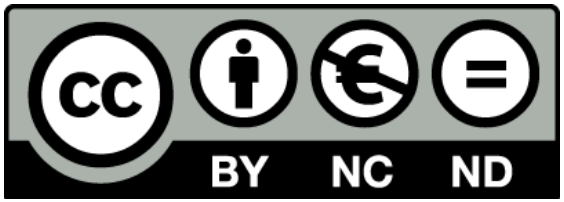
Laboratorio di Programmazione di Sistema

Operazioni a Livello di Bit

Luca Forlizzi, Ph.D.

Versione 20.1



Introduzione

- La maggior parte delle istruzioni *ASM*, hanno come operandi delle parole standard, le cui lunghezze sono fissate dai formati di dato definiti dagli *ASM-PM*
- In alcune applicazioni è utile operare su singoli bit o su insiemi di bit che non corrispondono a una parola, ad esempio
  - Gestione di molti dati, ciascuno dei quali richiede pochi bit
  - Bit speciali che controllano sensori o attuatori di dispositivi hardware
- Abbiamo accennato al fatto che alcuni *ASM-PM* definiscono delle parole speciali adatte ad alcune di queste operazioni; forniremo maggiori informazioni in proposito al termine di questa presentazione

Introduzione

- Vi sono, invece, delle tecniche generali per effettuare tali operazioni anche su *ASM-PM* privi di parole speciali, che descriveremo in dettaglio in questa presentazione
- Esse sono basate su un piccolo numero di *operazioni di manipolazione di bit*, disponibili in quasi tutti gli *ASM-PM*, dette anche *operazioni logiche* in quanto alcune di esse sono tipiche della logica Booleana
- Le istruzioni *ASM* che effettuano tali operazioni, vengono chiamate *istruzioni di manipolazione di bit* o *istruzioni logiche*

Relazioni tra Bit

- Ricordiamo che ciascun formato di dato **F** assegna una posizione a ciascuno dei bit che formano una parola definita da **F**
- Ovvero a ciascun bit viene assegnato un numero compreso tra 0 e $L - 1$, dove L è la lunghezza di **F**
- Se si interpreta in codifica naturale il contenuto di una parola, il bit in posizione i memorizza la $i + 1$-esima cifra di un numero intero e quindi ha valore pari a 2^i
- Per questo motivo, sono detti più *significativi* i bit che hanno posizioni maggiori

Relazioni tra Bit

- Dati due bit b_1, b_2 contenuti in memoria, diciamo che b_1 *precede* b_2 ($b_1 <_{\mathrm{p}} b_2$) se
  - b_1 appartiene ad un byte con indirizzo minore di quello che contiene b_2
  - b_1 e b_2 sono contenuti nello stesso byte e la posizione di b_1 è minore di quella di b_2
- Dati due bit b_1, b_2 contenuti in memoria, diciamo che b_2 *segue* b_1 ($b_2 >_{\mathrm{p}} b_1$) se b_1 precede b_2
- Dalla *precedenza* tra i bit della memoria derivano i seguenti concetti
  - *predecessore* e *successore* di un bit
  - *minimo* e *massimo* in un insieme di bit

Relazioni tra Bit

- Dati due bit b_1, b_2 contenuti in memoria diciamo che b_1 è il *predecessore* di b_2 se:
  - b_1 e b_2 appartengono allo stesso byte e la posizione di b_2 è pari alla posizione di b_1 aumentata di 1
  - Oppure, indicando con B_1 il byte cui appartiene b_1 e con B_2 il byte cui appartiene b_2, si ha:
    - B_1 è il predecessore di B_2
    - b_1 è il bit di posizione $M - 1$ di B_1 (M è la lunghezza di un byte)
    - b_2 è il bit di posizione 0 di B_2
- Dati due bit b_1, b_2 contenuti in memoria, diciamo che b_2 è il *successore* di b_1 se b_1 è il predecessore di b_2

Relazioni tra Bit

- Dato un'insieme F di bit contenuti in memoria, diciamo che un bit b_m è il *minimo* di F se
  - b_m appartiene ad F
  - b_m precede ogni bit di F tranne se stesso
- Dato un'insieme F di bit contenuti in memoria, diciamo che un bit b_M è il *massimo* di F se
  - b_M appartiene ad F
  - b_M segue ogni bit di F tranne se stesso

Istruzioni di Manipolazione di Bit

- Quasi tutti gli *ASM* forniscono istruzioni per effettuare le più comuni operazioni di manipolazione di bit
  - Operazioni di spostamento di bit
    - Scorrimento aritmetico
    - Scorrimento logico
    - Rotazione
  - Operazioni logiche bit-a-bit
    - **not**
    - **and**
    - **or**
    - **xor**
- Attraverso l'uso combinato di tali istruzioni, è possibile memorizzare e gestire dati in gruppi di bit che non formano parole dell'*ASM-PM*

Scorrimento

- Un'operazione di *scorrimento* (*shift*) si applica ad una parola di L bit e consiste nell'assegnare ai bit che la formano, i valori che in precedenza avevano altri bit della parola
- Le operazioni di *scorrimento aritmetico* tengono conto del fatto che molto spesso il contenuto delle parole viene interpretato come numero in complemento a 2
- Le operazioni di *scorrimento logico* invece considerano il contenuto delle parole come stringhe di bit, senza adottare una particolare interpretazione di dato
- In un'operazione di *rotazione* (*rotate*) il contenuto di nessun bit viene perduto

Scorrimento Logico a Sinistra

- *Scorrimento logico a sinistra di* 1:
  - Per $0 < p \leq L - 1$, il bit di posizione p assume il valore che, in precedenza, aveva il bit in posizione $p - 1$
  - Il bit in posizione 0 assume valore 0
- Si noti che il valore che il bit in posizione $L - 1$ aveva prima dell'operazione viene perduto
- *Scorrimento logico a sinistra di* k: ripetere k volte uno scorrimento logico a sinistra di 1

Scorrimento Aritmetico a Sinistra

- *Scorrimento aritmetico a sinistra di* 1:
  - Per $0 < p \leq L - 1$, il bit di posizione p assume il valore che, in precedenza, aveva il bit in posizione $p - 1$
  - Se il bit in posizione $L - 1$ cambia di segno, viene segnalato un *overflow*
  - Il bit in posizione 0 assume valore 0
- Si noti che il valore che il bit in posizione $L - 1$ aveva prima dell'operazione viene perduto
- *Scorrimento aritmetico a sinistra di* k: ripetere k volte uno scorrimento aritmetico a sinistra di 1

Scorrimento Logico a Destra

- *Scorrimento logico a destra di* 1:
  - Per $0 \leq p < L - 1$, il bit di posizione p assume il valore che, in precedenza, aveva il bit in posizione $p + 1$
  - Il bit in posizione $L - 1$ assume valore 0
- Si noti che il valore che il bit in posizione 0 aveva prima dell'operazione viene perduto
- *Scorrimento logico a destra di* k: ripetere k volte uno scorrimento logico a destra di 1

Scorrimento Aritmetico a Destra

- *Scorrimento aritmetico a destra di* 1:
  - Per $0 \leq p < L - 1$, il bit di posizione p assume il valore che, in precedenza, aveva il bit in posizione $p + 1$
  - Il bit in posizione $L - 1$ non viene modificato
- *Scorrimento aritmetico a destra di* k: ripetere k volte uno scorrimento aritmetico a destra di 1

Scorrimento e Aritmetica

- Una delle applicazioni principali delle operazioni di scorrimento è aritmetica
- Data una stringa di L bit, interpretata come numero senza segno N
  - effettuare lo scorrimento logico a sinistra di k sulla stringa, equivale a calcolare $(N \cdot 2^k) \mod 2^L$
  - effettuare lo scorrimento logico a destra di k sulla stringa, equivale a calcolare $N \div 2^k$

Scorrimento e Aritmetica

- Data una stringa di L bit, interpretata come numero con segno N
  - effettuare lo scorrimento logico a sinistra di k sulla stringa, equivale a calcolare $N \cdot 2^k$
  - effettuare lo scorrimento logico a destra di k sulla stringa, equivale a calcolare $N \div 2^k$
- Le operazioni di scorrimento hanno implementazioni hardware molto più efficienti delle operazioni di moltiplicazione e divisione

Principali Istruzioni/Operatori di Scorrimento

- In C
  - << scorrimento a sinistra, aritmetico o logico in base al tipo dell'operando sinistro
  - >> scorrimento logico a destra se il tipo dell'operando sinistro è senza segno, altrimenti il risultato è implementation defined
- In MIPS32
  - `sll` scorrimento logico a sinistra
  - `srl` scorrimento logico a destra
  - `sra` scorrimento aritmetico a destra
- In MC68000
  - `asl` scorrimento aritmetico a sinistra
  - `lsl` scorrimento logico a sinistra
  - `asr` scorrimento aritmetico a destra
  - `lsr` scorrimento logico a destra

Rotazione a Sinistra

- *Rotazione a sinistra di* 1:
  - Per $0 < p \leq L - 1$, il bit di posizione p assume il valore che, in precedenza, aveva il bit in posizione $p - 1$
  - Il bit in posizione 0 assume il valore che, in precedenza, aveva il bit in posizione $L - 1$
- *Rotazione a sinistra di* k: ripetere k volte una rotazione a sinistra di 1
- Istruzioni di rotazione a sinistra in MC68000-ASM1: `rol`, `roxl`
- Istruzione di rotazione a sinistra in MIPS32-MARS: `rol`
- In C le rotazioni a sinistra vengono effettuate combinando operazioni di *shift* e manipolazione di bit

Rotazione a Destra

- *Rotazione a destra di* 1:
  - Per $0 \leq p < L - 1$, il bit di posizione p assume il valore che, in precedenza, aveva il bit in posizione $p + 1$
  - Il bit in posizione $L - 1$ assume il valore che, in precedenza, aveva il bit in posizione 0
- *Rotazione a destra di* k: ripetere k volte una rotazione a destra di 1
- Istruzioni di rotazione a destra in MC68000-ASM1: `ror`, `roxr`
- Istruzione di rotazione a destra in MIPS32-MARS: `ror`
- In C le rotazioni a destra vengono effettuate combinando operazioni di *shift* e manipolazione di bit

Operazioni *bit-a-bit*

- Le operazioni *bit-a-bit* (*bitwise operations*) effettuano una computazione indipendente su ciascuno dei bit dei loro operandi
  - Tutti i bit di un operando vengono coinvolti, ma l'esito della computazione su ogni singolo bit di un operando, non dipende dal valore degli altri bit dello stesso operando
  - Per tutti i bit di un operando viene effettuata la stessa computazione
  - Le computazioni effettuate sui diversi bit di un operando, essendo indipendenti le une dalle altre, sono svolte in parallelo
- Le più comuni operazioni bit-a-bit effettuano, su ciascun bit, un'operazione Booleana

Operazioni Booleane

- Ricordiamo la definizione, tramite tabelle della verità, delle più comuni operazioni Booleane su bit

X	*Not* X
0	1
1	0

X	Y	X *Or* Y	X *And* Y	X *Or-esclusivo* Y
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	0

Complemento

- La più semplice operazione bit-a-bit è il *Complemento*, detto anche *Not*
- Il *Not* ha un singolo operando *O* e produce un risultato *R* che ha lo stesso numero di bit di *O*
- Ciascun bit di *R* ha valore inverso di quello del bit che ha la stessa posizione nell'operando *O*
- Esempi operazione *Not*

O	Not *O*
0x00	0xFF
0x01	0xFE
0xF0	0x0F
0x5C	0xA3
0xFF	0x00

And

- L'operazione *And* bit-a-bit si applica a due operandi O_1 ed O_2 che hanno lo stesso numero di bit N e produce un risultato R formato da N bit
- Il bit in posizione k del risultato ha valore pari al risultato dell'operazione Booleana di *And* tra il bit in posizione k di O_1 e il bit in posizione k di O_2
- Esempi operazione *And*

O_1	O_2	O_1 And O_2
0x00	0xFF	0x00
0x59	0xAD	0x09

Or

- L'operazione *Or* bit-a-bit si applica a due operandi O_1 ed O_2 che hanno lo stesso numero di bit N e produce un risultato R formato da N bit
- Il bit in posizione k del risultato ha valore pari al risultato dell'operazione Booleana di *Or* tra il bit in posizione k di O_1 e il bit in posizione k di O_2
- Esempi operazione *Or*

O_1	O_2	O_1 Or O_2
0x0F	0xF0	0xFF
0x59	0xAD	0xFD

Or-esclusivo

- L'operazione *Or-esclusivo* bit-a-bit, detta brevemente *Xor* o anche *Eor*, si applica a due operandi O_1 ed O_2 che hanno lo stesso numero di bit N e produce un risultato R formato da N bit
- Il bit in posizione k del risultato ha valore pari al risultato dell'operazione Booleana di *Or-esclusivo* tra il bit in posizione k di O_1 e il bit in posizione k di O_2
- Esempi operazione *Or-esclusivo*

O_1	O_2	O_1 Xor O_2
0x3C	0x0F	0x33
0x59	0xAD	0xF4

Modificare Bit con Operazioni Logiche

- Le operazioni logiche bit-a-bit permettono di modificare individualmente i bit della memoria
- Sia x il valore di un bit, l'operazione Booleana di *Or* ha la seguente proprietà
  - $x \; Or \; 0 = x$
  - $x \; Or \; 1 = 1$
- Quindi, data una parola D di N bit, è possibile porre a 1 determinati bit senza modificare gli altri, effettuando *Or* bit-a-bit tra D e una costante di N bit, chiamata *maschera*, che ha:
  - i bit che hanno la stessa posizione dei bit di D che si vuole porre a 1, al valore 1
  - i bit che hanno la stessa posizione dei bit di D che non si vuole modificare, al valore 0

Modificare Bit con Operazioni Logiche

- Esempi: maschere per porre a 1 alcuni bit di parole di 8 bit

Posizioni	Maschera
0	0x01
1	0x02
4	0x10
7	0x80
1 e 4	0x12
0, 1 e 4	0x13
1, 3, 6, 7	0xCA

Modificare Bit con Operazioni Logiche

- Sia x il valore di un bit, l'operazione Booleana di *And* ha la seguente proprietà
  - x *And* $0 = 0$
  - x *And* $1 = x$
- Quindi, data una parola D di N bit, è possibile porre a 0 determinati bit senza modificare gli altri, effettuando *And* bit-a-bit tra D e una costante di N bit, chiamata *maschera*, che ha:
  - i bit che hanno la stessa posizione dei bit di D che si vuole porre a 0, al valore 0
  - i bit che hanno la stessa posizione dei bit di D che non si vuole modificare, al valore 1

Modificare Bit con Operazioni Logiche

- Esempi: maschere per porre a 0 alcuni bit di parole di 8 bit

Posizioni	Maschera
0	0xFE
1	0xFD
4	0xEF
7	0x7F
1 e 4	0xED
0, 1 e 4	0xEC
1, 3, 6, 7	0x35

Modificare Bit con Operazioni Logiche

- Sia x il valore di un bit, l'operazione Booleana di *Or-esclusivo* ha la seguente proprietà
  - x *Xor* $0 = x$
  - x *Xor* $1 =$ inverso di x
- Quindi, data una parola D di N bit, è possibile invertire determinati bit senza modificare gli altri, effettuando *Xor* bit-a-bit tra D e una costante di N bit, chiamata *maschera*, che ha:
  - i bit che hanno la stessa posizione dei bit di D che si invertire, al valore 1
  - i bit che hanno la stessa posizione dei bit di D che non si vuole modificare, al valore 0

Modificare Bit con Operazioni Logiche

- Esempi: maschere per invertire alcuni bit di parole di 8 bit

Posizioni	Maschera
0	0x01
1	0x02
4	0x10
7	0x80
1 e 4	0x12
0, 1 e 4	0x13
1, 3, 6, 7	0xCA

Principali Istruzioni/Operatori bit-a-bit

- In C
  - `~` *Not*
  - `|` *Or*
  - `&` *And*
  - `^` *Xor*
- In MIPS32
  - `not` *Not*
  - `or` *Or*
  - `and` *And*
  - `xor` *Xor*
- In MC68000
  - `not` *Not*
  - `or` *Or*
  - `and` *And*
  - `eor` *Xor*

Or-esclusivo e Crittografia

- L'operazione di *Or-esclusivo* bit-a-bit trova interessanti applicazioni nel campo della crittografia
- Esse si basano sulla seguente proprietà dell'*Or-esclusivo*: date due parole X e M formate da un ugual numero di bit, si ha che $(X\ Xor\ M)\ Xor\ M = X$
- Uno dei modi più semplici per crittografare un messaggio, è quello di fare *Or esclusivo* tra ciascun carattere del messaggio e un valore segreto detto *chiave*
- La decrittazione avviene effettuando un *Or-esclusivo* tra ciascun carattere del messaggio crittato e la *chiave*
- La sezione 20.1 di **[Ki]** mostra un esempio completo
- Anche le tecniche crittografiche *One-time pad* utilizzano spesso l'*Or-esclusivo*

Bit-field

- Un gruppo F di bit che non forma una parola standard, è un *bit-field* se per ogni b contenuto in F
  - b è il minimo di F
  - Oppure il predecessore di b è contenuto in F
- In altre parole, un bit-field è un insieme di bit "senza buchi", che però non corrisponde ad una parola standard
- I bit-field sono utili per memorizzare dati che richiedono un numero di bit diverso dalle lunghezze delle parole standard, o per altre ragioni non possono essere memorizzati in parole standard
- Spesso vengono impiegati per ridurre il consumo di memoria
- Ad esempio, per memorizzare una grande quantità di interi compresi tra 0 e 15, si possono utilizzare bit-field formati da 4 bit

Bit-field

- Due bit-field si dicono *adiacenti* se uno di essi contiene un bit che è predecessore di uno dei bit dell'altro bit-field
- Alcuni *ASM-PM*, come sappiamo, definiscono alcuni bit-field come parole speciali, e quindi forniscono alcune istruzioni per eseguire operazioni su tali bit-field
- Nella maggior parte degli *ASM-PM*, invece, non è possibile eseguire operazioni direttamente su bit-field
- I dati vengono *memorizzati* nei bit-field e quando devono essere usati in operazioni vengono *estratti*, ovvero copiati in formati interi generali
- Le operazioni di *memorizzazione in bit-field* e di *estrazione da bit-field* possono essere realizzate mediante operazioni di manipolazione di bit

Estrazione da Bit-field

- L'estrazione da un bit-field è particolarmente semplice quando il bit del bit-field che ha posizione minima, è anche il bit di posizione 0 di una parola standard, e la lunghezza del bit-field è minore di quella di tale parola standard
- In questi casi, per estrarre il valore dal bit-field è sufficiente azzerare i bit della parola che non fanno parte del bit-field
- Se il bit del bit-field che ha posizione minima non ha posizione 0 in una parola standard, si può effettuare un'operazione di scorrimento per portarlo in posizione 0 e poi procedere come nel caso precedente

Estrazione da Bit-field

- Esempio: estrazione del bit-field costituito dai bit di posizione compresa tra 0 e 3 del byte S (di 8 bit), nel byte D (di 8 bit)
  - In C (`S` e `D` hanno tipo `unsigned char`):

```
D = S & 0x0F;
```

  - In *ASM* MIPS32:

```
lb      $t0,S
and     $t0,$t0,0x0F
sb      $t0,D
```

  - In *ASM* MC68000:

```
move.b  S,D
and.b   #$0F,D
```

Estrazione da Bit-field

- Esempio: estrazione del bit-field costituito dai bit di posizione compresa tra 2 e 6 del byte S (di 8 bit), nel byte D (di 8 bit)
  - In C (`S` e `D` hanno tipo `unsigned char`):

```
D = ( S >> 2 ) & 0x1F;
```

  - In *ASM* MIPS32:

```
lb      $t0,S
srl     $t0,$t0,2
and     $t0,$t0,0x1F
sb      $t0,D
```

  - In *ASM* MC68000:

```
move.b  S,d0
lsr.b   #2,d0
and.b   #$1F,d0
move.b  D
```

Memorizzazione in Bit-field

- Per memorizzare un valore in un bit-field, tipicamente si procede prima azzerando tutti i bit che compongono il bit-field e poi ponendo a 1 i soli bit necessari
- L'azzeramento viene effettuato tramite *And* bit-a-bit, e la scrittura dei valori 1 tramite *Or* bit-a-bit
- Quando il valore da memorizzare è una variabile, le maschere devono essere generate dinamicamente

Memorizzazione in Bit-field

- Esempio: memorizzazione del valore binario 101 nel bit-field costituito dai bit di posizione compresa tra 11 e 13 della parola D (di 16 bit)
  - In C (`D` ha tipo `unsigned short`):

```
D = D & ~0x3800 | 0x2800;
```

  - In *ASM* MIPS32:

```
lh      $t0,D
and     $t0,$t0,0xC7FF
or      $t0,$t0,0x2800
sh      $t0,D
```

  - In *ASM* MC68000:

```
and.w   #$C7FF,D
or.w    #$2800,D
```

Memorizzazione in Bit-field

- Esempio: memorizzazione del contenuto della parola S (di 16 bit) nel bit-field costituito dai bit di posizione compresa tra 3 e 10 della parola D (di 16 bit)
  - In C (`S` e `D` hanno tipo `unsigned short`):

```
D = D & ~0x07F8 | ( S << 3 );
```

  - In *ASM* MIPS32:

```
lh      $t0,D
and     $t0,$t0,0xF807
lh      $t1,S
sll     $t1,$t1,3
or      $t0,$t0,$t1
sh      $t0,D
```

  - In *ASM* MC68000:

```
and.w   #$F807,D
move.w  S,d0
lsl.w   #3,d0
or.w    d0,D
```

Bit-field in `struct`

- Grazie agli operatori di manipolazione di bit, è possibile utilizzare bit-field in C con le stesse precisione ed efficienza possibili in *ASM*
- Tuttavia, la gestione è scomoda e il rischio di commettere errori è elevato
- Per questo motivo il C permette di definire bit-field quali membri di `struct`
- Le operazioni di estrazione e memorizzazione in tali bit-field avvengono, in modo molto semplice, attraverso l'operatore `.`
- Il compilatore C traduce automaticamente gli accessi in operazioni di manipolazioni di bit

Bit-field in `struct`

- In una `struct` si possono inserire dei membri che sono dei bit-field, dichiarandoli di tipo `int`, `signed int` oppure `unsigned int` e scrivendo, dopo il nome del bit-field, il numero di bit che lo compongono preceduto da `:`
- Esempio di `struct` che contiene un bit-field con segno di 4 bit, uno senza segno di 7 bit e uno di 5 bit per cui è implementation defined l'avere o meno segno

```
struct es_bitfield {
        signed int      bf1:4;
        unsigned int    bf2:7;
        int             bf3:5;
};
```

Bit-field in `struct`

- L'accesso al bit-field `bf2` di una variabile `v` del tipo `struct es_bitfield` definito precedentemente, avviene mediante l'espressione `v.bf2`
- I bit-field definiti in questo modo possono essere operandi in espressioni (il compilatore effettua le opportune conversioni di tipo), come ad esempio in `x = ( v.bf3 + v.bf2 ) / 2`
- Applicare l'operatore di indirizzamento `&` ad un bit-field è una constraint violation, in quanto i bit-field non hanno un indirizzo di memoria

Bit-field in `struct`

- C Standard non definisce tutti i dettagli di come i bit-field sono posizionati all'interno delle `struct`, lasciando ai compilatori la libertà di organizzare i bit delle aree di memoria usate per memorizzare le variabili di un tipo `struct` nel modo più opportuno in relazione alla *ISA* su cui un programma viene eseguito
- L'implementazione può scegliere liberamente le parole all'interno delle quali sistemare un bit-field

Bit-field in `struct`

- Dati due bit-field $F1$ e $F2$ dichiarati in una `struct`, in quest'ordine e senza altre variabili in mezzo
  - Se $F1$ viene memorizzato in una parola D grande abbastanza da contenere anche $F2$, allora anche $F2$ deve essere memorizzato in D, e i due bit-field devono essere adiacenti
  - Altrimenti l'implementazione può scegliere tra due possibilità:
    - memorizzare $F2$ in parte in D e in parte in una diversa parola
    - memorizzare interamente $F2$ in una diversa parola, lasciando inutilizzati alcuni bit di D

Bit-field in `struct`

- È possibile dichiarare dei bit-field senza nome, allo scopo di "distanziare" di un numero preciso di posizioni, il bit-field che precede il bit-field anonimo, da quello che lo segue
- Se, dopo un bit-field $F1$, vengono dichiarati prima un bit-field di 0 bit e poi un bit-field $F2$, allora $F2$ deve essere memorizzato in una parola diversa da quella in cui è memorizzato $F1$
- Per ulteriori dettagli ed esempi sui bit-field in C, si vedano le sezioni 20.1 e 20.2 di **[Ki]** e **[C99]**

Bit-field Come Parole Speciali

- Le versioni più recenti di M68000, a partire da MC68020, hanno formati di dato speciali, chiamati *formati bit-field*, per gestire bit-field di lunghezza compresa tra 1 e 32 bit
- I formati bit-field in MC68020 definiscono sia parole di registro che parole di memoria
- Vi sono *istruzioni di manipolazione di bit-field* per
  - memorizzare un dato in un bit-field
  - estrarre il contenuto di un bit-field
  - azzerare, porre a 1 o invertire tutti i bit di un bit-field
  - confrontare con 0 il contenuto di un bit-field

Bit-field Come Parole Speciali

- Sia F un bit-field, e sia $b_{\max-\mathrm{F}}$ il bit più significativo di F
- Se F è contenuto in un registro, allora viene specificato attraverso un indirizzo generalizzato costituito dal nome del registro che contiene F, dalla posizione di $b_{\max-\mathrm{F}}$ all'interno del registro e dalla lunghezza di F
- Se F è contenuto in memoria, allora viene specificato attraverso un indirizzo generalizzato costituito dall'indirizzo del byte B che contiene $b_{\max-\mathrm{F}}$, dalla posizione di $b_{\max-\mathrm{F}}$ all'interno di B, e dalla lunghezza di F
- Per ulteriori informazioni si veda **[M68000]**

Bit-field Come Parole Speciali

- Le versioni più recenti di MIPS, a partire da MIPS32r2, hanno formati di dato speciali, chiamati *formati bit-field*, per gestire bit-field di lunghezza compresa tra 1 e 32 bit
- I formati bit-field in MIPS32r2 definiscono solo parole di registro
- Vi sono *istruzioni di manipolazione di bit-field* per
  - inserire il contenuto di un bit-field contenuto in un registro, all'interno di un altro registro, anche a partire da una posizione diversa
  - estrarre il contenuto di un bit-field contenuto in un registro, in un altro registro
- In tali istruzioni, un bit-field F viene specificato attraverso un indirizzo generalizzato costituito dal nome del registro che contiene F, dalla posizione nel registro del bit più significativo di F, e dalla lunghezza di F
- Per ulteriori informazioni si veda **[MIPS32]**

Bit-field Come Parole Speciali

- I bit-field possono avere lunghezza minore di quella del formato `byte` (ovvero 8)
- Ciò non contraddice il fatto che i byte siano le parole standard di lunghezza minima, in quanto i bit-field sono parole speciali
- Si osservi che, diversamente dai byte, i bit-field non sono vengono gestiti mediante una singola operazione dalle abstract machine di livello 1: infatti le istruzioni di manipolazione di bit-field, vengono implementate leggendo dalla memoria tutti i byte che contengono i bit del bit-field e poi effettuando a livello 1 le operazioni logiche e di scorrimento che abbiamo descritto in precedenza, per estrarre e manipolare i bif-field
- In altre parole una istruzione *I* di manipolazione di bit-field viene realizzata da una sequenza di istruzioni di livello 1 che effettua le operazioni che, senza disporre di *I*, dovrebbero essere svolte da una sequenza di istruzioni *ASM* o *LM*