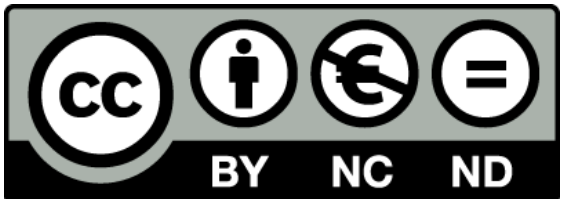
Laboratorio di Programmazione di Sistema

Interpretazione dei Dati nei Linguaggi Assembly

Luca Forlizzi, Ph.D.

Versione 20.2



Interpretazione di dato

- Come già sappiamo, negli *ASM* formato ed interpretazione dei dati non sono integrati in un unico concetto, ma sono due aspetti distinti dei dati, benché ovviamente collegati
- Nella maggior parte degli *ASM-PM*, ogni istruzione
  - Usa le stesse interpretazioni di dato per tutti gli operandi
  - Può usare diversi formati di dato
- Un formato di dato può essere associato a diverse interpretazioni di dato
- Tecnicamente, un'interpretazione di dato viene associata ad uno specifico formato di dato, tuttavia a formati differenti possono essere associate interpretazioni simili
  - Ad esempio due formati di diversa lunghezza possono entrambi essere interpretati come numeri interi

Interpretazione di dato

- Le interpretazioni di dato attribuiscono una semantica alle stringhe binarie, consentendo loro di rappresentare dati di varia natura
- Le principali interpretazioni di dato, presenti nella maggior parte degli *ASM-PM*, permettono di rappresentare
  - Numeri interi con una quantità fissata di cifre
  - Indirizzi di memoria
  - Numeri floating point
- In LPS non ci occupiamo della rappresentazione di numeri floating point
- In questa presentazione trattiamo interpretazioni di dato per numeri interi con una quantità fissata di cifre

Interpretazione di dato

- Gli *ASM-PM*, tipicamente, forniscono al programmatore la possibilità di usare (almeno) due possibili interpretazioni di dato per numeri interi
  - Una *codifica* (o *rappresentazione*) *senza segno* (*unsigned*), ovvero un'interpretazione che consente di rappresentare mediante stringhe binarie intervalli di interi non-negativi
  - Una *codifica* (o *rappresentazione*) *con segno* (*signed*), ovvero un'interpretazione che consente di rappresentare mediante stringhe binarie intervalli di interi sia negativi che non-negativi

Codifica Naturale

- La codifica senza segno di gran lunga più utilizzata è la *codifica naturale* (o *rappresentazione naturale*), in quanto semplice e immediata
- Data una parola `P`, che ha formato `F` di lunghezza `LEN` e che contiene la stringa binaria c
  - Il bit di `P` che ha posizione i, viene considerato la $i + 1$-esima cifra della rappresentazione binaria di un numero non-negativo (contando le cifre a partire da 1, da destra verso sinistra)
  - Quindi, se il bit in posizione i contiene un 1, esso ha valore pari a 2^i, altrimenti ha valore 0
  - Il valore del dato è pari alla somma dei valori di ciascuno dei bit

Codifica Naturale

- I bit (o le cifre della rappresentazione binaria) che hanno posizioni maggiori vengono detti più *significativi* di quelli che hanno posizioni minori, proprio perché la codifica naturale gli attribuisce un valore maggiore
- La codifica naturale con `LEN` cifre binarie consente di rappresentare interi compresi tra 0 e $2^{\mathrm{LEN}} - 1$

Codifiche con Segno

- Sono state ideate e utilizzate diverse codifiche con segno
- C Standard, ad esempio, prescrive che i tipi di dato interi che possono assumere anche valori negativi, debbano essere rappresentati mediante una delle seguenti
  - Modulo e segno
  - Complemento a 1
  - Complemento a 2
- Ogni implementazione di C Standard deve scegliere una di tali interpretazioni, e documentare la scelta fatta (implementation-defined behavior)

Codifiche con Segno

- La codifica modulo e segno è nota per la sua semplicità concettuale, ma non è mai stata molto utilizzata
- La codifica in complemento a 1 è invece stata utilizzata in diverse architetture di supercomputer Cray, perché consentiva di ottimizzare al massimo le unità aritmetico-logiche della CPU
- Praticamente tutte le architetture attualmente in uso utilizzano la codifica in complemento a 2
- Nel resto della presentazione, pertanto ci concentriamo su *ASM-PM* che adottano il complemento a 2 come codifica con segno

Codifica in Complemento a 2

- Consideriamo una parola `P`, che ha formato `F` di lunghezza `LEN`, e che contiene la stringa binaria c
- Nella *codifica in Complemento a 2*
  - Per $i < \texttt{LEN} - 1$, il bit in posizione i di `P` viene considerato la $i+1$-esima cifra della rappresentazione binaria del numero e quindi, se contiene un 1, esso ha valore pari a 2^i, altrimenti ha valore 0
  - Se il bit in posizione $\texttt{LEN} - 1$ contiene un 1, esso ha valore pari a $-2^{\texttt{LEN}-1}$, altrimenti ha valore 0
  - Il valore del dato è pari alla somma dei valori di ciascuno dei bit
- Dunque il valore del bit di posizione $\texttt{LEN} - 1$, determina il segno del numero: se tale bit contiene 1, il numero è negativo, in caso contrario il numero è maggiore o uguale a 0; per tale motivo il bit di posizione $\texttt{LEN} - 1$ viene chiamato *bit di segno*

Codifica in Complemento a 2

- La codifica in complemento a 2 con `LEN` cifre binarie consente di rappresentare interi compresi tra $-2^{\mathrm{LEN}-1}$ e $2^{\mathrm{LEN}-1}-1$
- Il complemento a 2 viene preferito ad altre codifiche con segno, in quanto ha le seguenti utili proprietà
  - Rappresentazione unica del numero 0
  - I comuni algoritmi per somma e differenze di interi in codifica naturale, possono essere usati anche per effettuare somma e differenza di due interi in complemento a 2
  - Esempio

Valore binario	Valore in codifica naturale	Valore in complemento a 2	
0101	5	5	+
1001	9	-7	=
1110	14	-2	

Codifica in Complemento a 2

- Il fatto che il comune algoritmo per la somma di interi in codifica naturale, permetta anche di calcolare la somma di due interi in complemento a 2, consente di utilizzare un'unica rete combinatoria per eseguire sia la somma tra interi in codifica naturale che quella tra interi in complemento a 2
- Ciò si riflette, ai livelli 2, 3 e 4, nel fatto che un'unica istruzione di addizione permette di eseguire sia la somma tra interi in codifica naturale che quella tra interi in complemento a 2
- Situazione analoga si verifica per la differenza

Codifica in Complemento a 2

- Gli *ASM-PM*, dunque, offrono al programmatore la possibilità di interpretare una stringa di bit sia come intero senza segno in codifica naturale, sia come intero con segno, mediante complemento a 2
- Nel seguito della presentazione, analizziamo in che modo il programmatore può scegliere l'interpretazione desiderata e come può operare con essa in modo corretto, in relazione alle principali operazioni su cui la scelta dell'interpretazione ha un impatto

Segno di un Intero

- Quasi tutti gli *ASM-PM* permettono di prendere decisioni su come proseguire l'esecuzione del programma sulla base del valore del segno di un intero
- Il segno di un valore intero coincide con il valore del bit di segno: se tale bit vale 1 allora il numero è negativo, altrimenti è non-negativo
- Come sappiamo, nella maggior parte degli *ASM-PM*, durante l'esecuzione dei programmi vengono memorizzate, nei CCB, delle condizioni che registrano il verificarsi di fatti
- In tali *ASM-PM*, vi sono istruzioni che registrano sotto forma di una coppia di condizioni, l'una opposta all'altra, il segno di un determinato operando o del risultato di un'operazione
  - Se il bit di segno dell'operando o del risultato vale 0, *Plus* diviene vera e *Minus* diviene falsa
  - Altrimenti, *Plus* diviene falsa e *Minus* diviene vera

Addizione e Sottrazione

- Come abbiamo già sottolineato, adottando il complemento a 2 come codifica con segno, si ha l'importante vantaggio che la medesima istruzione consente di calcolare in modo corretto la somma di due dati sia interpretandoli in codifica con segno che in codifica senza segno
- Situazione analoga per la sottrazione
- Quindi il programmatore non deve fare nulla per scegliere l'interpretazione da usare: addizione e sottrazione usano, al tempo stesso, entrambe le interpretazioni
- Tuttavia, in base alla codifica scelta, cambia la gestione delle situazioni in cui una operazione di addizione o sottrazione produca un risultato non correttamente rappresentabile

Addizione e Sottrazione

- Descriviamo cosa accade con riferimento all'addizione; per quanto riguarda la sottrazione vale un discorso simile
- L'addizione di due interi senza segno formati da `LEN` cifre può produrre un risultato di `LEN` $+ 1$ cifre
- Esempio: addizione ordinaria con `LEN` pari a 4

Valore binario	Valore in codifica naturale	
0110	6	$+$
1011	11	$=$
10001	17	

Addizione e Sottrazione

- Tipicamente, gli *ASM-PM* effettuano addizioni tra parole che hanno lo stesso formato di dato e producono un risultato in una parola che ha lo stesso formato degli operandi
- Dunque, se `LEN` è la lunghezza del formato degli operandi, può accadere che il risultato di un'addizione abbia $\texttt{LEN} + 1$ cifre, ma che la parola in cui memorizzare tale risultato abbia solo `LEN` bit: pertanto il bit più significativo del risultato non viene memorizzato
- Esempio: addizione su un *ASM-PM* con `LEN` pari a 4

Valore binario	Valore in codifica naturale	
0110	6	+
1011	11	=
0001	1	

Addizione e Sottrazione

- Il caso in cui la somma tra due interi in codifica naturale produce un risultato maggiore del massimo intero in codifica naturale rappresentabile, viene tipicamente usato per illustrare il concetto di *overflow*: il risultato di un'operazione è troppo grande per essere rappresentato in modo esatto
- In molte architetture reali, tuttavia, questo caso non viene considerato una situazione di *overflow*

Addizione e Sottrazione

- Vi è infatti un'altro modo di considerare la situazione: l'operazione di addizione effettuata dalla abstract machine non è in realtà l'ordinaria addizione, in quanto l'eventuale riporto generato dalle cifre più significative dei due operandi non viene considerato parte del risultato
- Ciò vuol dire che il risultato dell'addizione è la rappresentazione delle `LEN` cifre meno significative della somma dei due operandi
- In altre parole, indicando con v_1 e v_2 i valori dei due operandi, l'addizione effettuata dalla abstract machine calcola, in modo esatto, l'espressione matematica $(v_1 + v_2) \mod 2^{\texttt{LEN}}$, ovvero l'addizione modulo $2^{\texttt{LEN}}$

Addizione e Sottrazione

- Negli *ASM-PM* che memorizzano condizioni nei CCB, il verificarsi o meno, a seguito di un'operazione di somma, di un riporto in uscita dalle cifre più significative degli addendi, viene registrato sotto forma di una coppia di condizioni, l'una opposta all'altra, chiamate *Carry Set* e *Carry Clear*
  - Se le cifre più significative dei due numeri generano un riporto (il che implica che il risultato, interpretato come numero senza segno, non può essere contenuto nell'operando destinazione), *Carry Set* diviene vera e *Carry Clear* diviene falsa
  - Altrimenti, *Carry Set* diviene falsa e *Carry Clear* diviene vera

Addizione e Sottrazione

- Consideriamo ora cosa accade interpretando addendi e somma mediante la codifica in complemento a 2
- Nel caso di addizione tra valori dello stesso segno, può accadere che la `LEN`-esima cifra della somma abbia valore opposto a quello delle `LEN`-esime cifre dei due addendi
- Esempio con `LEN` pari a 4, i due addendi sono positivi ma la somma è negativa

Valore binario	Valore in complemento a 2	
0110	6	+
0100	4	=
1010	-6	

Addizione e Sottrazione

- La situazione in cui il risultato di un'addizione ha segno diverso da quello degli addendi viene considerata un *overflow* in quanto il risultato così ottenuto è scarsamente significativo
- Negli *ASM-PM* che memorizzano condizioni nei CCB, il verificarsi o meno di un overflow viene di solito registrato sotto forma di una coppia di condizioni, l'una opposta all'altra, chiamate *Overflow Set* e *Overflow Clear*
  - Se si verifica un *overflow* (il che implica che il risultato, interpretato come numero con segno, ha un segno diverso da quello atteso), *Overflow Set* diviene vera e *Overflow Clear* diviene falsa
  - Altrimenti, *Overflow Set* diviene falsa e *Overflow Clear* diviene vera
- Come vedremo, comunque, nel modo di gestire gli overflow vi sono differenze significative un'architettura e l'altra

Addizione e Sottrazione

- I seguenti esempi di addizioni, con operandi e risultato memorizzati in 4 bit, mostrano i valori assunti dalle condizioni *Carry Set*, *Overflow Set* in diverse situazioni
- Addendi di segno opposto

Valore binario	Valore in codifica naturale	Valore in complemento a 2	*Carry Set*	*Overflow Set*	
0110	6	6			+
1011	11	-5			=
0001	1	1	*true*	*false*	

Addizione e Sottrazione

- Addendi positivi

Valore binario	Valore in codifica naturale	Valore in complemento a 2	*Carry Set*	*Overflow Set*	
0110	6	6			+
0100	4	4			=
1010	10	-6	*false*	*true*	

- Addendi negativi

Valore binario	Valore in codifica naturale	Valore in complemento a 2	*Carry Set*	*Overflow Set*	
1100	12	-4			+
1011	11	-5			=
0111	7	7	*true*	*true*	

Confronto tra Interi

- Consideriamo i due numeri interi (di 16 bit) 0x6000 e 0x9000: quale dei due è minore dell'altro?

Confronto tra Interi

- Consideriamo i due numeri interi (di 16 bit) 0x6000 e 0x9000: quale dei due è minore dell'altro?
- Dipende dall'interpretazione di dato che si sceglie!
  - scegliendo la codifica naturale, 0x6000 è minore di 0x9000
  - scegliendo il complemento a 2, è invece il contrario in quanto 0x6000 è positivo mentre 0x9000 è negativo

Confronto tra Interi

- Un confronto tra due interi determina condizioni relative a di due tipi di relazioni tra i due numeri
  - *Relazioni di uguaglianza*, ovvero le relazioni “uguale a” e “diverso da”
  - *Relazioni d’ordine*, ovvero le relazioni “minore di”, “minore o uguale di”, “maggiore di”, “maggiore o uguale di”

Confronto tra Interi

- Sia in codifica naturale che in complemento a 2, due stringhe binarie rappresentano lo stesso numero se e solo se tutte le cifre dell'una sono uguali alle corrispondenti cifre dell'altra
- Pertanto le condizioni relative alle relazioni di uguaglianza sono indipendenti dalle interpretazioni di dato
- Negli *ASM-PM* che memorizzano condizioni nei CCB, le relazioni di uguaglianza tra due interi sono registrate sotto forma di una coppia di condizioni, l'una opposta all'altra, chiamate *Equal* e *Not Equal*
  - Se un confronto tra due valori rileva che essi sono uguali, *Equal* diviene vera e *Not Equal* diviene falsa
  - Altrimenti, *Equal* diviene falsa e *Not Equal* diviene vera

Confronto tra Interi

- Invece, come l'esempio precedente mostra, le condizioni relative alle relazioni d'ordine sono diverse per diverse interpretazioni di dato
- Dunque, negli *ASM* ci sono 8 diverse condizioni relative alle relazioni d'ordine: ciascuna condizione è il valore di una delle 4 relazioni d'ordine in base a una delle due possibili interpretazioni di dato per interi
- La seguente tabella ne descrive i nomi

	codifica con segno	**codifica senza segno**
"minore di"	*Less Than*	*Lower Than*
"minore o uguale di"	*Less Than or Equal*	*Lower Than or Same*
"maggiore di"	*Greater Than*	*Higher Than*
"maggiore o uguale di"	*Greater Than or Equal*	*Higher Than or Same*

Confronto tra Interi

- Negli *ASM-PM* che memorizzano condizioni nei CCB, le istruzioni che confrontano 2 interi determinano e memorizzano nei CCB tutte e 8 le condizioni descritte dalla precedente tabella, quindi il loro uso in un programma non dipende da un'interpretazione di dato
- L'esito di una istruzione condizionata, invece, è conseguenza del valore di una sola delle 8 condizioni
- Pertanto, la scelta dell'istruzione condizionata da utilizzare, dipende dall'interpretazione di dato che si vuole adottare

Confronto tra Interi

- Ad esempio, se si vogliono confrontare due interi v_1 e v_2 e si vuole effettuare un salto condizionato se $v_1 < v_2$ interpretando i due valori come numeri con segno, si utilizzerà un'istruzione di confronto seguita da un'istruzione di salto condizionata da *Less Than*
- Se si decidesse di cambiare interpretazione di dato, usando la codifica senza segno, l'istruzione di confronto rimarrebbe la stessa, mentre sarebbe necessario cambiare l'istruzione di salto condizionato con una che dipenda da *Lower Than*

Moltiplicazione e Divisione tra Interi

- Il prodotto tra due interi di `LEN` cifre può richiedere fino a `2LEN` cifre, sia interpretando mediante codifica naturale che mediante complemento a 2
- In genere, quindi, il risultato della moltiplicazione viene memorizzato usando un formato di lunghezza `2LEN` oppure 2 parole che hanno un formato di lunghezza `LEN`
- La divisione tra due interi di `LEN` cifre, può generare un quoziente ed un resto, ciascuno di al più `LEN` cifre, sia interpretando mediante codifica naturale che mediante complemento a 2
- In genere, essi vengono memorizzati usando un formato di lunghezza `2LEN` oppure 2 parole che hanno un formato di lunghezza `LEN`

Moltiplicazione e Divisione tra Interi

- Moltiplicazione e divisione tra interi con segno, richiedono algoritmi diversi da quelli per effettuare moltiplicazione e divisione tra interi senza segno
- Dunque, i linguaggi *ASM* (così come i corrispondenti *LM*) hanno istruzioni diverse per la moltiplicazione tra numeri con segno e la moltiplicazione tra numeri senza segno
- In modo analogo, gli *ASM* hanno istruzioni diverse per la divisione tra numeri con segno e per la divisione tra numeri senza segno

Tipi Interi Standard

- Come sappiamo, una delle caratteristiche del C è quella di poter accedere all'hardware con una flessibilità simile a quella possibile usando il linguaggio macchina
- Per supportare questa caratteristica, C Standard definisce diversi tipi di intero, in modo tale da consentire alle implementazioni di poter utilizzare, tramite diversi tipi di dato, tutti i diversi formati disponibili sulle ISA su cui sono definite
- Studiamo ora le caratteristiche fondamentali dei tipi interi definiti da C Standard che si collegano ai formati e alle interpretazioni di dato degli *ASM-PM* e delle *ISA*; per ulteriori informazioni sui tipi interi in C, si vedano **[C99]** e i capitoli 7 e 23 di **[Ki]**

Signed Standard Integer Types

- Gli *standard signed integer types*, sotto elencati, devono essere obbligatoriamente definiti da tutte le implementazioni C

1	`signed char`
2	`signed short int`
3	`signed int`
4	`signed long int`
5	`signed long long int`

- In tutti i nomi, tranne che in `signed char`, le keyword `signed` e `int` sono opzionali
- Tuttavia per usare il nome `signed int` almeno una delle due deve essere presente, tranne che in C Tradizionale
- Il tipo `signed long long int` non è presente in C89

Signed Standard Integer Types

- C Standard non stabilisce esattamente le caratteristiche di tutti questi tipi, dando lo status di implementation-defined a molte di esse
- Il motivo è che
  - ISA diverse hanno formati di dato con caratteristiche diverse
  - C Standard intende permettere a ciascuna implementazione di utilizzare nel modo più efficiente possibile tutti i formati di dato disponibili nella ISA per cui l'implementazione produce il codice eseguibile

Signed Standard Integer Types

- C Standard stabilisce però alcuni vincoli
  - Tutti i tipi devono rappresentare un intervallo di interi che comprende lo 0, mediante una delle tre codifiche possibili (modulo e segno, complemento a 1, complemento a 2)
  - Per ciascun tipo, tranne `signed char`, l'intervallo dei valori rappresentabili deve contenere l'intervallo dei valori rappresentabili dal tipo precedente nell'elenco
- È possibile che due diversi tipi dell'elenco rappresentino lo stesso intervallo; ad esempio `short` e `int` potrebbero entrambi rappresentare l'intervallo $[-2^{15}, 2^{15} - 1]$

Signed Standard Integer Types

- Il programmatore può conoscere quali sono gli estremi dell'intervallo di valori rappresentato da un tipo, così come altre caratteristiche, attraverso i valori di alcune costanti definite nell'header `limits.h`; ad esempio i valori minimo e massimo rappresentabili da `int` sono, rispettivamente, i valori di `INT_MIN` e `INT_MAX`
- C Standard stabilisce un intervallo minimo che ciascun tipo deve rappresentare; ad esempio devono valere le seguenti relazioni: `INT_MIN` ≤ -32767 e $32767 \leq$ `INT_MAX`

Unsigned Standard Integer Types

- Per ciascuno degli standard signed integer types, ogni implementazione C deve definire un corrispondente tipo per rappresentare interi non-negativi, avente nome ottenuto modificando la keyword `signed` in `unsigned`
- I tipi così ottenuti sono chiamati *standard unsigned integer types*

1	`unsigned char`
2	`unsigned short int`
3	`unsigned int`
4	`unsigned long int`
5	`unsigned long long int`

- In tutti i nomi, tranne che in `unsigned char`, la keyword `int` è opzionale
- Il tipo `unsigned long long int` non è presente in C89

Unsigned Standard Integer Types

- L'idea dietro a questa definizione è che ciascuna coppia formata da tipo `signed` e dal corrispondente tipo `unsigned` dovrebbe essere usata da un'implementazione C per rappresentare uno dei formati di dato per interi disponibile nella ISA per cui l'implementazione produce il codice eseguibile
- Attraverso i tipi `signed` si effettuano operazioni sui dati interpretando i formati con una codifica con segno, mentre attraverso i tipi `unsigned` si effettuano operazioni interpretando i formati con una codifica senza segno
- Infatti C Standard stabilisce che i valori di ogni tipo `signed` devono occupare la stessa quantità di bit dei valori del corrispondente tipo `unsigned`

Unsigned Standard Integer Types

- Il valore minimo rappresentabile da ciascun tipo `unsigned` è ovviamente 0
- Il valore massimo rappresentabile da ciascun tipo è il valore di una costante definita nell'header `limits.h`; ad esempio il valore massimo rappresentabile da `unsigned int` è il valore di `UINT_MAX`
- Anche per i tipi `unsigned`, C Standard stabilisce un intervallo minimo che ciascuno di essi deve rappresentare; ad esempio deve valere $65535 \leq$ `UINT_MAX`

Aritmetica sui tipi interi in C

- Le operazioni aritmetiche sui tipi interi presentano una importante differenza, che ne rispecchia una analoga vista in precedenza nei linguaggi *ASM*, a seconda se si applichino a valori di un tipo `signed` o a valori di un tipo `unsigned`
- Nel caso in cui un'operazione aritmetica ordinaria tra valori di un tipo `unsigned` produca un risultato R che non possa essere rappresentato in tale tipo (perché negativo o perché maggiore del massimo intero rappresentabile), l'operatore aritmetico del C produce come risultato il valore di R modulo il valore del più grande intero rappresentabile dal tipo aumentato di 1

Aritmetica sui tipi interi in C

- Ad esempio, in un'implementazione per cui `UINT_MAX` vale 65535, il prodotto tra 1000 e 100 non è rappresentabile nel tipo `unsigned int`
- Il risultato di `1000U * 100U` è comunque definito ed è pari al valore $100000 \mod 65536 = 34464$ (infatti il quoziente della divisione intera di 100000 per 65536 è 1, e quindi il resto è pari a $100000 - 1 \cdot 65536$
- Questo comportamento degli operatori aritmetici applicati a valori di un tipo `unsigned` non viene considerato un'anomalia: è esattamente questa la semantica che C Standard definisce per tali operatori

Aritmetica sui tipi interi in C

- Nel caso di operazioni aritmetiche tra valori di un tipo `signed` si vorrebbe invece che gli operatori del C producessero lo stesso risultato delle operazioni aritmetiche ordinarie
- Se un'operazione aritmetica ordinaria tra valori di un tipo `signed` produce un risultato R che non può essere rappresentato in tale tipo (ovvero in caso di overflow), si ha un undefined behavior
- C Standard sceglie di rendere indefinito il comportamento, per permettere alle implementazioni C di gestire le situazioni di overflow nel modo più appropriato, in relazione alla ISA per cui l'implementazione produce il codice eseguibile
- Infatti, come vedremo analizzando i casi di riferimento usati in LPS, vi sono differenze significative tra diverse architetture nel modo di gestire gli overflow

Codifiche di interi in MC68000

- In MC68000, i dati possono essere interpretati come interi senza segno o interi con segno codificati in complemento a 2
- Entrambe le interpretazioni di dato possono essere applicate a tutti e 3 i formati di dato generali

Istruzioni che Determinano Condizioni

- Numerose istruzioni memorizzano condizioni in `ccr`, seguendo il tipico approccio della maggior parte degli *ASM*
  - *Plus* e *Minus* vengono determinate da molte istruzioni in base al segno di un risultato o di un determinato operando
  - *Carry Set*, *Overflow Set* e le loro opposte, vengono determinate dalle istruzioni aritmetiche e da altre istruzioni
  - Le condizioni relative a relazioni di uguaglianza e a relazioni d'ordine, vengono determinate da istruzioni che confrontano due operandi tra loro, oppure confrontano 0 con il valore di un determinato operando o risultato
- Tutte queste istruzioni determinano sia le condizioni relative alla codifica con segno che a quella senza segno, pertanto sono indipendenti dall'interpretazione di dato desiderata
- In **[M68000]**, viene descritto come ciascuna istruzione dell'ASM MC68000 modifica le condizioni memorizzate in `ccr`

Istruzioni Condizionate

- MC68000 dispone di due tipi di istruzioni condizionate
  - le istruzioni di salto condizionato `b`**cc**
  - le istruzioni di memorizzazione di condizione `s`**cc**
- In una precedente lezione abbiamo descritto le regole di entrambi i tipi di istruzioni
- Ora, dopo aver studiato il ruolo delle interpretazioni di dato e descritto le condizioni *Carry Set*, *Overflow Set* e le loro opposte, possiamo elencare tutti i possibili valori dei codici **cc** e i corrispondenti significati

Istruzioni Condizionate

- La tabella seguente mostra i valori di **cc** e i corrispondenti significati, per le condizioni *Plus*, *Carry Set*, *Overflow Set*, e le loro opposte

Codice	Condizione	Significato corrispondente
`mi`	*Minus*	un valore è negativo
`pl`	*Plus*	un valore è non-negativo
`cc`	*Carry Clear*	non si è generato un riporto dalle cifre più significative
`cs`	*Carry Set*	si è generato un riporto dalle cifre più significative
`vc`	*Overflow Clear*	non si è verificato un *overflow*
`vs`	*Overflow Set*	si è verificato un *overflow*

Istruzioni Condizionate

- La tabella seguente (già presentata in una precedente lezione) mostra i valori di **cc** e i corrispondenti significati, per le condizioni relative alle relazioni di uguaglianza

Codice	Condizione	Operatore C corrispondente
`eq`	*Equal*	`==`
`ne`	*Not Equal*	`!=`

Istruzioni Condizionate

- La tabella seguente (già presentata in una precedente lezione) mostra i valori di **cc** e i corrispondenti significati, per le condizioni relative alle relazioni d'ordine determinate applicando la codifica con segno

Codice	Condizione	Operatore C corrispondente
`lt`	*Less Than*	<
`le`	*Less Than or Equal*	<=
`gt`	*Greater Than*	>
`ge`	*Greater Than or Equal*	>=

Istruzioni Condizionate

- La tabella seguente mostra i valori di **cc** e i corrispondenti significati, per le condizioni relative alle relazioni d'ordine determinate applicando la codifica senza segno

Codice	Condizione	Operatore C corrispondente
`lo`	*Lower Than*	<
`ls`	*Lower Than or Same*	<=
`hi`	*Higher Than*	>
`hs`	*Higher Than or Same*	>=

Moltiplicazione e Divisione

- Le istruzioni `muls` e `divs`, già presentate in una precedente lezione, effettuano moltiplicazione e divisione interpretando i loro operandi come numeri con segno
- Le istruzioni `mulu` e `divu` effettuano le analoghe operazioni interpretando i loro operandi come numeri senza segno
- Tranne che per l'interpretazione di dato usata, l'istruzione `mulu` funziona in modo analogo a `muls` e l'istruzione `divu` in modo analogo a `divs`

Codifiche di interi in MIPS32

- In MIPS32, i dati possono essere interpretati come interi senza segno o interi con segno codificati in complemento a 2
- Entrambe le interpretazioni di dato possono essere applicate a tutti i formati di dato generali

Addizione e Sottrazione

- Nell'eseguire una somma o sottrazione, se le cifre più significative dei due numeri generano un riporto (il che implica che il risultato, interpretato come numero senza segno, non può essere contenuto nell'operando destinazione), non vi è alcuna conseguenza
- Le situazioni di *overflow* causate dalla somma o sottrazione di numeri con segno possono essere gestite in due modi diversi
  - Le istruzioni `add` e `sub`, in caso di *overflow*, avviano una *eccezione*, ossia una particolare azione che interrompe la normale esecuzione del programma, che approfondiremo in future lezioni di LPS
  - Le istruzioni `addu` e `subu` producono in ogni caso il risultato senza segnalare in alcun modo un eventuale *overflow*

Istruzioni Condizionate

- MIPS32-MARS dispone di due tipi di istruzioni condizionate
  - le istruzioni di salto condizionato
  - le istruzioni di memorizzazione di condizione
- In una precedente lezione abbiamo descritto alcune di queste istruzioni, ora completiamo l'elenco
- MIPS32 non ha istruzioni condizionate relative a situazioni di overflow o riporto generato dalle cifre più significative degli operandi

Istruzioni Condizionate

- Come detto in una precedente lezione, le istruzioni condizionate relative alle relazioni di uguaglianza sono
  - le istruzioni di salto condizionato a 2 operandi b**cc**z
  - le istruzioni di salto condizionato a 3 operandi b**cc**
  - le istruzioni di memorizzazione di condizione s**cc**
- I valori dei codici **cc** sono riportati nella tabella seguente, con i corrispondenti significati

Codice	Condizione	Operatore C corrispondente
`eq`	*Equal*	`==`
`ne`	*Not Equal*	`!=`

Istruzioni Condizionate

- In una precedente lezione sono state presentate le seguenti istruzioni condizionate relative a relazioni d'ordine
  - le istruzioni di salto condizionato a 2 operandi `b`**`cc`**`z`
  - le istruzioni di salto condizionato a 3 operandi `b`**`cc`**
  - le istruzioni di memorizzazione di condizione `s`**`cc`** (c'è un'irregolarità sintattica nel caso del codice `lt`, si veda la documentazione di MARS)
- Tali istruzioni adottano la codifica con segno
- I valori dei codici **cc** sono riportati nella tabella seguente, con i corrispondenti significati

Codice	Condizione	Operatore C corrispondente
`lt`	*Less Than*	<
`le`	*Less Than or Equal*	<=
`gt`	*Greater Than*	>
`ge`	*Greater Than or Equal*	>=

Istruzioni Condizionate

- MIPS32-MARS ha anche istruzioni condizionate relative a relazioni d'ordine che adottano la codifica senza segno
  - istruzioni di salto condizionato a 3 operandi `b`**`cc`**`u`
  - istruzioni di memorizzazione di condizione `s`**`cc`**`u` (c'è un'irregolarità sintattica nel caso del codice `lt`, si veda la documentazione di MARS)
- I valori dei codici **cc** sono riportati nella tabella seguente, con i corrispondenti significati

Codice	Condizione	Operatore C corrispondente
`lt`	*Less Than*	<
`le`	*Less Than or Equal*	<=
`gt`	*Greater Than*	>
`ge`	*Greater Than or Equal*	>=

Moltiplicazione e Divisione

- MIPS32-MARS ha diverse istruzioni per effettuare moltiplicazioni e divisioni
- Le istruzioni viste in precedenti lezioni interpretano i loro operandi come interi con segno
- Come nel caso di MC68000, per effettuare moltiplicazioni e divisioni tra interi senza segno si utilizzano istruzioni diverse
- Le istruzioni per effettuare moltiplicazioni e divisioni tra interi senza segno hanno regole simili a quelle che effettuano analoghe operazioni tra interi con segno; hanno anche nomi simili, che si caratterizzano per il suffisso `u`
- Inoltre, come per addizione e sottrazione, esistono istruzioni che ignorano eventuali overflow ed altre che invece, in caso di overflow, generano un'eccezione
- Per i dettagli si rimanda a **[MIPS32]**

Sessione di Esercizi: Interpretazione_Dati_ASM

- Svolgere il gruppo di esercizi **Interpretazione_Dati_ASM**
- Per ulteriori spiegazioni sui contenuti degli esercizi, si vedano **[C99]**, **[M68000]** e **[MIPS32]**