Architettura degli elaboratori - lezione 6

Appunti di Davide Vella 2024/2025

Claudio Schifanella
claudio.schifanella@unito.it
Link al moodle :
https://informatica.i-learn.unito.it/course/view.php?id=3106

11/03/2025

Contenuti

Salti condizionati

```c
int v[10], k, i;
while (v[i] == k) {
    ...
    i = i + 1;
}
```

i = x22
k = x24
v = x25

```
while :
slli x23, x22, 2
add x23, x23, x25
lw x26, 0(x23)
bne x26, x24, endwhile
...
addi x22, x22, 1
j while
endwhile :
```

Nota sui salti

Nelle condizioni, è sempre meglio usare la condizione opposta a quella che dobbiamo "tradurre".

✓ **Esempio**

```c
while (v[i] == k) {
...
}
```

Non controlliamo per "v[i] == k", ma controlliamo "v[i] != k"

```
bne x26, x24, endwhile //dove x26 contiene v[i] e x24 k
```

SLT

Istruzione che scrive 1 in rd se $rs1 < rs2$. È di tipo R.

$$\text{slt x21, x19, x20}$$

Si può pensare di usare dopo slt l'istruzione "beq rs1, x0, L1" per saltare ad un certo punto se rs1 e minore di rs2.

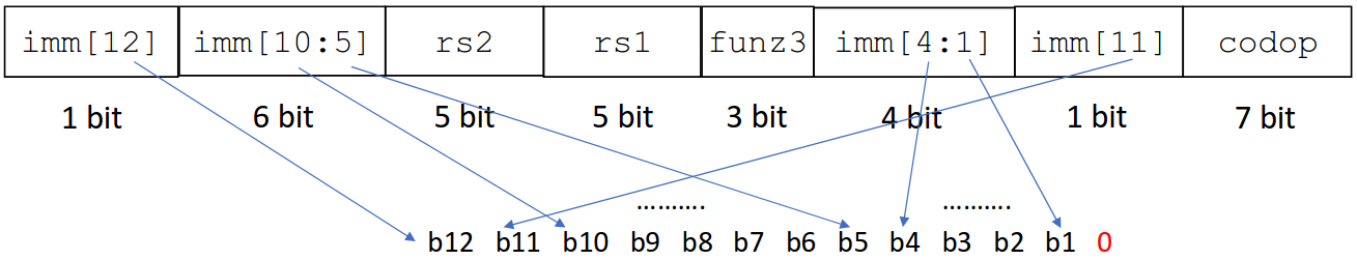
Esempio

```
if (i < j)
    k = 1;
else
    k = 0;
```

i = x19
j = x20
k = x21

```
slt x21, x19, x20 //salviamo in k il risulato di i < j
beq x21, x0, L1 //saltiamo a L1 se x21 == 0, dove x21 = 0 quando rs1 > rs2
```

Formato SB



- codop : codice operazione
- imm[11] : contiene una parte dell'offset
- imm[4:1] : (dal 1 al 4) contiene una parte dell'offset
- funz3 : differenzia le varie istruzioni di salto
- rs1/rs2 : registri di input
- imm[10:5] : (dal 5 al 10) contiene una parte dell'offset
- imm[12] : contiene una parte dell'offset
  Il formato può rappresentare indirizzi di salto da -4096 a +4094, in multipli di due.

Moltiplicazione

Per poter moltiplicare il valore di due registri si può usare (in alcuni casi, non sempre è implementata) l'istruzione "mul".
In C :

$$a = b * c$$

In assembly :

$$\text{mul x10, x12, x12}$$

Questo potrebbe essere un problema in caso in cui la il risultato della moltiplicazione in overflow per un registro. Per questo si può usare "mulh", la quale usa 2 registri.

Divisione

Uguale per sopra, si può usare l'istruzione "div" per dividere il valore di due registri.

Procedure (funzioni)

Porzioni di codice associate ad un nome che possono essere invocate più volte e che eseguono un compito specifico, avendo come input una lista di parametri e come out un valore di ritorno.

Pro

- Astrazione
- Riusabilità del codice
- ..
- ..

Esempio C

Programma (procedura) (chiamante) :

```
f = f + 1;
risultato = somma(f,g);
```

Procedura somma() (chiamata) :

```
int somma(int x, int y) {
    int rst;
    rst = x + y;
    return rst;
}
```

Si può pensare di implementare la funzione in assembly con dei jump e delle etichette. Questo non sarebbe sbagliato, però dobbiamo salvarci anche il punto di ritorno (return rst e deve diventare un jump a chiamante).

Passi da seguire

Chiamante :

- Mettere i parametri in un luogo accessibile alla procedura
- Trasferire il controllo alla procedura
  Chiamato :
- Acquisire le risorse necessarie per l'esecuzione della procedura
- Eseguire il compito richiesto
- Mettere il risultato in un luogo accessibile al programma chiamante
- Restituire il controllo al punto di origine

Istruzione di invocazione : jal

Per saltare all'etichetta usiamo "jal" :

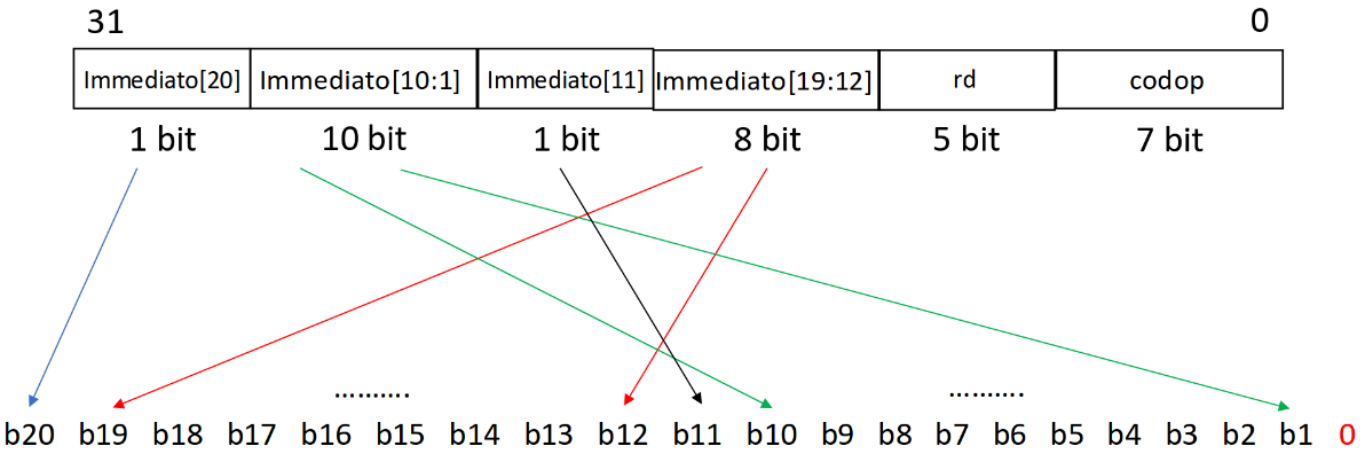
$$\text{jal IndirizzoProcedura}$$

In realtà "jal" è una pseudo istruzione, la vera istruzione è :

$$\text{jal x1, IndirizzoProcedura}$$

Dove x1 è il registro dove salviamo l'indirizzo della istruzione a cui saltare quando finisce la procedure. Si usa sempre x1, nessun altra.

Formato J



- codop :
- rd :
- imm[19:12]
- imm[11]
- imm[10:1]

- imm[20]

Istruzione ritorno al chiamante : jalr

Salta ad un indirizzo qualsiasi :

$$\text{jalr rd, offset(rs1)}$$

Per tornare indietro quindi facciamo :

$$\text{jalr x0, 0(x1)}$$

Problemi

1. Si deve fare attenzione, i registri sono condivisi, non c'é spazio infinito. Può succedere che il registro x5 contiene un valore usato dal chiamanate per qualcosa, ma magari il chimato usa x5 per mettere il risultato. Come si può rimediare al problema? Possiamo salvare il valore di x5 in memoria (con l'istruzione sw) prima di utilizzarlo e poi lo ripristiamo alla fine (sw/lw sono istruzioni oerose, quindi è meglio evitarle).
2. Se chiamiamo una procedura dentro una procedura, andiamo a sovrascrivere x1, quindi non sappiamo più dove tornare. Come possiamo rimediare al problema? Di nuovo, possiamo salvare il valore di x1 prima di chiamare la procedura e dopo che la procedura finisce, possiamo caricare il valore che abbiamo salvato prima.
3. Può succedere che i parametri e le variabili di una procedura superano il numero di registri disponibili. Come facciamo? Salviamo di nuovo in memoria.

Stack

Area di memoria dinamica. Per accedere si usa il "Last In First Out"

Operazioni

- PUSH : aggiunge un elemento in cima allo stack.
- POP rimuove un elemento dalla cima dello stack.
- SP : la cime delle stack è identificato dallo stack pointer.

Convenzione RISC-V

- Grow-down : lo stack cresce da indirizzi di memoria alti verso indirizzi di memoria bassi
- last-full : lo stack pointer (SP) contiene l'indirizzo dell'ultima cella di memoria occupata nello stack,
- Il valore di SP è salvato nel registro x2 (o sp).

Push

```
addi sp, sp, -4 // -4 perché vogliamo salvare una word, sennò -2 se una hw...
sw x20, 0(sp)
```

Pop

```
lw x20, 0(sp)
addi sp, sp, +4 // -4 perché vogliamo salvare una word, sennò -2 se una hw...
```

Esempio

```
int somma(int x, int y) {
int rst;
rst = x + y + 2;
return rst;
}
….
f=f+1
risultato=somma(f,g)
…
```

x = x10
y = x11
rst = x20
f = x6

```
somma :
addi sp, sp, -8 //allochiamo due interi nello stack
sw x5, 0(sp)
sw x20, 4(sp) //salviamo due interi nella memoria allocata prima
add x5, x10, x11
addi x20, x5, 2
addi x10, x20, 0
lw x5, 0(sp)
lw x20, 4(sp)
addi sp, sp, 8 //liberiamo la memoria
jalr x0,0(x1)
...

...
addi x6, x6, 1
```

```
...
jal somma
```

Convenzione sui registri

Registro	Nome	Utilizzo
x0	zero	Costante zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Puntatore a thread
x8	s0 / fp	Frame pointer (il contenuto va preservato se utilizzato dalla procedura chiamata)
x10-x11	a0-a1	Passaggio di parametri nelle procedure e valori di ritorno
x12-x17	a2-a7	Passaggio di parametri nelle procedure
x5-x7 x28-x31	t0-t2 t3-t6	Registri temporanei, non salvati in caso di chiamata
x9	s1	Registri da salvare: il contenuto va preservato se utilizzati dalla procedura chiamata
x18-x27	s2-s11	

Chi è responsabile di salvare i registri quando si effettuano chiamate di funzioni?

- La funzione chiamante conosce quali registri sono importanti per sé e che dovrebbero essere salvati
- La funzione chiamata conosce quali registri userà e che dovrebbero essere salvati prima di modificarli
  Bisogna evitare le inefficienze → Minimo salvataggio dei registri:
- La funzione chiamante potrebbe salvare tutti i registri che sono importanti per sé, anche se la procedura chiamata non li modificherà
- La funzione chiamata potrebbe salvare tutti i registri che si appresta a modificare, anche quelli che non verranno poi utilizzati dalla procedura chiamante una volta che la procedura chiamata le avrà restituito il controllo

Domande

Perché usando le istruzioni di formato SB possiamo saltare solo di numeri pari? Perché tanto le istruzioni occupano 32 bit, quindi se vogliamo saltare (per esempio) 2 istruzioni, vogliamo saltare sicuramente un numero pari.
Perchè possiamo saltare di 2 in 2 nelle istruzioni formato SB e non di 4 in 4 (visto che le

istruzioni occupano 32 bit = 4 byte)? Perché, anche se non le vedremo, l'architettura prevede di supportare anche istruzioni da 16 bit