possibile fare scritture in memoria condizionate da una maschera
- Solo i byte in **a** per i quali il valore più significativo della corrispondente maschera è 1 sono scritti in memoria all'indirizzo **mem\_addr**; gli altri byte in memoria rimangono invariati (simbolo “-” nella figura)
- La latenza è 6 clock e l'indirizzo non deve essere necessariamente allineato

```
void _mm_maskmoveu_si128 (__m128i a, __m128i mask, char* mem_addr)
```



## Ricerca di minimi e massimi 1/2

- Le operazioni dell'esempio precedente sono molto comuni
- Per questa ragione, esistono istruzioni specifiche per l'identificazione di MIN e MAX
- Sono applicabili a registri estesi contenenti tipi di dato interi – signed (epi) e unsigned (epu) – a 8,16, e 32 bit

**\_\_m128i \_mm\_max\_epi8 (\_\_m128i a, \_\_m128i b)**

**\_\_m128i \_mm\_max\_epi16 (\_\_m128i a, \_\_m128i b)**

**\_\_m128i \_mm\_max\_epi32 (\_\_m128i a, \_\_m128i b)**

**\_\_m128i \_mm\_max\_epu8 (\_\_m128i a, \_\_m128i b)**

**\_\_m128i \_mm\_max\_epu16 (\_\_m128i a, \_\_m128i b)**

**\_\_m128i \_mm\_max\_epu32 (\_\_m128i a, \_\_m128i b)**

## Ricerca di minimi e massimi 2/2

- Esclusivamente per interi unsigned a 16 bit, esiste la possibilità di identificare il MIN e la sua posizione all'interno di un registro mediante una singola istruzione (minpos)
- Il registro destinazione contiene il valore minimo, bit [15..0], e la sua posizione, bit [18..16]

\_\_m128i \_mm\_minpos\_epu16 (\_\_m128i a)

Esempio: `dest = _mm_minpos_epu16(a)`

Diagram illustrating the initial state of two registers:

- Register a:** Contains the value 87 7 87 95 2 15 127 3. A 16-bit segment (the last two bytes, 127 3) is highlighted with a double-headed arrow labeled 16.
- Register dest:** Contains the value 0000 0000 0000 0000 0000 0000 0003 2.



# Esercizi

**Esercizio 1:** Scrivere il codice per individuare il valore minimo all'interno di un vettore di byte signed utilizzando:

- i) **codice scalare**
- ii) **maschere e operazioni logiche**
- iii) **istruzioni di blending (se possibile)**
- iv) **istruzioni per ricerca minimi (se possibile)**

Confrontare le prestazioni mediante *performance counters*

**Esercizio 2:** Individuare il valore minimo e relativo indice all'interno di un vettore di byte signed

*Suggerimento: utilizzare un registro per gli indici (o iterazione loop e indici fissi)*

# Tipi di dati signed e unsigned

Si consideri il tipo di dato il byte:

- Nel caso unsigned, il range è  $[255, 0]$
- Nel caso signed, il range è  $[+127, -128]$

Esempio, il byte 10001111:

- equivale a 143 come unsigned
- equivale a -113 come signed

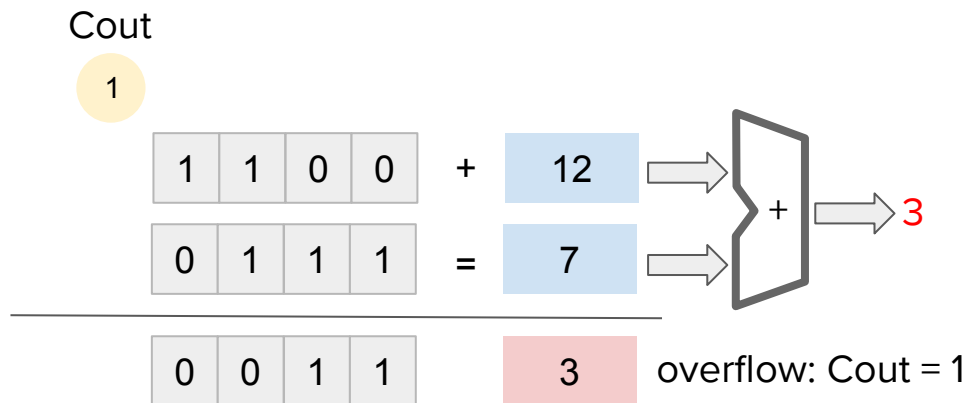
Indipendentemente dalla rappresentazione, le operazioni aritmetiche (eg, somma e sottrazione) sono eseguite attraverso le medesime reti (ALU, sommatori nell'esempio). La differenza risiede nella gestione dei segnali di FLAG; tuttavia, la maggior parte delle istruzioni SIMD x86 non modificano i FLAG della CPU

# Elaborazioni con saturazione

- Per la modalità stessa con la quale sono elaborati simultaneamente multipli dati con il paradigma SIMD, sarebbe controproducente gestire situazioni di overflow
- Inoltre, tali istruzioni sono frequentemente utilizzate per elaborare segnali audio e video per i quali spesso si preferisce *saturare* ai valori limite
- La saturazione consiste nell'eliminare situazioni di overflow impostando come risultato di un'operazione il valore massimo o minimo possibile con la rappresentazione dei dati utilizzata
- Esempio: se  $A=-100$  e  $B=-50$  sono due byte signed, il risultato di  $A + B$  non sarebbe rappresentabile con 8 bit generando overflow
- Applicando la saturazione il risultato è impostato a  $-128$  (valore massimo rappresentabile con un byte signed)

# Somme tra dati unsigned 1/4

Si consideri la somma tra i numeri a 4 bit di tipo unsigned 12 e 7

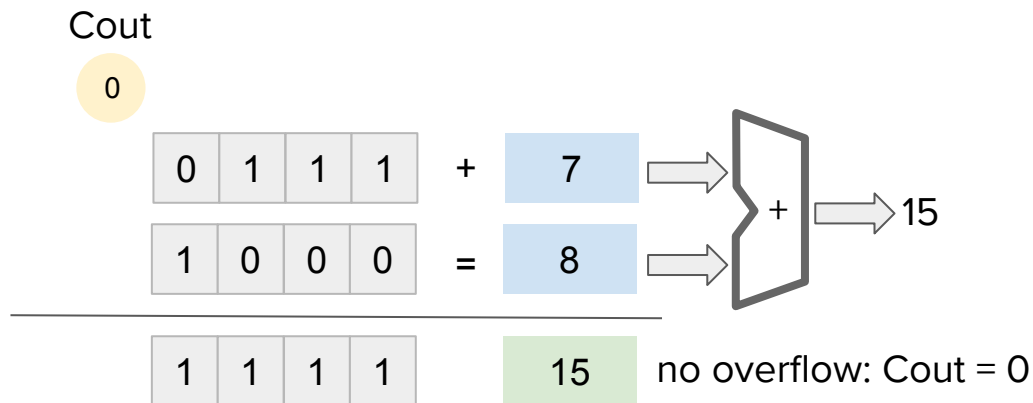


Applicando la saturazione il risultato dell'operazione sarebbe 15



## Somme tra dati unsigned 2/4

Si consideri la somma tra i numeri a 4 bit di tipo unsigned 7 e 8

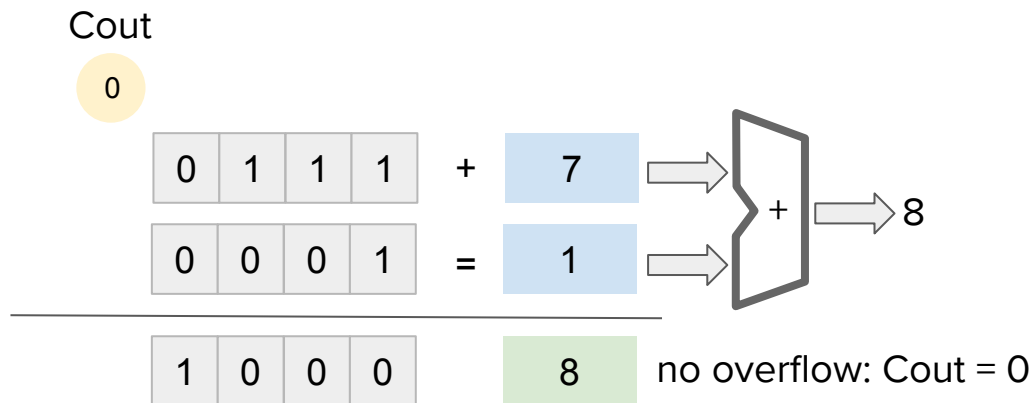


Applicando la saturazione il risultato dell'operazione sarebbe identico (15)



## Somme tra dati unsigned 3/4

Si consideri la somma tra i numeri a 4 bit di tipo unsigned 7 e 1

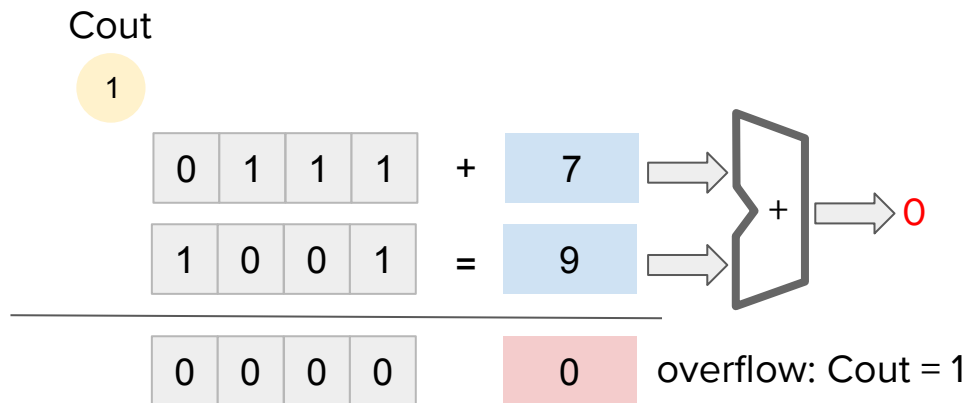


Applicando la saturazione il risultato dell'operazione sarebbe identico (8)



## Somme tra dati unsigned 4/4

Si consideri la somma tra i numeri a 4 bit di tipo unsigned 7 e 9



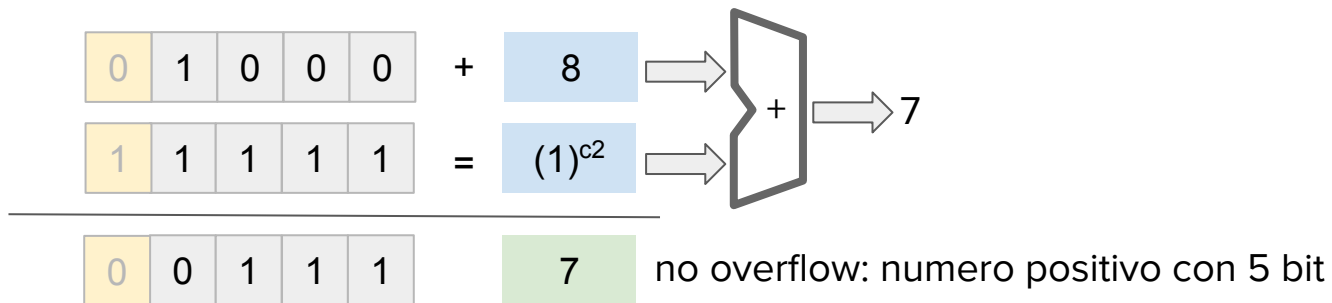
Applicando la saturazione il risultato dell'operazione sarebbe 15



# Sottrazioni tra dati unsigned 1/4

Si consideri la sottrazione tra i numeri a 4 bit di tipo unsigned 8 e 1

$$8 - 1 = 8 + (1)^{c2}$$



Applicando la saturazione il risultato dell'operazione sarebbe identico (7)

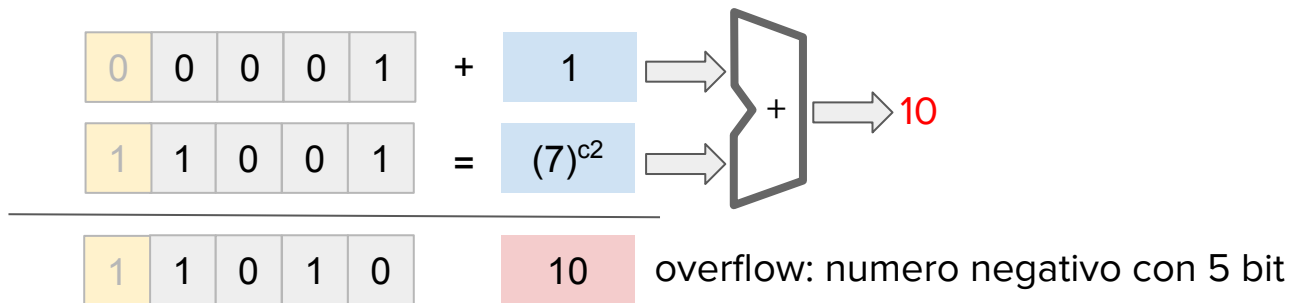




## Sottrazioni tra dati unsigned 2/4

Si consideri la sottrazione tra i numeri a 4 bit di tipo unsigned 1 e 7

$$1 - (7) = 1 + (+7)^{c2}$$



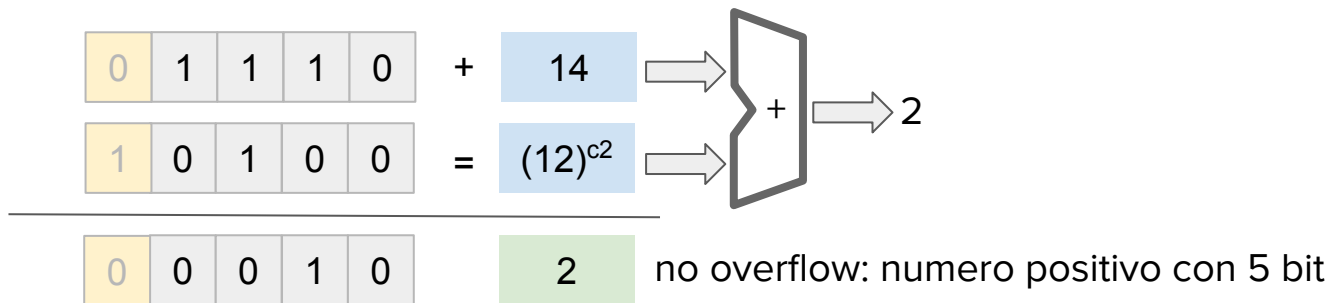
Applicando la saturazione il risultato dell'operazione sarebbe 0



## Sottrazioni tra dati unsigned 3/4

Si consideri la sottrazione tra i numeri a 4 bit di tipo unsigned 14 e 12

$$14 - (12) = 14 + (12)^{c2}$$



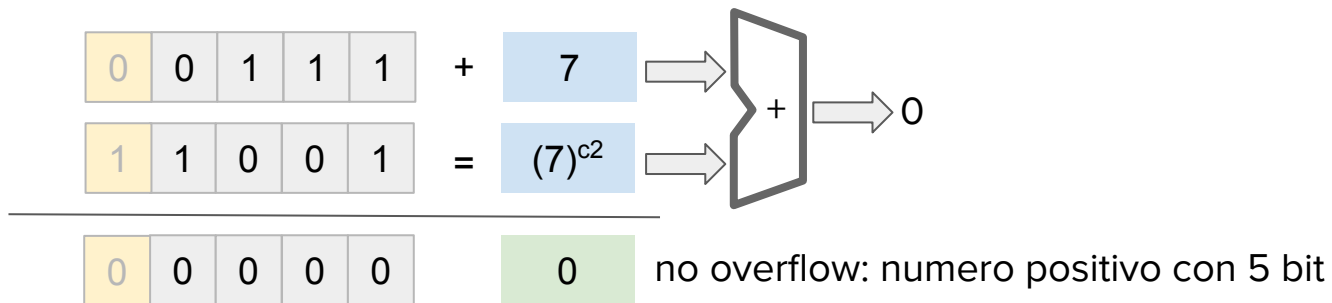
Applicando la saturazione il risultato dell'operazione sarebbe identico (2)



## Sottrazioni tra dati unsigned 4/4

Si consideri la sottrazione tra i numeri a 4 bit di tipo unsigned 7 e 7

$$7 - (7) = 7 + (7)^{c2}$$

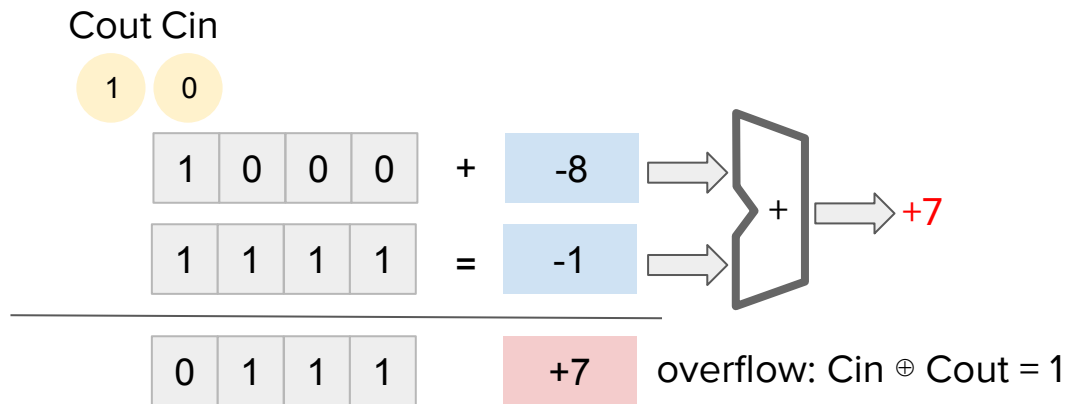


Applicando la saturazione il risultato dell'operazione sarebbe identico (0)

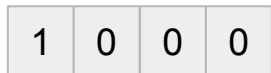


# Somme tra dati signed 1/4

Si consideri la somma tra i numeri a 4 bit di tipo signed -8 e -1

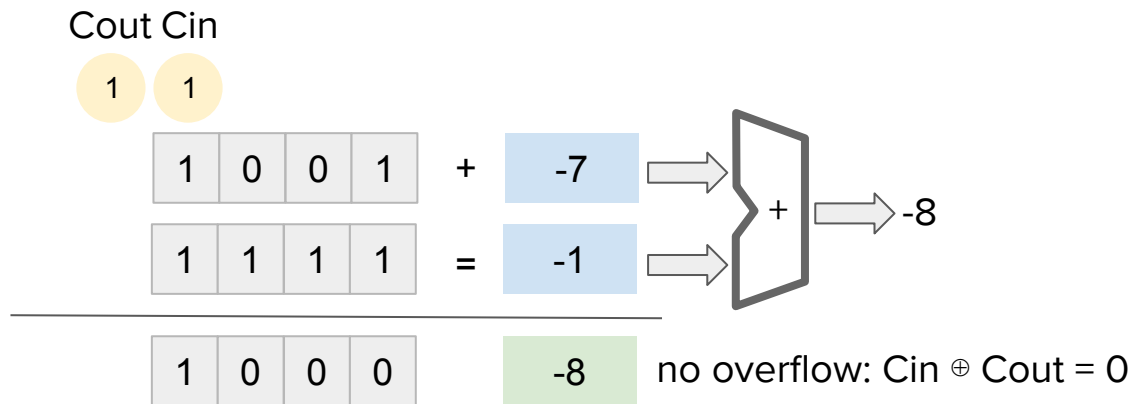


Applicando la saturazione il risultato dell'operazione sarebbe -8

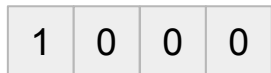


## Somme tra dati signed 2/4

Si consideri la somma tra i numeri a 4 bit di tipo signed -7 e -1

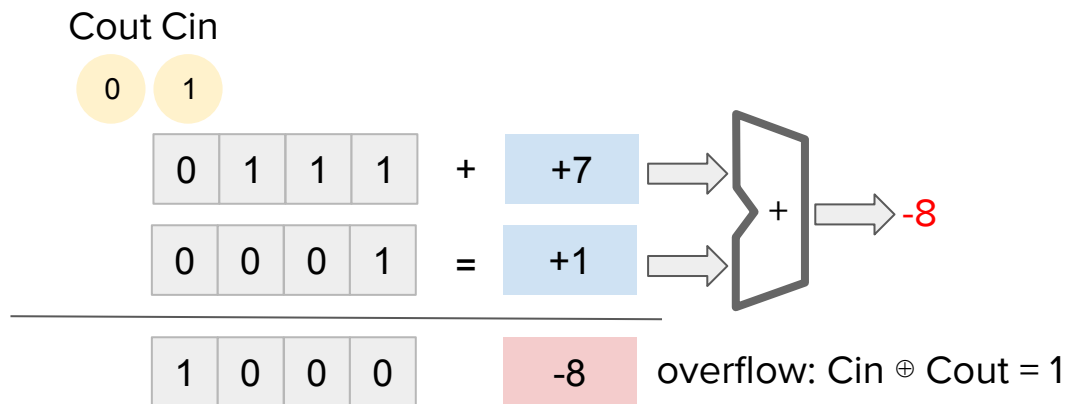


Applicando la saturazione il risultato dell'operazione sarebbe identico (-8)



## Somme tra dati signed 3/4

Si consideri la somma tra i numeri a 4 bit di tipo signed +7 e +1

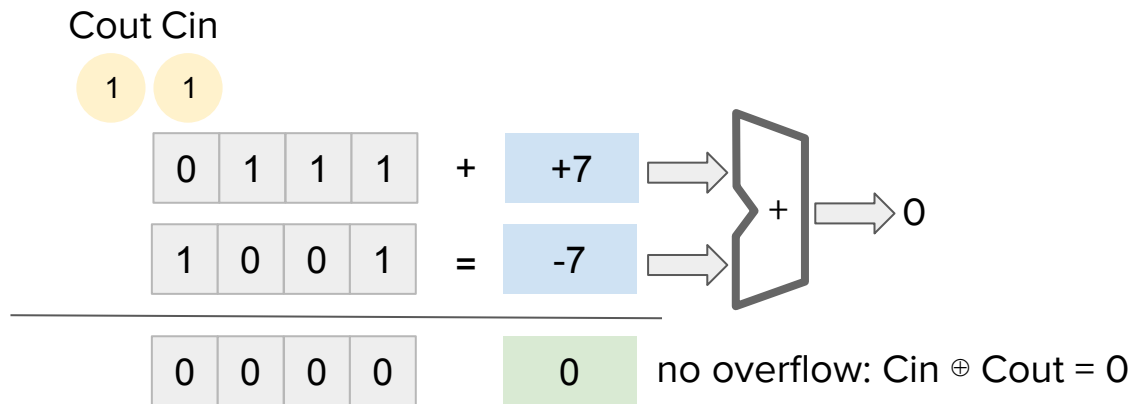


Applicando la saturazione il risultato dell'operazione sarebbe +7



## Somme tra dati signed 4/4

Si consideri la somma tra i numeri a 4 bit di tipo signed +7 e -7



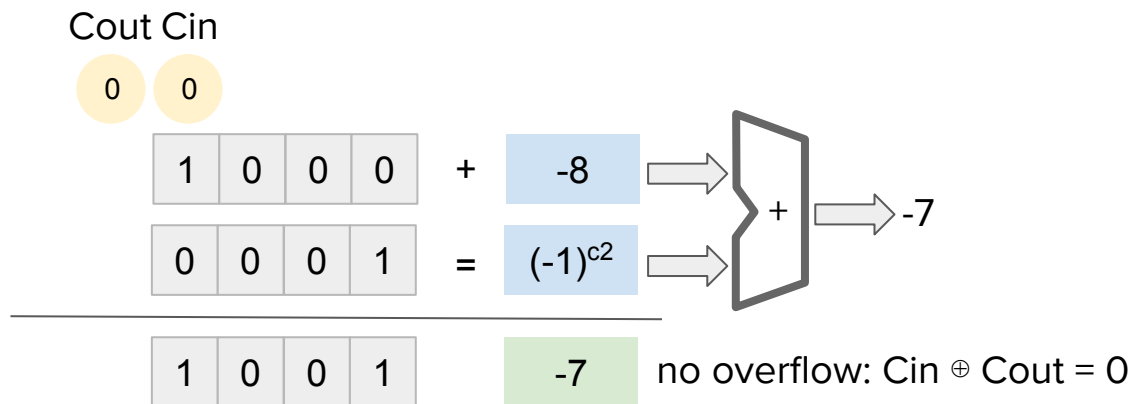
Applicando la saturazione il risultato dell'operazione sarebbe identico (0)



# Sottrazioni tra dati signed 1/4

Si consideri la sottrazione tra i numeri a 4 bit di tipo signed -8 e -1

$$8 - (-1) = -8 + (-1)^{C2}$$



Applicando la saturazione il risultato dell'operazione sarebbe identico (-7)

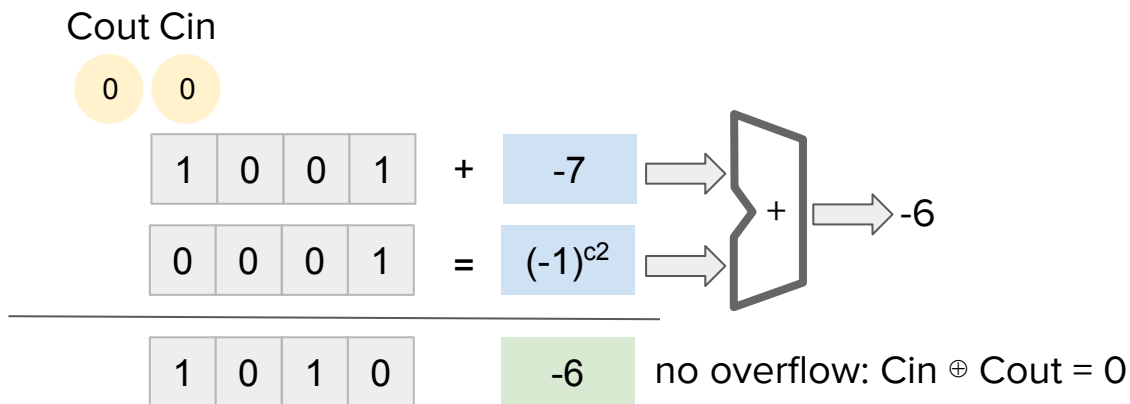
1	0	0	1
---	---	---	---



## Sottrazioni tra dati signed 2/4

Si consideri la sottrazione tra i numeri a 4 bit di tipo signed -7 e -1

$$-7 - (-1) = -7 + (-1)^{C2}$$



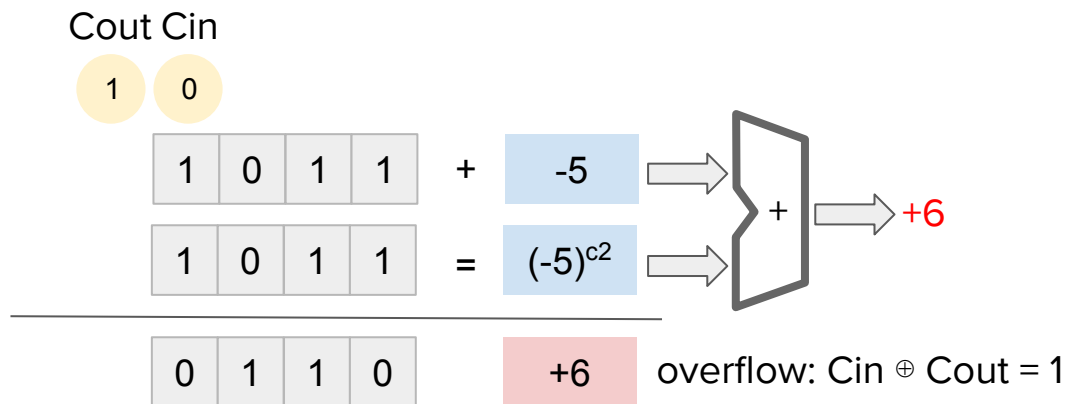
Applicando la saturazione il risultato dell'operazione sarebbe identico (-6)

1	0	1	0
---	---	---	---

## Sottrazioni tra dati signed 3/4

Si consideri la sottrazione tra i numeri a 4 bit di tipo signed -5 e +5

$$-5 - (+5) = -5 + (+5)^{C2}$$



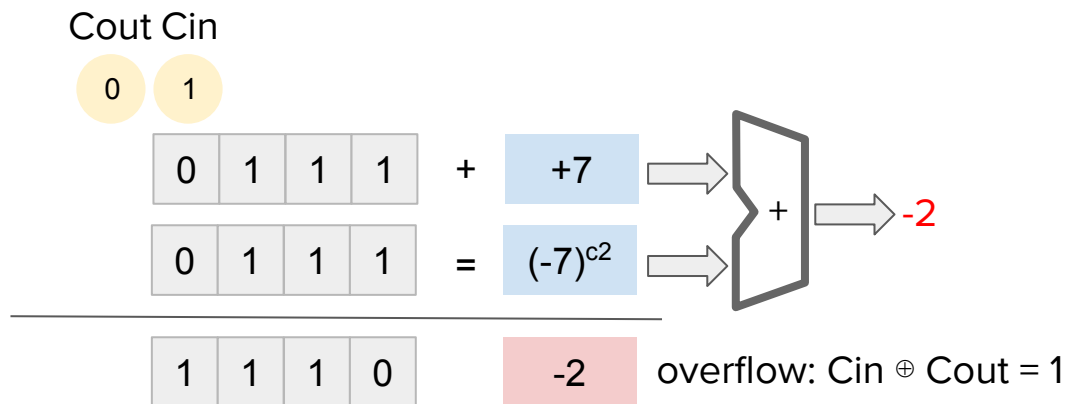
Applicando la saturazione il risultato dell'operazione sarebbe -8



## Sottrazioni tra dati signed 4/4

Consideriamo la sottrazione tra i numeri a 4 bit di tipo signed +7 e -7

$$+7 - (-7) = +7 + (-7)^{C2}$$



Applicando la saturazione il risultato dell'operazione sarebbe +7



# Istruzioni aritmetiche con byte: addizione

Nell'ISA SSE sono presenti le seguenti istruzioni aritmetiche per dati di tipo BYTE:

**`__m128i _mm_add_epi8 (__m128i a, __m128i b)`**

Somma ciascuna coppia  $a[i]$  e  $b[i]$  di elementi presenti nei registri **a** e **b**

$\text{dest}[i] = a[i] + b[i] \quad i \in [0, 15]$

Gli operandi possono essere BYTE signed o unsigned

**`__m128i _mm_adds_epi8 (__m128i a, __m128i b)`**

Somma con saturazione ciascuna coppia  $a[i]$  e  $b[i]$  di elementi presenti in **a** e **b**

$\text{dest}[i] = \text{saturation}(a[i] + b[i]) \quad i \in [0, 15]$

Gli operandi possono essere solo BYTE signed

# Istruzioni aritmetiche con byte: addizione tra dati unsigned

`__m128i _mm_add_epi8 (__m128i a, __m128i b)`

**a**

200	64	250	4	45	128	77	6	87	7	68	195	0	255	128	3	+
-----	----	-----	---	----	-----	----	---	----	---	----	-----	---	-----	-----	---	---

**b**

100	68	210	34	145	120	68	26	87	71	32	106	30	53	128	103	=
-----	----	-----	----	-----	-----	----	----	----	----	----	-----	----	----	-----	-----	---

**a + b**

44	132	204	38	190	248	145	32	174	78	100	45	30	52	0	106
----	-----	-----	----	-----	-----	-----	----	-----	----	-----	----	----	----	---	-----

I casi di overflow sono evidenziati in rosso

# Istruzioni aritmetiche con byte: addizione tra dati signed

`__m128i _mm_add_epi8 (__m128i a, __m128i b)`

**a**

-20	64	50	4	-45	127	-77	6	-87	7	68	-95	0	15	-127	3
-----	----	----	---	-----	-----	-----	---	-----	---	----	-----	---	----	------	---

 +

**b**

100	68	10	34	-127	-128	68	26	87	71	32	106	-30	53	-127	-103
-----	----	----	----	------	------	----	----	----	----	----	-----	-----	----	------	------

 =

**a + b**

80	-124	60	38	84	-1	-9	32	0	78	100	11	-30	68	2	-100
----	------	----	----	----	----	----	----	---	----	-----	----	-----	----	---	------

I casi di overflow sono evidenziati in rosso

# Istruzioni aritmetiche con byte: addizione con saturazione

`__m128i _mm_adds_epi8 (__m128i a, __m128i b)`

**a**

-20	64	50	4	-45	127	-77	6	-87	7	68	-95	0	15	-127	3	+
-----	----	----	---	-----	-----	-----	---	-----	---	----	-----	---	----	------	---	---

**b**

100	68	10	34	-127	-128	68	26	87	71	32	106	-30	53	-127	-103	=
-----	----	----	----	------	------	----	----	----	----	----	-----	-----	----	------	------	---

**Sat(a + b)**

80	127	60	38	-128	-1	-9	32	0	78	100	11	-30	68	-128	-100
----	-----	----	----	------	----	----	----	---	----	-----	----	-----	----	------	------

Non si verificano mai situazioni di overflow (si satura, evidenziato in blu)

Le addizioni con saturazione sono supportate solo con dati BYTE (epi8) e INT16 (ep16) di tipo signed

# Istruzioni aritmetiche con byte: sottrazione tra dati unsigned

`__m128i _mm_sub_epi8 (__m128i a, __m128i b)`

**a**

-200	64	250	4	45	128	77	6	87	7	68	195	0	255	128	3	-
------	----	-----	---	----	-----	----	---	----	---	----	-----	---	-----	-----	---	---

**b**

100	68	210	34	145	120	68	26	87	71	32	106	30	53	128	103	=
-----	----	-----	----	-----	-----	----	----	----	----	----	-----	----	----	-----	-----	---

**a - b**

100	252	40	-226	156	8	9	236	0	192	36	89	226	202	0	156
-----	-----	----	------	-----	---	---	-----	---	-----	----	----	-----	-----	---	-----

I casi di overflow sono evidenziati in rosso



# Istruzioni aritmetiche con byte: sottrazione tra dati signed

`__m128i _mm_sub_epi8 (__m128i a, __m128i b)`

**a**

-20	64	50	4	-45	127	-77	6	-87	7	68	-95	0	15	-127	3	-
-----	----	----	---	-----	-----	-----	---	-----	---	----	-----	---	----	------	---	---

**b**

100	68	10	34	-127	-128	68	26	87	71	32	106	-30	53	-127	-103	=
-----	----	----	----	------	------	----	----	----	----	----	-----	-----	----	------	------	---

**a - b**

-120	-4	40	-30	82	-1	111	-20	82	-64	36	55	30	-38	0	106
------	----	----	-----	----	----	-----	-----	----	-----	----	----	----	-----	---	-----

I casi di overflow sono evidenziati in rosso

# Istruzioni aritmetiche con byte: sottrazione con saturazione

**\_\_m128i \_mm\_subs\_epi8 (\_\_m128i a, \_\_m128i b)**

**a**

-20	64	50	4	-45	127	-77	6	-87	7	68	-95	0	15	-127	3
-----	----	----	---	-----	-----	-----	---	-----	---	----	-----	---	----	------	---

-

**b**

100	68	10	34	-127	-128	68	26	87	71	32	106	-30	53	-127	-103
-----	----	----	----	------	------	----	----	----	----	----	-----	-----	----	------	------

=

**Sat(a - b)**

-120	-4	40	-30	82	127	-128	-20	-128	-64	36	-128	30	-38	0	106
------	----	----	-----	----	-----	------	-----	------	-----	----	------	----	-----	---	-----

Non si verificano mai situazioni di overflow

Le addizioni con saturazione sono supportate solo con dati BYTE (epi8) e INT16 (ep16) di tipo signed

# Absolute value (ABS)

- Calcolano il valore assoluto degli elementi di un registro contenente valori signed
- Disponibile per interi a 8, 16 e 32 bit

**\_\_m128i \_mm\_abs\_epi8 (\_\_m128i a)**

**\_\_m128i \_mm\_abs\_epi16 (\_\_m128i a)**

**\_\_m128i \_mm\_abs\_epi32 (\_\_m128i a)**

325678	-254	-563434	3643
--------	------	---------	------

---

325678	254	563434	3643
--------	-----	--------	------

## Add e Sub *orizzontali* senza saturazione

- Oltre alle canoniche operazioni di somma e sottrazione *verticale*, è possibile eseguire anche le stesse *orizzontalmente* tra interi signed a 16 e 32 bit contenuti in due registri

`__m128i _mm_hadd_epi16 (__m128i a, __m128i b)`

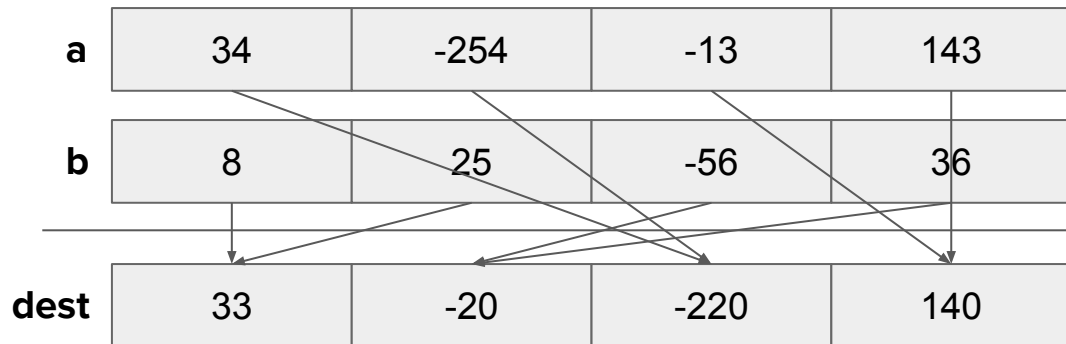
`__m128i _mm_hadd_epi32 (__m128i a, __m128i b)`

`__m128i _mm_hsub_epi16 (__m128i a, __m128i b)`

`__m128i _mm_hsub_epi32 (__m128i a, __m128i b)`

Esempio:

`dest = _mm_hadd_epi32 (a, b)`



## Add e Sub *orizzontali* con saturazione

- Le operazioni di somma e sottrazione orizzontale con saturazione esistono solo per interi `signed` a `16` bit

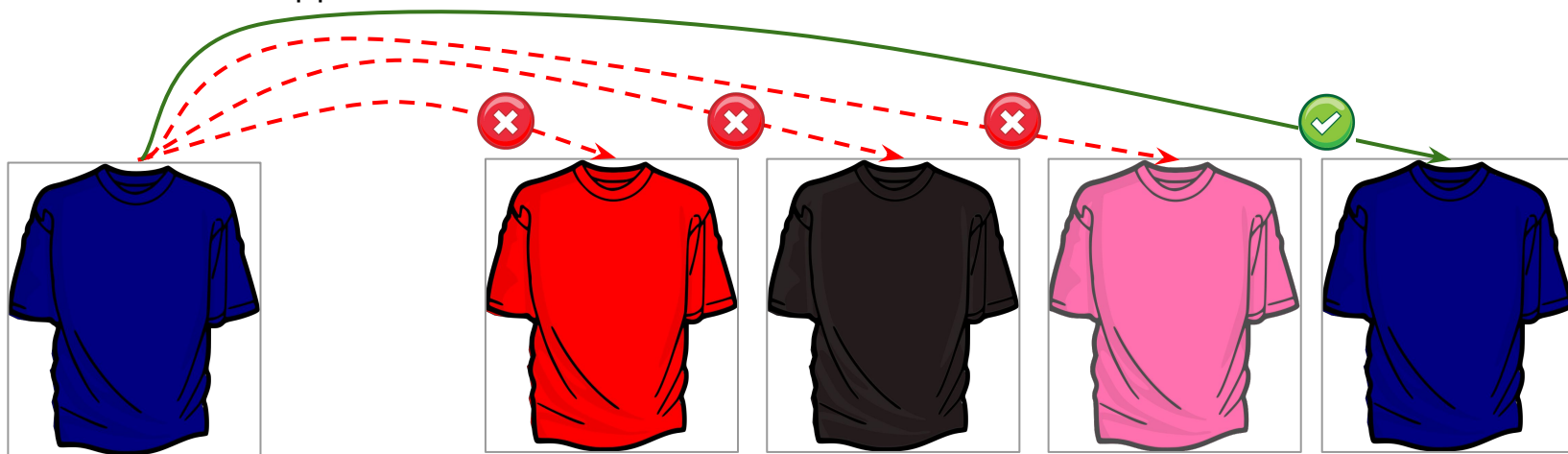
```
__m128i _mm_hadds_epi16 (__m128i a, __m128i b)
```

```
__m128i _mm_hsubs_epi16 (__m128i a, __m128i b)
```

- Le istruzioni di somma e sottrazione verticale possono essere utilizzate anche con registri a 64 bit (`__m64`)

## Ulteriori istruzioni aritmetiche per SAD

- Esistono nel set di istruzioni SSE altre istruzioni aritmetiche con interi
- Tra queste, ci sono istruzioni che consentono di eseguire SAD (Sum of Absolute Differences), un'operazione spesso utilizzata nell'ambito della computer vision
- SAD può essere utile, per esempio, per confrontare se due immagini o patch sono *simili* oppure no



# Somma degli elementi di un array di byte unsigned

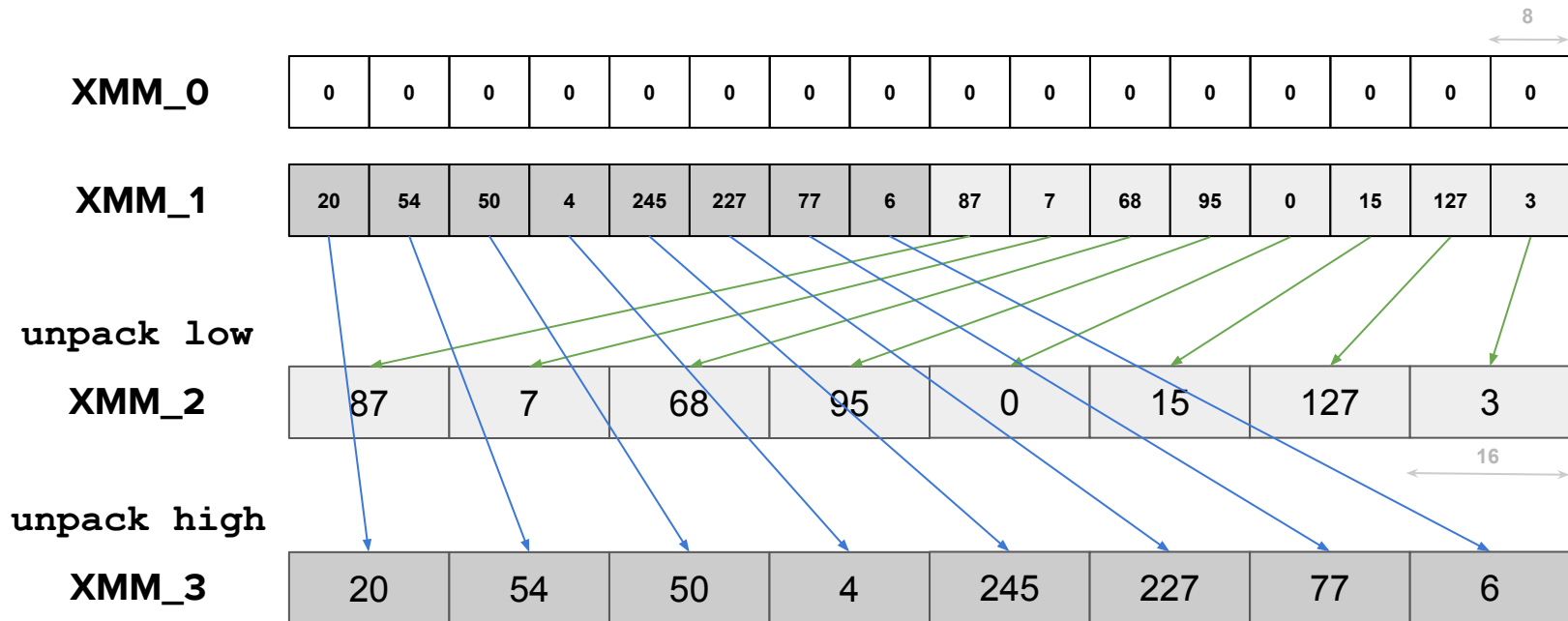
Problema: calcolare la somma di tutti gli elementi di un array di byte unsigned di dimensione pari a `length`

```
unsigned char A[length];  
unsigned int sum=0;  
  
for (i=0; i<length; i++)  
    sum = sum + A[i];
```

- E' fondamentale prestare attenzione a possibili condizioni di overflow
- Inoltre, dovendo addizionare valori a 8 bit, per prima cosa è necessario estendere la dimensione del risultato (la somma di due valori a 8 bit è codificata con 9 bit)
- Per l'estensione si possono utilizzare le istruzioni `unpack` (pagina successiva)

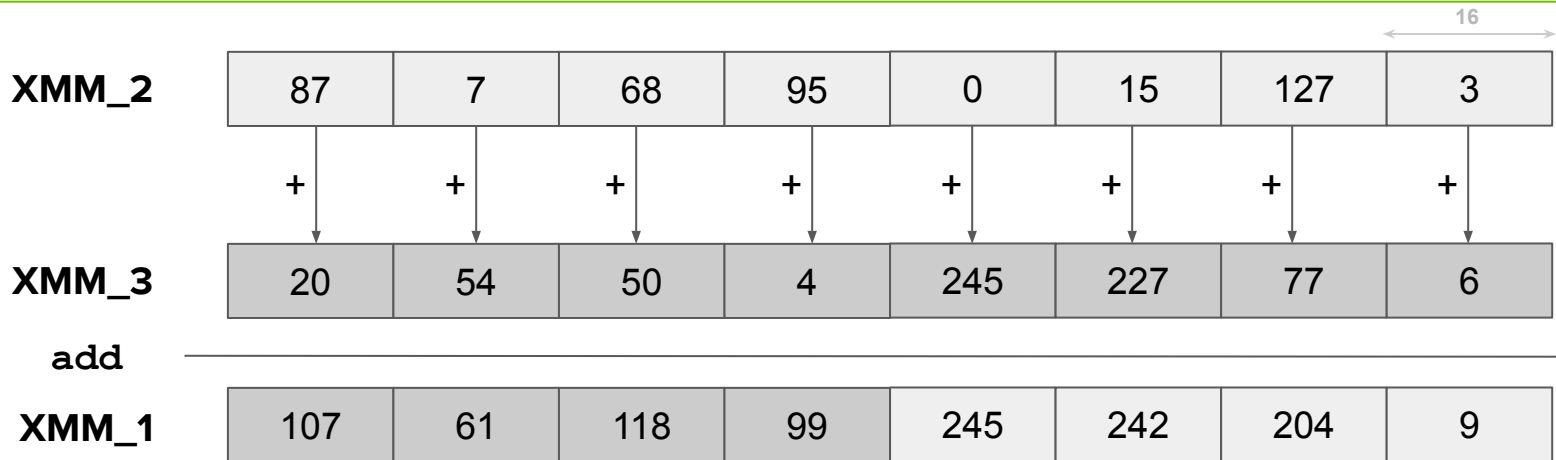
# Somma degli elementi di un array di byte unsigned: 8 → 16 bit

- Non essendo sufficienti 8 bit per la somma di due byte, è necessario eseguire le addizioni con dati (almeno) convertiti a 16 bit
- Le istruzioni `unpack` consentono questo utilizzando un registro contenente tutti zero



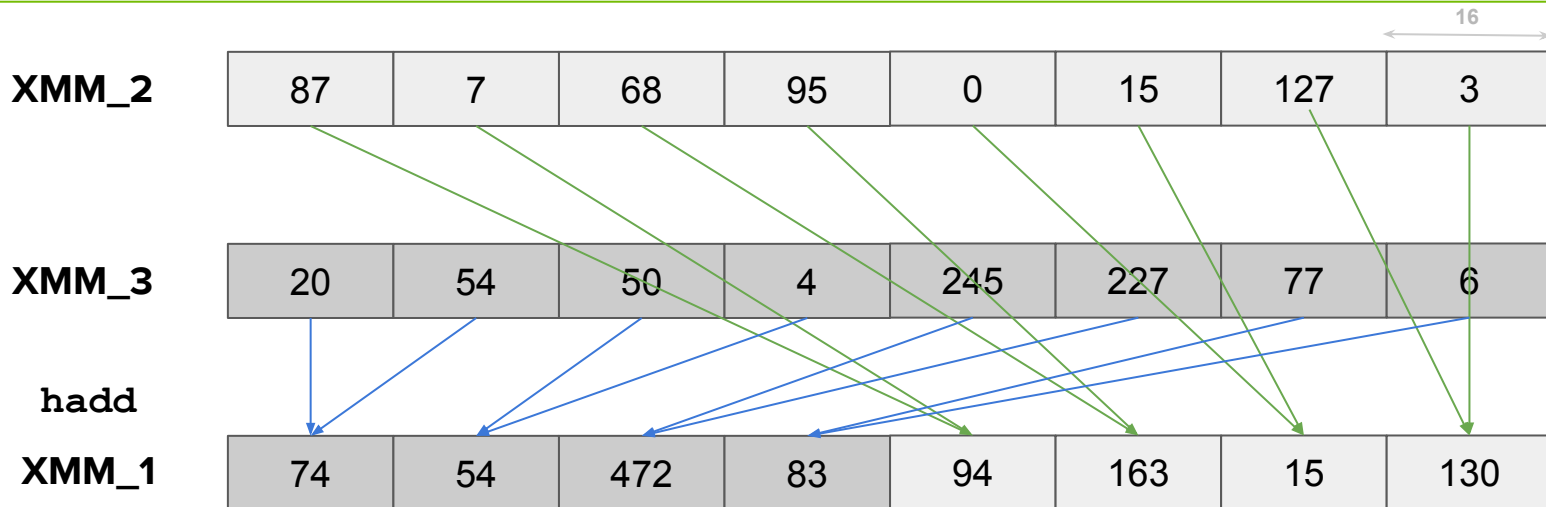


## Somma degli elementi di un array di byte unsigned (con add)



- Una volta estesi i byte a 16 bit, con le istruzioni `unpack` è possibile eseguire come passo iniziale l'addizione tra coppie di elementi a 16 bit *verticali* mediante l'istruzione `add` (in figura) prima di aggiornare un registro usato come accumulatore delle somme parziali
- In alternativa, è anche possibile eseguire l'addizione orizzontale tra elementi a 16 bit utilizzando l'istruzione `hadd` (come mostrato nella pagina successiva)

## Somma degli elementi di un array di byte unsigned (con hadd)



- Il risultato dell'addizione (add o hadd) tra due elementi è un valore a 9 bit ( $\leq 255 \times 2$ ), contenuto nei 16 bit (x8) di XMM\_1, che può aggiornare un accumulatore (eg, a 16 bit)
- Tuttavia, con accumulo a 16 bit si può verificare overflow se `length > 2048 byte`
- Infatti, con dati 16 bit unsigned, si verifica overflow con più di 128 addizioni (i.e., con 129 elementi la somma può raggiungere il valore  $129 \times (255 + 255) = 65790 > 65535$ )

# Somma degli elementi di un array di byte unsigned e overflow

- Se il numero di elementi dell'array è minore o uguale a 2048 è possibile ignorare il problema e procedere in sicurezza con l'accumulo di multipli (x8) valori a 16 bit
- In caso contrario, esistono almeno due soluzioni:
  - si eseguono addizioni in gruppi (max 2048 element)i e, al termine di ciascun gruppo di operazioni, si utilizza un accumulatore di maggiori dimensioni (eg, a 32 bit)
  - si sommano tutti gli elementi con un unico ciclo aumentando la dimensione dei dati (eg, passando a 32 bit, con gli stessi limiti appena evidenziati) per evitare problemi di overflow dopo 128 addizioni (ie. dopo aver sommato più di 128x16 elementi)

La prima soluzione ha un costo maggiore nella gestione dei loop multipli (eg, modulo 2048 elementi) ma, rispetto alla seconda, esegue complessivamente meno istruzioni di unpacking e addizioni di dati di maggiori dimensioni che riducono il grado di parallelismo ottenibile

# Somma degli elementi di un array di byte unsigned: hadd vs add

- Dall'analisi delle specifiche delle istruzioni, per la specifica finalità dell'esempio, risulta vantaggioso l'utilizzo della `add` (minore latenza e CPI) vs `hadd`
- Inoltre, sebbene irrilevante sommando due byte, il risultato della `hadd` è signed

```
__m128i _mm_hadd_epi16 (__m128i a, __m128i b)
```

## Synopsis

```
__m128i _mm_hadd_epi16 (__m128i a, __m128i b)
#include <tmmintrin.h>
Instruction: phaddw xmm, xmm
CPUID Flags: SSSE3
```

## Description

Horizontally add adjacent pairs of 16-bit integers in `a` and `b`, and pack the signed 16-bit results in `dst`.

## Operation

```
dst[15:0] := a[31:16] + a[15:0]
dst[31:16] := a[63:48] + a[47:32]
dst[47:32] := a[95:80] + a[79:64]
dst[63:48] := a[127:112] + a[111:96]
dst[79:64] := b[31:16] + b[15:0]
dst[95:80] := b[63:48] + b[47:32]
dst[111:96] := b[95:80] + b[79:64]
dst[127:112] := b[127:112] + b[111:96]
```

## Latency and Throughput

Architecture	Latency	Throughput (CPI)
Alderlake	2	1
Icelake Xeon	-	1.07
Sapphire Rapids	2	1
Skylake	3	2

```
__m128i _mm_add_epi16 (__m128i a, __m128i b)
```

## Synopsis

```
__m128i _mm_add_epi16 (__m128i a, __m128i b)
#include <emmintrin.h>
Instruction: paddw xmm, xmm
CPUID Flags: SSE2
```

## Description

Add packed 16-bit integers in `a` and `b`, and store the results in `dst`.

## Operation

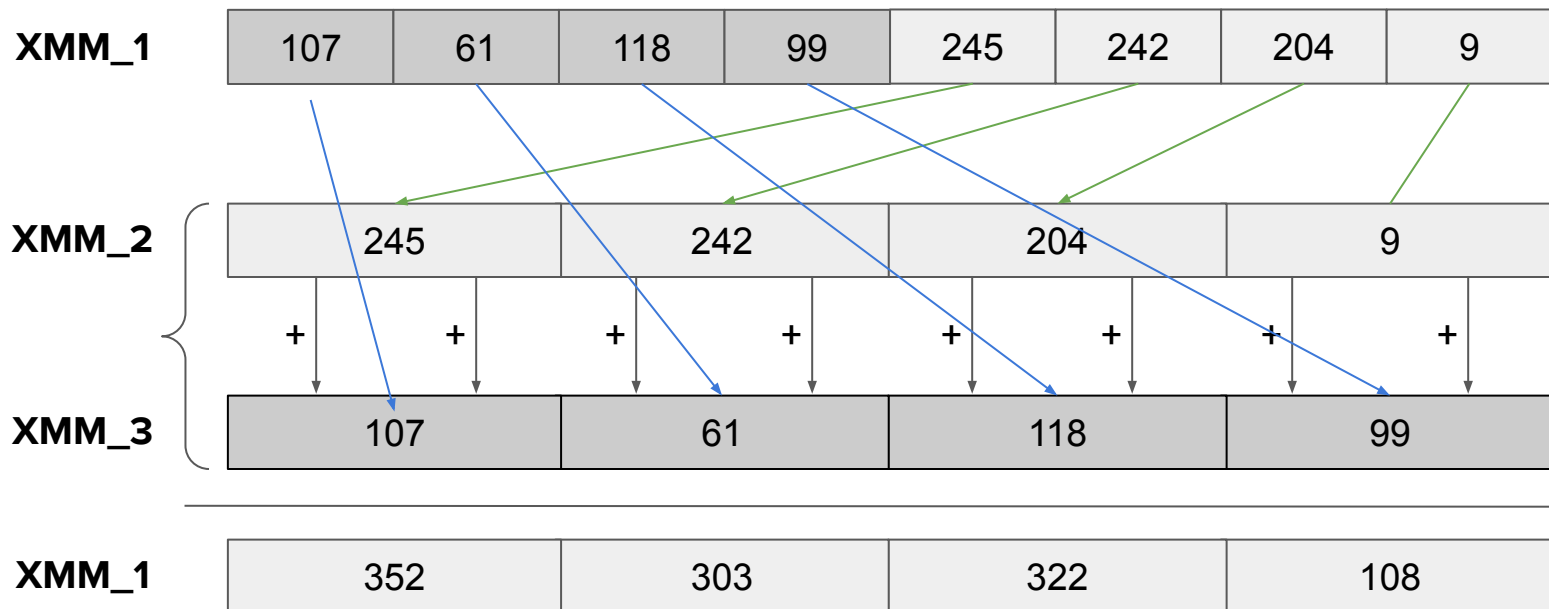
```
FOR j := 0 to 7
    i := j*16
    dst[i+15:i] := a[i+15:i] + b[i+15:i]
ENDFOR
```

## Latency and Throughput

Architecture	Latency	Throughput (CPI)
Alderlake	1	0.333333333
Icelake Xeon	1	0.33
Sapphire Rapids	1	0.333333333
Skylake	1	0.33

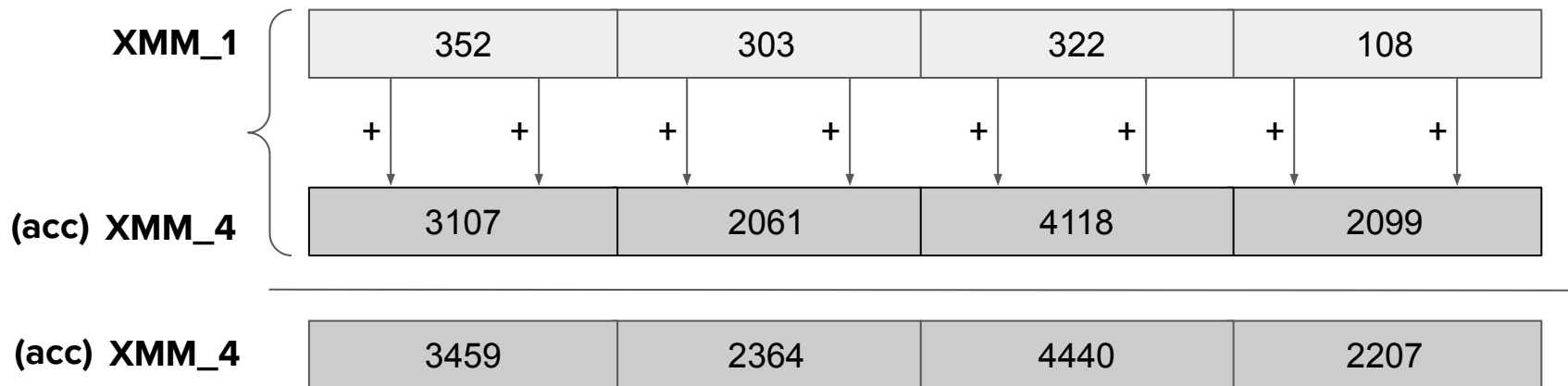
# Somma degli elementi di un array: addizioni a 32 bit

- Per gestire l'overflow è possibile estendere (unpack) le somme parziali da 16 a 32 bit, riducendo così il parallelismo, e poi aggiornare un accumulatore con elementi a 32 bit
- L'overflow non è eliminato ma si verifica con array maggiori di  $2^{32}/2^{10}$  (x4) byte



# Somma degli elementi di un array: accumulo a 32 bit

- Una volta eseguite le addizioni tra 4 coppie di dati a 32 bit è possibile procedere con la stessa operazione per aggiornare le 4 somme a 32 bit in un registro accumulatore



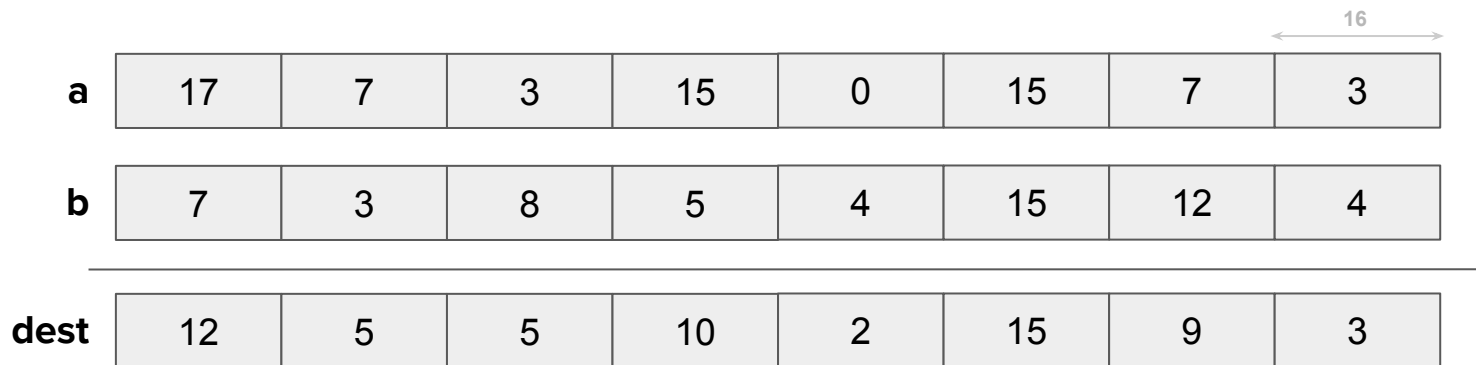
- Terminate tutte le somme è possibile sommare i 4 valori a 32 bit presenti nel registro accumulatore per ottenere il risultato desiderato
- Questo può essere fatto con istruzioni `unpack` e `add e/o extract`

## Media (approssimata): Average

- Esegue un media approssimata sommando due valori corrispondenti nei registri **a** e **b** e successivamente eseguendo uno shift a destra di un bit del risultato
- Disponibile solo per valori unsigned a 8 e 16 bit
- La latenza è 1 clock

`__m128i _mm_avg_epu8 (__m128i a, __m128i b)`

`__m128i _mm_avg_epu16 (__m128i a, __m128i b)`



# Cambiamento di segno condizionato: Sign 1/2

- Dati due registri estesi **a** e **b**, ogni elemento **i** del registro destinazione **dest** è:

$$\left\{ \begin{array}{l} \mathbf{dest}_i = \mathbf{0} \text{ se il corrispondente elemento } \mathbf{b}_i = \mathbf{0} \\ \mathbf{dest}_i = \mathbf{a}_i \text{ se il corrispondente elemento } \mathbf{b}_i > \mathbf{0} \\ \mathbf{dest}_i = -\mathbf{a}_i \text{ se il corrispondente elemento } \mathbf{b}_i < \mathbf{0} \end{array} \right.$$

- Disponibile per interi con segno a 8, 16 e 32 bit

`__m128i _mm_sign_epi8 (__m128i a, __m128i b)`

`__m128i _mm_sign_epi16 (__m128i a, __m128i b)`

`__m128i _mm_sign_epi32 (__m128i a, __m128i b)`



## Cambiamento di segno condizionato: Sign 2/2

- Esempio:

```
dest = _mm_sign_epi32(a,b);
```

<b>a</b>	-1362	-3034	32	4567
----------	-------	-------	----	------

<b>b</b>	-56	0	322	-2108
----------	-----	---	-----	-------

---

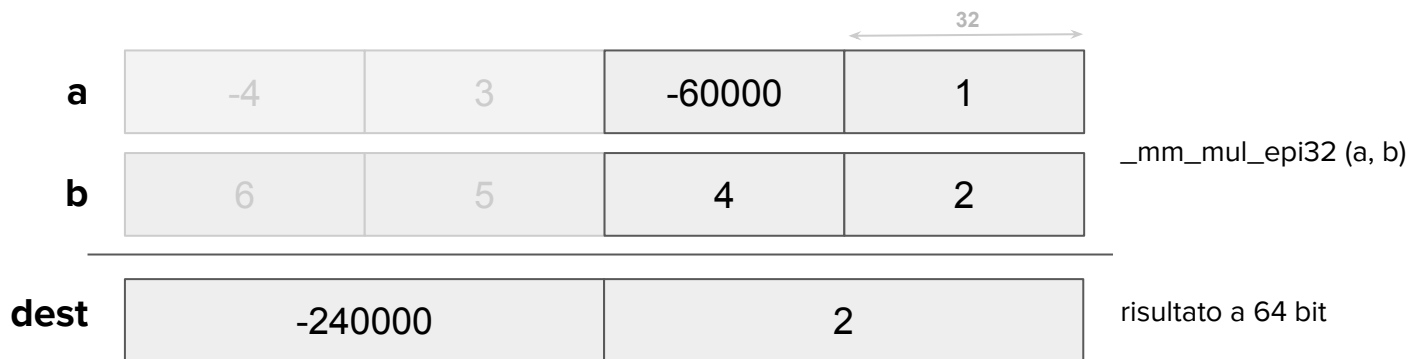
<b>dest</b>	1362	0	32	-4567
-------------	------	---	----	-------

# Moltiplicazioni tra interi: Mul

- La moltiplicazione tra interi richiede per il risultato un numero di bit doppio
- Nel caso sia necessario eseguire moltiplicazioni tra interi, esistono alcune istruzioni
- Mul, per tipi signed e unsigned a 32 bit, esegue moltiplicazioni complete a 32 bit

`__m128i _mm_mul_epu32 (__m128i a, __m128i b)`

`__m128i _mm_mul_epi32 (__m128i a, __m128i b)`

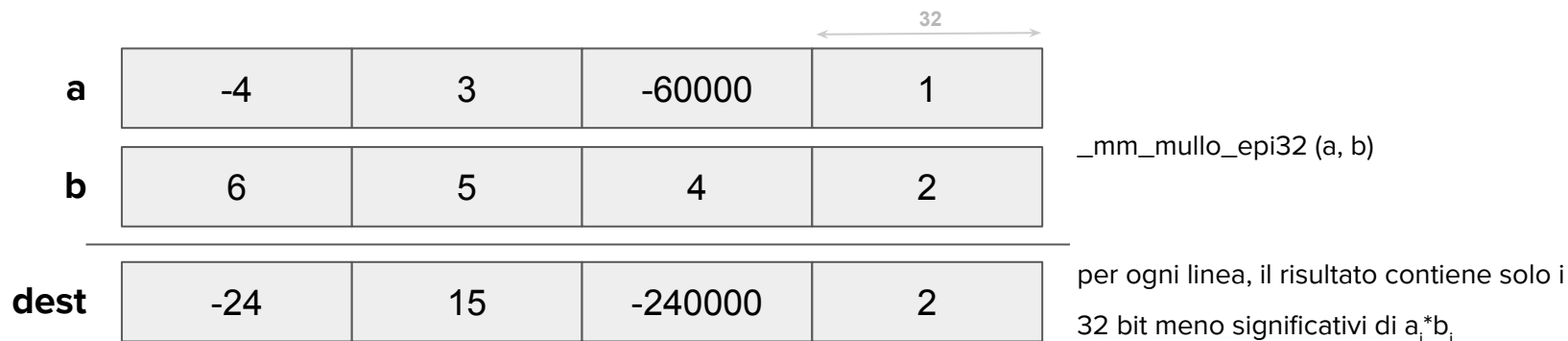


# Moltiplicazioni tra interi: Mullo

- E' anche possibile eseguire le moltiplicazioni tra tutte le linee corrispondenti
- Tuttavia, servirebbe un registro di dimensione doppia per memorizzare i risultati
- Per questo motivo, esistono istruzioni che forniscono il risultato con un numero di bit pari alla dimensione del dato di origine troncando la parte più significativa

**\_\_m128i \_\_mm\_mullo\_epi16 (\_\_m128i a, \_\_m128i b)**

**\_\_m128i \_\_mm\_mullo\_epi32 (\_\_m128i a, \_\_m128i b)**



# Moltiplicazioni tra interi: Mulhi

- Nel caso delle istruzioni *mulhi*, solo per interi a 16 bit, del risultato a 32 bit della moltiplicazione sono memorizzati nel registro destinazione solo i 16 bit più significativi

**\_\_m128i \_mm\_mulhi\_epu16 (\_\_m128i a, \_\_m128i b)**

**\_\_m128i \_mm\_mulhi\_epi16 (\_\_m128i a, \_\_m128i b)**

a	<div style="text-align: right; margin-right: 20px;">← 16 →</div>							
	42	65	2	18	60000	0	2	60000

b	42	7	8	200	2	54	3	4
---	----	---	---	-----	---	----	---	---

\_mm\_mulhi\_epu16 (a, b)

dest	0	0	0	0	1	0	0	3
------	---	---	---	---	---	---	---	---

per ogni linea, il risultato  
contiene solo i 16 bit più  
significativi di  $a_i * b_i$

# Moltiplicazioni tra interi: Mulhrs

- Mulhrs (Multiply High Round Scale) è un'istruzione piuttosto complessa
- Per ogni linea a 16 bit nei registri estesi:
  1. moltiplica due interi con segno a 16 bit, generando un risultato a 32 bit
  2. esegue uno shift a destra del risultato mantenendo solo i 18 bit più significativi
  3. somma al valore a 18 bit il valore 1 per eseguire un arrotondamento
  4. memorizza i 16 bit [16..1] del risultato nella linea corrispondente del registro destinazione
- La latenza è pari a 5 clock

```
__m128i _mm_mulhrs_epi16 (__m128i a, __m128i b)
```

# Moltiplicazioni tra interi: Mulhrs

Moltiplicazione convenzionale a 16 bit: `2000` x `1000` = `2000000`

Con `Mulhrs`, il risultato è 61 (`0000 0000 0011 1101`):

	<code>0000 0111 1101 0000</code>	x
	<code>0000 0011 1110 1000</code>	=
<hr/>		
<code>0000 0000 0001 1110 1000 0100 1000 0000</code>	>> 14	
<hr/>		
<code>0000 0000 0001 1110 10</code>	+1	
<hr/>		
<code>0000 0000 0001 1110 11</code>	=	
<hr/>		
<code>0000 0000 0011 1101</code>	$61_{10}$	

# Esercizio

**Esercizio:** Calcolare il prodotto scalare tra vettori di 4 elementi (byte signed) contenuti all'interno di due array, confrontando mediante l'utilizzo di *performance counters*, al variare delle dimensioni degli array e del grado di ottimizzazione del compilatore:

i) **codice scalare**

ii) **codice vettoriale**

-20	-5	50	-14	45	-57	77	-6	86	-7	68	-95	0	15	127	3	.....
-----	----	----	-----	----	-----	----	----	----	----	----	-----	---	----	-----	---	-------

-89	54	10	-44	-25	-27	-71	67	-17	35	87	56	-67	112	24	-43	.....
-----	----	----	-----	-----	-----	-----	----	-----	----	----	----	-----	-----	----	-----	-------

---

$\langle u_0, v_0 \rangle$	$\langle u_1, v_1 \rangle$	$\langle u_2, v_2 \rangle$	$\langle u_3, v_3 \rangle$	.....
----------------------------	----------------------------	----------------------------	----------------------------	-------

# Intrinseci universali

- Esistono librerie che consentono di scrivere codice SIMD con intrinseci indipendente dall'ISA/architettura target
- Tra queste:
  - OpenCV Universal Intrinsics  
[https://docs.opencv.org/4.9.0/d6/dd1/tutorial\\_univ\\_intrin.html](https://docs.opencv.org/4.9.0/d6/dd1/tutorial_univ_intrin.html)
  - Google Highway (Efficient and performance-portable vector software)  
<https://github.com/google/highway>