SOLUZIONE ESEMPIO SECONDO COMPITINO 2011

ESERCIZIO 1:

Quante volte la CPU deve accedere alla memoria quando **preleva ed esegue** un'istruzione che ha due operandi, uno con modo di indirizzamento diretto e uno con modo di indirizzamento indiretto ?

a) 2

b) 3

c) 1

d) 4

f) nessuna delle risposte precedenti è corretta;

Soluzione

Ho 2 operandi:

- primo operando: indirizzamento diretto ==> Sull'istruzione vi è l'indirizzo della cella di Memoria su cui vi è l'operando ==> 1 accesso alla memoria
- secondo operando: indirizzamento indiretto ==> Sull'istruzione vi è l'indirizzo della cella di Memoria su cui vi è scritto l'indirizzo della cella di Memoria che contiene l'operando ==> 2 accessi alla memoria

Globalmente ho quindi 3 accessi, quindi risposta B.

ESERCIZIO 2:

Si consideri la rappresentazione di numeri a virgola mobile che utilizza 4 bit per il campo esponente e 11 bit per la mantissa. Il numero -41,125 viene rappresentato dalla sequenza di bit:

a) 1110001001001000

b) 1010101001001000

c) 1010010010001100

d) 1010010010000101

e) nessuna delle risposte precedenti è corretta;

Soluzione

Rappresentazione in virgola mobile con:

- 4 bit per esponente, quindi:
  - *esponente* = esponente_polarizzato - (2^{4-1}-1) = *esponente_polarizzato -7*
  - *esponente_polarizzato = esponente + 7*
- 11 bit per la mantissa (+ 1 bit (implicito) del primo 1)
- 1 bit (implicito) per il segno

-41.125 in base decimale = - (32+8+1+1/8) = -(2^5+2^3+2^0+2^{-3}) = -101001.001 in base due

-101001.001 = -1.01001001 * 2^5 = in forma polarizzata: -1.01001001 * 2^{5+7} = -1.01001001 * 2^{12}
12 in base decimale = 8+4 = 2^3+2^2 = 1100 in base due
segno - = 1 in virgola mobile

esponente polarizzato
segno mantissa
-41.125 in base decimale = 1 1100 01001001000 in virgola mobile

quindi risposta A

ESERCIZIO 3:

Si consideri una pipeline a 4 stadi: fetch (IF), decodifica (ID), elaborazione (EI), e scrittura dei risultati (WO), per cui:

- i salti incondizionati sono risolti (identificazione salto e calcolo indirizzo target) alla fine del secondo stadio (ID);
- i salti condizionati sono risolti (identificazione salto, calcolo indirizzo target e calcolo condizione) alla fine del terzo stadio (EI);
- il primo stadio (IF) indipendente dagli altri;

inoltre si assuma che non ci siano altre istruzioni che possano mandare in stallo la pipeline e che non sia implementato alcun meccanismo di trattamento dei salti.

Sapendo che:

- il 10% delle istruzioni sono di salto condizionale
- il 3% delle istruzioni sono di salto incondizionale
- il 55% delle istruzioni di salto condizionale hanno la condizione soddisfatta (prese)

Il fattore di velocizzazione della pipeline è di:

a) 3,230988

b) 3,508772

c) 2,356502

d) 3,960031

e) nessuna delle risposte precedenti è corretta;

Soluzione
Percentuali:

- 3% = 0.03 salto incondizionale
- 10% = 0.10 salto condizionale, di cui:
  - 55% = 0.55 preso ==> quindi il 0.10 * 0.55 = 0.055 delle istruzioni totali
  - 45% = 0.45 non preso ==> quindi il 0.10 * 0.45 = 0.045 delle istruzioni totali

Nessun trattamento dei salti. Verrà quindi processata l'istruzione successiva.

Legenda:
~~X~~ = stadio di una istruzione errata che verrà eliminata con lo svuotamento della pipeline

Salti Incodizionati:

	1	2	3	4
Istruzione jump	IF	ID	EI	WO
Istruzione successiva		~~IF~~	IF	EI

Ho 1 ciclo di stallo: non ho caricato l'istruzione corretta.

Salti Codizionati Presi:

	1	2	3	4
Istruzione branch	IF	ID	EI	WO
Istruzione successiva		~~IF~~	~~ID~~	IF

Ho 2 cicli di stallo: ho caricato l'istruzione errata, ma per scoprirlo devo aspettare la fine di EI.

Salti Condizionati Non Presi:

	1	2	3	4
Istruzione branch	IF	ID	EI	WO
Istruzione successiva		IF	ID	EI

Non ho nessun ciclo di stallo in quanto è stata caricata l'istruzione corretta.

Percentuali con cicli di stallo:

- salto incodizionato: 0.03*1 =0.03
- salto condizionato preso: 0.055*2 = 0.11
- salto condizionato non preso: 0.045*0=0

quindi la *frazione di cicli di stallo* = 0.03+0.11+0 = 0.14
Essendovi 4 stadi → k = 4

$$fattore\ di\ velocizzazione = S_k = \frac{1}{1+frazione\ cicli\ stallo} k = \frac{1}{1+0.14} 4 = 3{,}5087719$$

Quindi risposta B

ESERCIZIO 4:

Si spieghi in dettaglio la codifica in complemento a 2 dei numeri interi. Si discutano poi i problemi legati alla realizzazione della moltiplicazione di due interi rappresentati in complemento a 2, esemplificando tali problemi su un caso concreto di moltiplicazione.

Soluzione:
Attraverso la rappresentazione in complemento a due, avendo n bit a disposizione possiamo rappresentare tutti i numeri interi da -2^{n-1} a $+2^{n-1}-1$.
Il segno del numero viene indicato mediate il bit più a sinistra, il quale rappresenta - se è a 1 e rappresenta + se è a 0.
I numeri positivi vengono rappresentati codificando in binario il numero con anteposto lo 0 del segno (cioè la stessa dei numeri positivi per modulo e segno). Dentro i numeri positivi compaiono i numero dallo 0, che sarà quindi + 0, a $+2^{n-1}-1$.
I numeri negativi vengono rappresentati a partire da -1 il quale viene rappresentato da n uni. Dalla rappresentazione di -1 vi è uno spostamento all'indietro, cioè alla rappresentazione di -1 viene sottratto un uno per ogni passo indietro rispetto ad esso. Il numero negativo con modulo più grande

sarà quindi composto da un 1 seguito da n-1 zeri. Dentro i numeri negativi compaiono i numeri da -1 a -2^{n-1}. In pratica per avere la rappresentazione di un numero negativo -k vengono utilizzati due metodi:

- viene calcolato k in binario, viene complementato ed al complemento viene sommato 1
- viene calcolato k in binario, vengono scritti gli stessi bit da destra a sinistra sino al primo 1, i numeri a sinistra di tale 1 vengono complementati.

Nella moltiplicazione di due interi rappresentati in complemento a 2 nasco problemi legati al bit più significativo, cioè al bit di segno. Se nella moltiplicazione di due numeri è presente almeno un numero negativo, il suo bit di segno verrà calcolato nella moltiplicazione, ed andrà a rendere errato il risultato.
Esempio:
-5 (1011) x 3(0011) utilizzando i prodotti parziali otteniamo:

$$\begin{array}{r} 1011 \\ \times 0011 \\ \hline 1011 \\ 1011- \\ \hline 100001 \end{array}$$

convertendo in decimale 100001 da complemento a 2 otteniamo: -31
Per risolvere tali problemi bisogna utilizzare la rappresentazione in complemento a due per i prodotti parziali. In pratica la soluzione adottata è l'algoritmo di Booth.

ESERCIZIO 5:

Si descrivano i possibili formati di codifica di una istruzione, specificando per ogni formato la sua composizione tipica, i pregi e i difetti.

Soluzione
Il formato delle istruzioni descrive i campi dell'istruzione, la sua lunghezza ed numero di indirizzi. In qualsiasi formato è incluso il codice operativo, che discrimina quale operazione fare, ed zero, uno o più operandi in modo implicito o esplicito.
--- introduzione generale ---
Il formato delle istruzioni può essere a lunghezza:

- fissa, tutte le istruzioni hanno la stessa lunghezza, ma posso avere più formati cambiando i campi. Questo formato è estremamente efficiente nell'uso della pipeline.
- variabile, ogni istruzione ha una lunghezza che dipende dal numero di operandi e nel campo con l'opcode devo anche specificare il numero di operandi. Permettono una grande flessibilità, ma incrementano notevolmente la complessità.
- ibrida, ho diversi formati con lunghezza fissa ma diversa

La lunghezza è data da un compromesso tra repertorio di istruzioni potente e necessità di risparmiare spazio ed è condizionata da diversi fattori: dimensione memoria (deve poter essere completamente indirizzata), organizzazione della memoria, struttura del bus (in base a quanto è capiente posso discriminare il numero di accessi), complessità della CPU (CPU con più istruzioni necessita di più bit per l'opcode) e la velocità richiesta della CPU(se la CPU utilizza la pipeline). L'allocazione dei bit nei campi d'indirizzo dipende da: numero dei metodi d'indirizzamento, numero di operandi, numero di registri (più sono, più vengono utilizzati, minore sarà il numero di bit utilizzati per il più costoso indirizzamento a memoria), numero di banchi di registro (solo alcune architetture li posseggono), intervallo di indirizzi da rendere disponibile, granularità d'indirizzamento (byte o parola, a volte può essere utile il byte anche se più costoso nel numero di accessi).

Prima di inoltrasi nella descrizione dei formati è utile precisare degli svantaggi e vantaggi generali:

- Una grande varietà di istruzioni e di lunghezze differenti permettono l'utilizzo di un vasto numero di opcode e metodi di indirizzamento, permettendo una grande flessibilità. Questo tipo di istruzioni può essere realizzato utilizzando istruzioni a lunghezza variabile, ma il loro

utilizzo complica notevolmente la CPU(rendendola anche più costosa), rende difficile il fetch e globalmente diminuisce la reattività sulle operazione più frequentemente utilizzate. Tale caratteristiche si avvicinano alla filosofia CISC.

- Un numero fisso di formati ha come problemi la mancanza di ortogonalità(cioè operandi indipendenti dal codice operativo) tra opcode e metodo d'indirizzamento ed un numero limitato di operandi utilizzabili per operazione. In generale vi è una mancanza di flessibilità. La CPU è però più semplice da realizzare, generalmente più, permette il caricamento dell'istruzione in modo uniforme veloce ed una dimensione fissa delle istruzioni favorisce l'uso della pipeline. Tale caratteristiche si avvicinano alla filosofia RISC.

I possibili formati di codifica di una istruzione sono:

- PDP-8: architettura molto vecchia in cui era disponibile un solo registro, l'accumulatore. Nonostante le forti limitazioni vi è un indirizzamento abbastanza flessibile. Le istruzioni erano a lunghezza fissa, con molti formati, che permettevano un estensione degli opcode grazie all'utilizzo di microperazioni.
  Un formato ottimizzato al massimo per quel tempo.
- PDP-10: innovativo rispetto al PDP-8: un unico formato a lunghezza fissa, le stesse tipologie di operazioni per diversi tipi di operandi, solo indirizzamento diretto ed introdotto il concetto di ortogonalità.
  Facilitava il lavoro dei programmatori e dei compilatori ma non utilizzava in modo efficiente lo spazio a disposizione. (scelta progettuale)
- PDP-11: adotta una formato ibrido in cui ogni operando può utilizzare un qualsiasi tipo di indirizzamento. Ha un'ampia variabilità di formati e metodi di indirizzamento che la rendono costosa e complicata da realizzare, ma i programmi risultano essere efficienti e compatti.
- VAX: un sistema estremamente variabile con lunghezza varabile e formati diversi, in quanto l'opcode può stare su un byte o su due. E' un sistema molto flessibile e potente che facilita il lavoro di programmatori e compilatori, ma il sistema è molto complesso.
- PENTIUM: le istruzioni sono composte da vari pezzi i quali vengono assemblati in base alle necessità. Il risultato sono quindi istruzioni a lunghezza variabile e di vari formati che richiedono una complessa decodifica. Tale approccio è stato utilizzato per mantenere la retro compatibilità.
- PowerPC: un sistema con istruzioni a lunghezza fissa e diversi formati, avendo una suddivisione non omogenea dei campi. Pensato per computer RISC.

ESERCIZIO 6:

Nel contesto di una pipeline descrivere la problematica della dipendenza dal controllo e si discuta in particolare la tecnica del buffer circolare, spiegando in quali situazioni tale tecnica è particolarmente efficace.

Soluzione

La dipendenza da controllo si verifica quando vi sono istruzioni che causano una violazione di sequenzialità la quali invalidano il principio di pipeling sequenziale.
Le istruzioni che causano queste dipendenze sono tutte le istruzioni che modificano il PC (cioè salti incodizionati e condizionati, chiamate e ritorni a procedure, interruzioni) ed invalidano la pipeline in quanto la fase fetch successiva carica l'istruzione seguente, che non può essere quella giusta.
Vi sono due macro soluzioni:

- mettere in stallo la pipeline introducendo nop (fasi non operative) fino a quando non si è calcolato l'indirizzo della prossima istruzione. Questo metodo ha una pessima efficienza, ma è semplice da realizzare

- individuare le istruzione critiche così da poter anticipare l'esecuzione ed eseguirla in modo ottimale. Questo metodo richiede una compilazione complessa o una logica di controllo dedicata. Nel caso particolare dei salti condizionati, le soluzioni di questo tipo sono: flussi multipli, prefeth della destinazione, buffer circolare, predizione del salto -queste 4 sono soluzione hardware - , salto ritardato -soluzione software (uso il compilatore) -

In particolare, il buffer circolare è composto da una piccola e molto veloce memoria, detta buffer circlare, dove vengono mantenute le ultime n istruzioni prelevate. In caso di salto si controlla se l'istruzione di destinazione è già presente nel buffer, così da evitare il fetch della stessa.
Il riconoscimento delle istruzioni presenti nel buffer avviene in modo simile a quanto succede in cache, cioè i bit meno significativi vengono utilizzati per l'indirizzamento all'interno del buffer, mentre quelli più significativi vengono comparati per determinare se è avvenuto un hit.
I vantaggi dati dal buffer sono:

- è possibile anticipare il fetch di alcune istruzioni successive a quella corrente portandole all'interno del buffer. Applicando tale tecnica si possono verificare due eventi favorevoli:
  - se non vi è salto non è necessario caricare le istuzioni successive dalla memoria
  - se si salta in avanti di poche istruzioni l'istruzione sarà probabilmente già contenuta nel buffer
- se il salto condizionato realizza un ciclo, le cui istruzioni possono essere tutte contenute nel buffer, evitando quindi di effettuare fetch ripetuti delle stesse istruzioni

ESERCIZIO 7:

Spiegare in che modo un compilatore possa aiutare l'utilizzo efficace dei registri da parte di una architettura RISC.

Soluzione:
Nell'architettura RISC l'ottimizzazione dell'utilizzo dei registri viene lasciata al compilatore.
L'obbiettivo del compilatore è mantenere gli operandi necessari nei registri per il maggior tempo possibile e di minimizzare il numero di accessi alla memoria.
Le operazioni eseguite dal compilatore sono influenzate dal fatto che i linguaggi ad alto livello non fanno riferimento esplicito ai registri (ad eccezione del C) e lavorano con le variabili.
Il compilatore esegue un'approfondita analisi statica del programma sottopostogli e, supponendo che vi siano a disposizione un numero illimitato di registri, assegna un registro simbolico (o virtuale) ad ogni variabile. Mappa poi tali registri virtuali sui registri reali del processore.
I registri simbolici possono essere mappati sullo stesso registro reale, cioè lo possono condividere, se il loro uso non si sovrappone temporalmente.
Se i registri reali non sono sufficienti per contenere tutte le variabili riferite in un dato intervallo di tempo, alcune di queste vengono mantenute in memoria principale.
Il mapping delle variabili è equivalente alla risoluzione del problema di colorazione di un grafo.
Tale problema è "difficile" da risolvere, cioè sono noti soltanto algoritmi con complessità esponenziale in grado di farlo. Nel problema colorazione di un grafo dato un grafo costituito da nodi connessi da archi, si vuole assegnare un colore per ogni nodo in modo tale che:

- nodi adiacenti connessi da archi abbiano colori diversi
- si usi il minore numero possibile di colori.

Riportando tale problema al mapping, i nodi sono i registri virtuali, gli archi rappresentano variabili riferite nello stesso intervallo di tempo, i colori rappresentano i registri reali.
Essendovi a disposizione n registri reali, il grafo dovrà essere colorato con n colori. Se vi sono nodi che non potranno essere colorati, allora verranno copiati in memoria.

ESERCIZIO 8:

Sia data la seguente sequenza di istruzioni assembler, dove i dati immediati sono espressi in esadecimale

```
SW   $8, 150($0)
ADD  $2, $0, $8
LW   $4, 10($2)
ADDI $4, $4, 3
ADDI $2, $4, 4
SW   $4, 208($2)
ADD  $5, $4, $2
```

Si consideri la pipeline MIPS a 5 stadi vista a lezione, con possibilità di data-forwarding e con possibilità di scrittura e successiva lettura dei registri in uno stesso ciclo di clock:

a) si individuino e discutano le dipendenze dovute ai dati;

b) mostrare come evolve la pipeline durante l'esecuzione del codice

Soluzione:

Riscrivo la consegna chiarificando le istruzioni macchina:

SW	$8, 150($0)	mem[150 + [R0]] ← R8
ADD	$2, $0, $8	R2 ← [R0] + [R8]
LW	$4, 10($2)	R4 ← mem[10 + [R2]]
ADDI	$4, $4, 3	R4 ← [R4] + 3
ADDI	$2, $4, 4	R2 ← [R4] + 4
SW	$4, 208($2)	mem[208 + [R2]] ← R4
ADD	$5, $4, $2	R5 ← [R4] + [R2]

Dipendenze:

DIPENDENZA	Dipendenza dati delle fasi in una pipeline (generale) con dataforward Dipendenza dati delle fasi in una MIPS senza dataforward
R2 in LW $4, 10($2) dipende da ADD $2, $0, $8	input EX_{LW} ha bisogno di output EX_{ADD} input ID_{LW} ha bisogno del dato aggiornato da WB_{ADD} (aggiornamento e lettura del dato permesso nello stesso ciclo di clock - VERO SEMPRE! Non riscrivo sotto)
R4 in ADDI $4, $4, 3 dipende da LW $4, 10($2)	input EX_{ADDI} ha bisogno di output MEM_{LW} input ID_{ADDI} ha bisogno del dato aggiornato da WB_{LW}
R4 in ADDI $2, $4, 4 dipende da ADDI $4, $4, 3 (R4 in ADDI $2, $4, 4 dipende anche da LW $4, 10($2), ma R2 è già modificato da ADDI $4, $4, 3 - ciò crea una dipendenza WriteAfterWrite)	input EX_{ADDI} ha bisogno di output EX_{ADD} input ID_{ADDI} ha bisogno del dato aggiornato da WB_{ADDI}
R4 in SW $4, 208($2) dipende da ADDI $4, $4, 3 (R4 in SW $4, 208($2) dipende anche da LW $4, 10($2), ma R2 è già modificato da ADDI $4, $4, 3 - ciò crea una dipendenza WAW)	input MEM_{SW} ha bisogno di output EX_{ADDI} (questo bypass non c'è nelle MIPS) input ID_{SW} ha bisogno del dato aggiornato da WB_{ADDI}
R2 in SW $4, 208($2)	input EX_{SW} ha bisogno di output EX_{ADDI}

dipende da ADDI $2, $4, 4 (R2 in SW $4, 208($2) dipende anche da ADD $2, $0, $8, ma R2 è già modificato da ADDI $2, $4, 4 - ciò crea una dipendenza WAW)	input ID_{SW} ha bisogno del dato aggiornato da WB_{ADDI}
R2 in ADD $5, $4, $2 dipende da ADDI $2, $4, 4 (R2 in ADD $5, $4, $2 dipende anche da ADD $2, $0, $8, ma R2 è già modificato da ADDI $2, $4, 4 - ciò crea una dipendenza WAW)	input EX_{ADD} ha bisogno di output EX_{ADDI} input ID_{ADD} ha bisogno del dato aggiornato da WB_{ADDI}
R4 in ADD $5, $4, $2 dipende da ADDI $4, $4, 3 (R4 in ADD $5, $4, $2 dipende anche da LW $4, 10($2), ma R4 è già modificato da ADDI $4, $4, 3 - ciò crea una dipendenza WAW)	input EX_{ADD} ha bisogno di output EX_{ADDI} input ID_{ADD} ha bisogno del dato aggiornato da WB_{ADDI}

--- parentesi Teorico/Pratica ---
I forward possibili nella MIPS sono solo:

- da EX → a EX
- da MEM → a EX

Vi è inoltre la possibilità di scrittura e successiva lettura dei registri in uno stesso ciclo di clock e ciò permette il passaggio di dati da WB a ID entro lo stesso ciclo.

		1	2	3	4	5	6	7	8	9	10	11	12	13	14
SW	$8, 150($0)	IF	ID	EX	MEM	WB									
ADD	$2, $0, $8		IF	ID	EX	MEM	WB								
LW	$4, 10($2)			IF	ID	EX	MEM	WB							
ADDI	$4, $4, 3				IF	ID	ID	EX	MEM	WB					
ADDI	$2, $4, 4					IF	IF	ID	EX	MEM	WB				
SW	$4, 208($2)							IF	ID	ID	ID	EX	MEM	WB	
ADD	$5, $4, $2								IF	IF	IF	ID	EX	MEM	WB

by Caesar

NON HO LA PRESUZIONE DI DIRE CHE QUESTO E' IL CORRETTO MODO DI SVOLGERE IL COMPITINO.
LO CARICO PERCHE' POTREBBE ESSERE UTILE AVERE UN CONFRONTO O MAGARI UNO SPUNTO PER RISOLVERE ALCUNI ESERCIZI.
SE VEDETE ERRORI AGGIORNATE PURE IL FILE CON UNA NUOVA VERSIONE CORRETTA E MIGLIORATA.
ABBIATE SPIRITO CRITICO =)

SALUTI

Caesar