UNIVERSITÀ di VERONA

Dipartimento di INFORMATICA

Prof. Franco Fummi
Prof. Luca Geretti

ARCHITETTURA DEGLI ELABORATORI

Eserciziario di Microistruzioni, Memorie, Pipeline e Teoria

Creato da

Enrico Giordano

Serena Cavaletti

Katia Cracco

Roxana Belciug

Indice

Microistruzioni

Memorie Ram, Cache e Virtuali

Pipeline

Domande di teoria

Microistruzioni: regole ed accorgimenti

Per svolgere questi esercizi, è necessario innanzitutto sapere come funzionano le istruzioni Assembly Intel (in sintassi AT&T). Ciò è importante perché non bisogna imparare a memoria tutte le microistruzioni, ma bisogna ragionare per "istruire" il nostro microprocessore. Un buon esperimento consiste nel creare dei piccoli programmi in Assembly e vedere, tramite il Debugger, quali registri vengono modificati e come viene impostato in particolare il PC, il registro puntatore di Stack ESP e EBP. Capire bene le microistruzioni significa evitare i Segmentation Fault nella creazione di programmi in Assembly; spesso però, dalle Segmentation Fault si imparano le microistruzioni.

Bisogna tener presente che un solo valore per volta può passare nel bus; per sicurezza, ricordarsi sempre di controllare che non si stiano manipolando due valori differenti in un'unica microistruzione. Ad esempio, se devo utilizzare il valore del registro EAX e lo devo sommare al valore di EBX, sicuramente uno dei due deve essere salvato in un registro ausiliario della ALU (in alcune architetture di riferimento si chiama Y, in altre V).

Particolare attenzione verso queste scritture (indirizzamenti):

- %EAX *Indirizzamento diretto a registro:* il valore viene prelevato dal registro;

- (%EAX) *Indirizzamento indiretto a registro:* il valore si trova nell'indirizzo di memoria puntato da eax, quindi occorre decodificarlo inviando il valore del registro EAX nel registro MAR (Memory Address Register), aspettare la lettura in memoria (WMFC), prelevare il valore ottenuto tramite MDR (Memory Data Register).

- 4(%EAX) Come per l'indirizzamento indiretto, con l'aggiunta che il valore ottenuto da MDR deve essere sommato al valore prima della parentesi (in questo caso 4).
- $4(%EAX) *Indirizzamento indiretto con spiazzamento:* come per l'istruzione sopra, con l'aggiunta che, dopo la somma di 4, deve essere aggiunto lo spiazzamento.

Bisogna ricordare infine di dividere la fase di Fetch, che in tutte le microistruzioni è uguale, la fase di Decode, che in alcuni casi non è presente, e la fase di Execute.

Di seguito vengono riportate alcune istruzioni Assembly Intel 80X86 e la relativa sequenza di passi.

MOV src, dst

Il valore di "src" viene spostato in "dest", bisogna quindi prelevare il valore del primo registro "src" e caricarlo nel registro "dst". Questa istruzione consente l'inizializzazione di un registro o di un'area di memoria. Accetta inoltre i modificatori l,w e b per indicare la dimensione dell'operatore "src".

ADD src, dest

Viene sommato il valore di "src" a "dest" e il valore ottenuto viene caricato in "dest"; bisogna perciò prelevare il valore del primo registro "src" e caricarlo nel registro ausiliario della ALU, poi prelevare il valore di "dest" e direttamente sommarlo al valore, precedentemente caricato nel registro ausiliario, tramite il comando ADD (segnale di controllo della ALU), per poi salvarlo nel registro di uscita della ALU (chiamato Z). Infine il valore in Z deve essere inviato in "dest".

SUB src, dest

Questa non è molto differente dalla ADD; semplicemente bisogna fare il complemento a 2 di "src" ed eseguire le medesime microistruzioni di ADD.

J value

Questa indica che deve essere eseguito un salto verso "value" (può essere un'etichetta, un valore di registro, ecc...) in base alla situazione descritta dopo J:

JZ: Jump if Zero

JE: Jump if Equal

JNE: Jump if Not Equal

JL : Jump if Less

JLE: Jump if less or equal

JG: Jump if Great

JGE: Jump if greater or equal

Si porta un'eccezione per JMP, ovvero salto incondizionato, in cui si salta direttamente senza controlli. Prima di tutto, oltre alla fase di Fetch, bisogna controllare se il valore ottenuto da una precedente CMP (Compare) soddisfa la condizione; se non la soddisfa, la microistruzione finisce subito, altrimenti bisogna eseguire il salto. Deve perciò essere decodificato il valore "value" che può provenire da diversi luoghi, in base alla situazione, (se è un'etichetta, proviene dall'OFFSET, se è un registro proviene da esso, se è un indirizzamento vedere sopra).
Una volta ottenuto il valore deve essere sommato al valore di PC e dopo essere stato sommato, deve essere inserito nel PC.

N.B. Si fa un'eccezione per i salti assoluti, in quanto il valore "value" deve essere inserito direttamente in PC senza somma.

PUSH value

L'operazione di Push consiste nell'inserire un valore "value" nella cima dello Stack; occorre perciò decrementare il registro ESP di 4 per puntare ad una cella vuota dello Stack (superiore a quella attuale), caricare il nuovo valore di ESP sia in ESP sia in MAR, prendere il valore "value" e caricarlo nel registro MDR e dare il comando WRITE, in modo da scrivere il valore nella cella prescelta. Dopo WMFC l'istruzione termina. Può essere seguito solo dalle lettere "w" (16 bit) o "l" (32 bit). Il registro ESP viene decrementato di tanti byte quanti ne vengono messi sullo stack (4 con "l" e 2 con "w")

POP dest

L'operazione di Pop consiste del prelevare il valore della cima dello Stack e caricarlo nel luogo "dest"; è l'esatto opposto della Push. Deve essere incrementato di 4 il valore di ESP in modo da puntare ad una cella sottostante (quindi non vuota) dello Stack, salvare il nuovo valore di ESP in ESP e caricarlo in MAR, poi si dà il comando di lettura READ e, dopo WMFC, si preleva il valore ottenuto da MDR e si carica nella destinazione "dest".

Può essere seguito solo dalle lettere “w” (16 bit) o “l” (32 bit). Il registro ESP viene incrementato di tanti byte quanti ne vengono messi sullo stack (4 con “l” e 2 con “w”)

N.B. Il recupero di informazioni dello stack mediante più chiamate dell’istruzione POP deve avvenire nell’ordine inverso del loro inserimento nello stack mediante le istruzioni PUSH.

CALL value e RET

Call indica una chiamata a funzione; la funzione è identificata da “value”, che può essere un indirizzo, un'etichetta, un indirizzamento diretto. Si può dire che prima viene eseguita un'operazione di Push del valore di PC; infatti viene caricato il valore incrementato di 4 di ESP in MAR, poi viene caricato il valore di PC in MDR e si dà il comando di scrittura. Questo serve perché si salva in una cella (possibilmente vuota) dello Stack il valore di ritorno della funzione, cioè il valore del PC (per così dire serve al microprocessore per ricordarsi cosa stava facendo prima di iniziare ad eseguire la funzione chiamata), per poi riottenerla con l'istruzione finale della funzione chiamata RET.

La RET estra dalla cima dello stack un valore, lo interpreta come indirizzo di un’istruzione e fa saltare il processore a tale istruzione (il valore sullo stack corrisponde all’indirizzo dell’istruzione successiva all’istruzione CALL). In poche parole esegue l'esatto opposto di CALL, cioè preleva dallo Stack il valore di PC.

N.B. Al momento dell’esecuzione della RET in cima allo stack deve essere presente il valore che era stao messo dalla CALL. Quindi nella funzione il numero di istruzioni PUSHW (e PUSHL) deve essere uguale al numero di istruzione POPW (e POPL) affinchè al momento dell’esecuione della RET lo stack sia nelle stesse condizioni in cui si trovava all’inizio della funzione.

Le microistruzioni dell’istruzione RET sono:

1. PC_{OUT}, MAR $_{IN}$, READ, SELECT[4], ADD, Z_{IN}
2. WMFC, Z_{OUT}, PC_{IN}
3. MDR_{OUT}, IR_{IN}
4. ESP_{OUT}, SELECT[4], ADD, Z_{IN}

5. Z_{OUT}, ESP_{IN}, MAR_{IN}, READ

6. WMFC

7. MDR_{OUT}, $PC_{IN,}$ END

Dopo la fase di “Push” di PC, bisogna fare attenzione al tipo di salto:

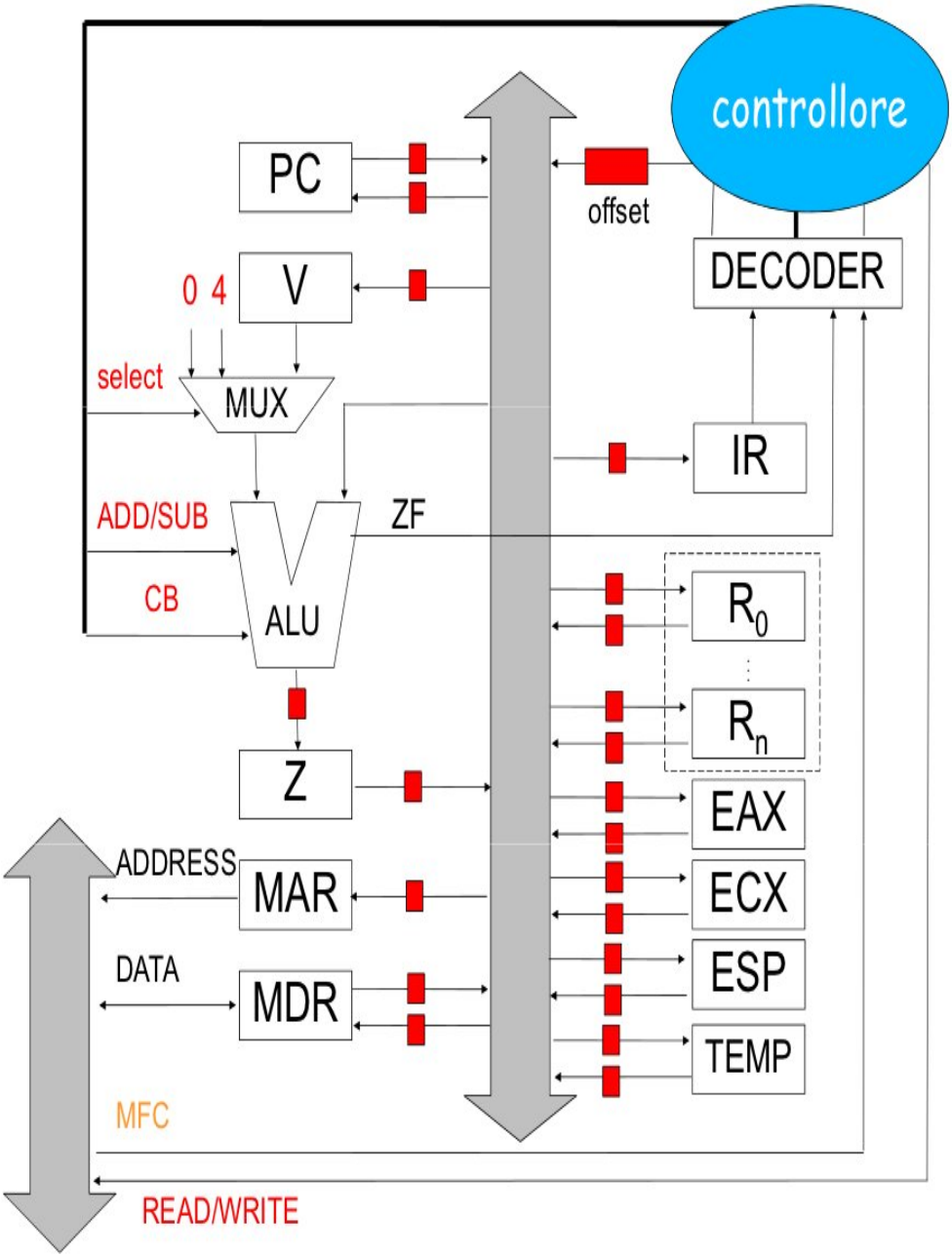
Salto Assoluto: il valore “value” deve essere inserito direttamente in PC; PC=[Value]

Salto Relativo: il valore “value” deve essere sommato a PC e il valore ottenuto deve essere inserito nel registro PC; PC=[PC+Value]

Ovviamente tutti gli esercizi possono avere delle varianti di tema; può succedere che si presenti una ADD non semplicemente tra 2 valori, ma ad esempio ADD $4(%eax + %ebx), %ecx (un caso un po' assurdo). Bisogna essere in grado di unire tutte le varianti e scindere l'istruzione in altrettante istruzioni conosciute, in modo da non sbagliare. Sicuramente un buon esercizio può aiutare a riconoscere le parti.

Di seguito si trova l'architettura di riferimento per svolgere gli esercizi. È buona cosa ricordarla a memoria per svolgere gli esercizi durante l'esame.

Architettura di riferimento per le microistruzioni



Esercizio 1

Elencare le micro istruzioni relative alla completa esecuzione della seguente istruzione assembler (Intel 80386 AT&T), assumendo che la CPU abbia un solo BUS, che l'istruzione sia composta da una sola parola e che (%EBX) rappresenti un metodo di indirizzamento indiretto a registro (usare solamente le righe necessarie):

MOVL (%EAX), %EBX

Soluzione:

1. PC_{OUT}, MAR_{IN}, READ, SELECT[4], ADD, Z_{IN}
2. WMFC, Z_{OUT}, PC_{IN}
3. MDR_{OUT}, IR_{IN}

} Fetch

4. EAX_{OUT}, MAR_{IN}, READ
5. WMFC
6. MDR_{OUT}, EBX_{IN}, END

Commenti:

Fetch:

1. Viene caricato, letto tramite MAR ed incrementato il PC (nell'architettura Intel a 32 bit, per fare ciò bisogna incrementare di 4)
2. Viene messo in PC il valore PC+4
3. Escono i dati da MDR che vanno messi nel registro delle istruzioni (IR)

4.Esce il valore di EAX e viene passato al registro degli indirizzi MAR per la lettura
5. Aspetto
7. L'indirizzo viene passato a EBX. Termine istruzione

Esercizio 2

Elencare le micro istruzioni relative alla completa esecuzione della seguente istruzione assembler (Intel 80386 AT&T), assumendo che la CPU abbia un solo BUS, che l'istruzione sia composta da una sola parola e che (%EBX) rappresenti un metodo di indirizzamento indiretto a registro (usare solamente le righe necessarie):

ADDL (%EBX), %EAX

Soluzione:

1.PC_{OUT}, MAR_{IN}, READ, SELECT[4], ADD, Z_{IN}
2.WMFC, Z_{OUT}, PC_{IN}
3.MDR_{OUT}, IR_{IN}

} Fetch

4.EBX_{OUT}, MAR_{IN}, READ
5.WMFC, EAX_{OUT}, V_{IN}
6.MDR_{OUT}, SELECT[V], ADD, Z_{IN}
7.Z_{OUT}, EAX_{IN}, END

Commenti:

Fetch:

1. Viene caricato, letto tramite MAR ed incrementato il PC (nell'architettura Intel a 32 bit, per fare ciò bisogna incrementare di 4)
2. Viene messo in PC il valore PC+4
3. Escono i dati da MDR che vanno messi nel registro delle istruzioni (IR)

4. Esce il valore contenuto in EBX e viene passato al registro MAR affinchè possa fare la codifica dell'indirizzo
5. Si attende la risposta del MAR attraverso il comando WMFC (il valore deve essere letto dalla memoria); nel frattempo, siccome il BUS non è occupato, il valore contenuto nel registro EAX viene memorizzato temporaneamente nel registro ausiliario della ALU, V
6. Il contenuto del valore precedentemente puntato dal contenuto di EBX esce dall'MDR e viene sommato a EAX in quanto il metodo di indirizzamento è indiretto a registro
7. Il risultato della somma dei due registri viene memorizzato in EAX. Termine dell'istruzione.

Esercizio 3

Elencare le micro istruzioni relative alla completa esecuzione (caricamento, decodifica, esecuzione) della seguente istruzione assembler (Intel 80386 AT&T), assumendo che la CPU abbia un solo BUS, che l'istruzione sia composta da una sola parola, che (%EAX) rappresenti un metodo di indirizzamento indiretto a registro e che l'indirizzo di salto della procedura sia relativo al PC:

CALL (%EAX)

Soluzione:

1.PC_{OUT}, MAR_{IN}, READ, SELECT[4], ADD, Z_{IN}
2.WMFC, Z_{OUT}, PC_{IN}
3.MDR_{OUT}, IR_{IN}

} Fetch

4.ESP_{OUT}, SELECT[4], SUB, Z_{IN}
5.Z_{OUT},ESP_{IN},MAR_{IN}
6.PC_{OUT}, MDR_{IN}, WRITE
7.WMFC, EAX_{OUT}, MAR_{IN}, READ

} Salvataggio stack

8.WMFC,PC_{OUT},V_{IN}
9.MDR_{OUT}, SELECT[V], ADD, Z_{IN}
10.Z_{OUT}, PC_{IN}, END

Commenti:

Fetch:

1. Viene caricato, letto tramite MAR ed incrementato il PC (nell'architettura Intel a 32 bit, per fare ciò bisogna incrementare di 4)
2. Viene messo in PC il valore PC+4
3. Escono i dati da MDR che vanno messi nel registro delle istruzioni (IR)

Salvataggio dello stack:

4. Viene decrementato il valore dell'ESP (stack pointer) per puntare ad una cella vuota dello Stack, inizio dell'operazione di PUSH nella pila di PC.
5. ESP viene caricato nella MAR per la fase di scrittura.
6. Il PC passa nel registro dei dati MDR e viene eseguita l'operazione di scrittura, salvando in memoria (nella cella nuova dello Stack) l'indirizzo attuale di PC.

7.Attesa scrittura.Nel frattempo il valore di EAX viene passato al MAR come indirizzo di memoria.
8.Attesa lettura; nel frattempo PC viene caricato nel registro V, siccome il BUS non è occupato.
9.Dal MDR esce il valore puntato dal contenuto di EAX che viene sommato al PC in quanto il salto è relativo.
10. L'indirizzo reale dell'istruzione da eseguire viene passato al PC. Termine dell'istruzione.

Esercizio 4

Elencare le micro istruzioni relative al caricamento, decodifica ed esecuzione della seguente istruzione assembler (Intel 80386 AT&T), assumendo che la CPU abbia un solo BUS, che l'istruzione sia composta da una sola parola, che (%EAX) rappresenti un metodo di indirizzamento indiretto a registro e che il salto condizionato sia di tipo diretto:

JZ (%EAX)

Soluzione:

1.PC_{OUT}, MAR_{IN}, READ, SELECT[4], ADD, Z_{IN}
2.WMFC, Z_{OUT}, PC_{IN}
3.MDR_{OUT}, IR_{IN}

} Fetch

4.if (!ZERO) END, EAX_{OUT}, MAR_{IN}, READ
5.WMFC, PC_{OUT}, V_{IN}
6.MDR_{OUT}, SELECT[V], ADD, Z_{IN}
7.Z_{OUT}, PC_{IN}, END

Commenti:

Fetch:

1. Viene caricato, letto tramite MAR ed incrementato il PC (nell'architettura Intel a 32 bit, per fare ciò bisogna incrementare di 4)
2. Viene messo in PC il valore PC+4
3. Escono i dati da MDR che vanno messi nel registro delle istruzioni (IR)

4. Se il risultato non è zero termine dell'istruzione, altrimenti il contenuto di EAX viene passato al MAR come un indirizzo e viene letto
5. Attesa per l'interpretazione dell'indirizzo; nel frattempo, siccome il BUS non è occupato, il valore corrente del PC viene messo nella locazione registro ausiliario della ALU , V
6. L'indirizzo prodotto da MDR viene sommato a quello del PC attuale
7. L'indirizzo della prossima istruzione da eseguire viene messo nel PC. Termine dell'istruzione.

Esercizio 5

Elencare le micro istruzioni (insieme dei segnali di controllo) relative alla completa esecuzione (caricamento, decodifica, esecuzione) della seguente istruzione assembler (Intel 80386 AT&T), assumendo che la CPU abbia un solo BUS, che l'istruzione sia composta da una sola parola, che (%EAX) rappresenti un metodo di indirizzamento indiretto a registro e che il salto condizionato sia di tipo diretto:

JZ (%EAX)+%SI

Soluzione:

1.PC_{OUT}, MAR_{IN}, READ, SELECT[4], ADD, Z_{IN}
2.WMFC, Z_{OUT}, PC_{IN}
3.MDR_{OUT}, IR_{IN}

} Fetch

4.if (!ZERO) END, EAX_{OUT}, MAR_{IN}, READ
5.WMFC, SI_{OUT}, V_{IN}
6.MDR_{OUT}, SELECT[V], ADD, Z_{IN}
7.Z_{OUT}, V_{IN}
8.PC_{OUT}, SELECT[V], ADD, Z_{IN}
9.Z_{OUT}, PC_{IN}, END

Commenti:

Fetch:

1. Viene caricato, letto tramite MAR ed incrementato il PC (nell'architettura Intel a 32 bit, per fare ciò bisogna incrementare di 4)
2. Viene messo in PC il valore PC+4
3. Escono i dati da MDR che vanno messi nel registro delle istruzioni (IR)

4. Se il risultato non è zero termine dell'istruzione, altrimenti il contenuto di EAX viene passato al MAR come un indirizzo e viene letto
5. Attesa per l'interpretazione dell'indirizzo; nel frattempo il valore di SI viene messo nel registro ausiliario V
6. L'indirizzo prodotto da MDR viene sommato a SI e viene memorizzato in Z
7. Il valore contenuto in Z viene trasferito in V
8. Il valore di V viene sommato al valore attuale del PC, il tutto viene salvato in Z
9. L'indirizzo della prossima istruzione da eseguire viene messo nel PC. Termine dell'istruzione.

Esercizio 6

Elencare le micro istruzioni (insieme dei segnali di controllo) relative alla completa esecuzione (caricamento, decodifica, esecuzione) della seguente istruzione assembler (Intel 80386 AT&T), assumendo che la CPU abbia **TRE** BUS, che l'istruzione sia composta da una sola parola, che (%EAX) rappresenti un metodo di indirizzamento indiretto a registro e che il salto sia di tipo relativo:

CALL (%EAX)

Soluzione:

1.$PC_{OUT\ A}$, NOP, $MAR_{IN\ C}$, READ
2.$PC_{OUT\ A}$, SET CARRY-IN, ADD, $PC_{IN\ C}$, WMFC
3.$MDR_{OUT\ B}$, IR_{IN}

} Fetch

4.$ESP_{OUT\ A}$, SET CARRY-IN, SUB, $ESP_{IN\ C}$,$MAR_{IN\ C}$
5.$PC_{OUT\ A}$, NOP, $MDR_{IN\ C}$, WRITE
6.WMFC

} Salvataggio stack

7.$EAX_{OUT\ A}$, NOP, $MAR_{IN\ C}$, READ
8.WMFC
9.$MDR_{OUT\ A}$, $PC_{OUT\ B}$, ADD, $PC_{IN\ C}$, END.

Commenti:

Fetch:
1. Il valore corrente del PC viene passato nel registro MAR
2. Il valore del PC viene incrementato di uno
3. Viene passato al registro delle istruzioni l'indirizzo della prossima istruzione da eseguire

Salvataggio stack:
4. Il valore dello stack pointer viene decrementato,salvato e passato al registro degli indirizzi.
5. Il valore del PC precedentemente incrementato di uno passa nel *memory date register*
6. Attesa scrittura del PC nello stack.

7. Il contenuto del registro EAX viene passato al MAR per esser interpretato
8. Attesa
9. Il contenuto di EAX esce dal MDR come un indirizzo che viene sommato al PC in quanto il salto è relativo, il nuovo indirizzo dell'istruzione da eseguire viene quindi memorizzato nel PC affinché l'esecuzione possa continuare.

Esercizio 7

Elencare le micro istruzioni (insieme dei segnali di controllo) relative alla completa esecuzione (caricamento, decodifica, esecuzione) della seguente istruzione assembler (Intel 80386 AT&T), assumendo che la CPU abbia **TRE** BUS, che l'istruzione sia composta da una sola parola e che (%EAX) rappresenti un metodo di indirizzamento indiretto a registro (usare solamente le righe necessarie). Salto relativo:

JMP (%EAX)

Soluzione:

1.$PC_{OUT\ A}$, NOP, $MAR_{IN\ C}$, READ
2.$PC_{OUT\ A}$, SET CARRY-IN, ADD, $PC_{IN\ C}$, WMFC
3.$MDR_{OUT\ B}$, IR_{IN}

} Fetch

4.$EAX_{OUT\ A}$, NOP, MAR_{IN}, READ
5.WMFC
6.$MDR_{OUT\ A}$, NOP, PC_{IN}, END

Commenti:

Fetch:
1. Viene scaricato il valore del PC e trasferito al MAR per essere interpretato
2. Il valore corrente del PC viene incrementato di uno e il nuovo valore ottenuto PC+1 viene memorizzato in PC
3. L'indirizzo elaborato dal MDR viene trasferito nell'IR

4. Il valore di EAX viene passato al MAR per essere interpretato come un indirizzo
5. Attesa dell'interpretazione
6. L'indirizzo ottenuto viene passato al PC in modo tale che quest'ultimo contenga l'indirizzo della prossima istruzione da eseguire. Fine.

Esercizio 8

Elencare le microistruzioni relative alla completa esecuzione (caricamento, decodifica, esecuzione) della seguente istruzione assembler (Intel 80386 AT&T), assumendo che la CPU abbia un solo BUS, che l'istruzione sia composta da una sola parola, che (%EAX) rappresenti un metodo di indirizzamento indiretto a registro e che l'indirizzo di salto della procedura sia relativo al PC

JNZ (%EAX)

Soluzione:

1.PC_{OUT}, MAR_{IN}, READ, SELECT[4], ADD, Z_{IN}
2.WMFC, Z_{OUT}, PC_{IN}
3.MDR_{OUT}, IR_{IN}

} Fetch

4.if (ZERO) END, EAX_{OUT}, MAR_{IN}, READ
5.WMFC, PC_{OUT}, V_{IN}
6.MDR_{OUT}, SELECT[V], ADD, Z_{IN}
7.Z_{OUT}, PC_{IN}, END

Commenti:

Fetch:

1. Viene caricato, letto tramite MAR ed incrementato il PC (nell'architettura Intel a 32 bit, per fare ciò bisogna incrementare di 4)
2. Viene messo in PC il valore PC+4
3. Escono i dati da MDR che vanno messi nel registro delle istruzioni (IR)

4. Se il risultato è zero termine dell'istruzione, altrimenti il contenuto di EAX viene passato al MAR come un indirizzo e viene letto (per ottenere il valore effettivo)
5. Attesa per l'interpretazione dell'indirizzo, il valore corrente del PC viene messo nella locazione V
6. L'indirizzo ottenuto da MDR viene sommato a quello del PC attuale
7. L'indirizzo della prossima istruzione da eseguire viene messo nel PC. Fine dell'istruzione.

Esercizio 9

Elencare le micro istruzioni relative alla completa esecuzione (caricamento, decodifica, esecuzione) della seguente istruzione assembler (Intel 80386 AT&T), assumendo che la CPU abbia un solo BUS, che l'istruzione sia composta da una sola parola, che (%Exx) rappresenti un metodo di indirizzamento indiretto a registro e che l'indirizzo di salto della procedura sia assoluto:

JNZ (%EAX + %EBX)

Soluzione:

1.PC_{OUT}, MAR_{IN}, READ, SELECT[4], ADD, Z_{IN}
2.WMFC, Z_{OUT}, PC_{IN}
3.MDR_{OUT}, IR_{IN}

} Fetch

4.if (ZERO) END, EAX_{OUT}, V_{IN}
5.EBX_{OUT}, SELECT[V], ADD, Z_{IN}
6.Z_{OUT}, MAR_{IN}, READ
7.WMFC
8.MDR_{OUT},PC_{IN}, END

Commenti:

Fetch:

1. Viene caricato, letto tramite MAR ed incrementato il PC (nell'architettura Intel a 32 bit, per fare ciò bisogna incrementare di 4)
2. Viene messo in PC il valore PC+4
3. Escono i dati da MDR che vanno messi nel registro delle istruzioni (IR)

4. Se il risultato è zero termine dell'istruzione, altrimenti il contenuto di EAX viene passato in V
5. EBX viene sommato al valore di EAX ed il risultato viene memorizzato in Z
6. La somma ottenuta viene passata al MAR come un indirizzo da interpretare
7. Attesa
8. L'indirizzo della prossima istruzione da eseguire (quindi la somma di EAX ed EBX), viene caricato nel PC (il salto infatti è assoluto). Termine istruzione.

Esercizio 10

Elencare e commentare le micro istruzioni relative alla completa esecuzione (caricamento, decodifica, esecuzione) della seguente istruzione assembler (Intel 80386 AT&T), assumendo che la CPU abbia un solo BUS, che l'istruzione sia composta da una sola parola, che (%Exx) rappresenti un metodo di indirizzamento indiretto a registro e che l'indirizzo di salto della procedura sia assoluto:

CALL (%EAX + %EBX)

Soluzione:

1.PC_{OUT}, MAR_{IN}, READ, SELECT[4], ADD, Z_{IN}
2.WMFC, Z_{OUT}, PC_{IN}
3.MDR_{OUT}, IR_{IN}

} Fetch

4.ESP_{OUT}, SELECT[4], SUB, Z_{IN}
5.Z_{OUT}, ESP_{IN}, MAR_{IN}
6.PC_{OUT},MDR_{IN},WRITE
7.WMFC, EAX_{OUT},V_{IN}

} Salvataggio Stack

8.EBX_{OUT}, SELECT[V], ADD, Z_{IN}
9. Z_{OUT}, MAR_{IN}, READ
10.WMFC
11.MDR_{OUT},PC_{IN}, END.

Commenti:

Fetch:
1. Viene caricato, letto tramite MAR ed incrementato il PC (nell'architettura Intel a 32 bit, per fare ciò bisogna incrementare di 4)
2. Viene messo in PC il valore PC+4
3. Escono i dati da MDR che vanno messi nel registro delle istruzioni (IR)

Salvataggio dello stack:
4. Viene decrementato il valore dell'ESP (stack pointer) per puntare ad una cella vuota dello Stack, inizio dell'operazione di PUSH nella pila di PC.
5. ESP viene caricato nella MAR per la fase di scrittura.
6. Il PC passa nel registro dei dati MDR e viene eseguita l'operazione di scrittura, salvando in memoria (nella cella nuova dello Stack) l'indirizzo attuale di PC.
7. Attesa scrittura, Nel frattempo il valore di EAX esce e viene salvato nel registro ausiliario della ALU V
8. Il valore di EBX viene sommato a EAX
9. EAX+EBX viene passato al registro degli indirizzi MAR per la lettura
10. Attesa (ciò che uscirà da MDR sarà il nuovo valore di PC cioè l'indirizzo dell'istruzione da eseguire)
11. L'indirizzo reale dell'istruzione da eseguire viene passato al PC in quanto il salto è assoluto.
    Termine istruzione

Esercizio 11

Elencare e commentare le micro istruzioni relative alla completa esecuzione (caricamento, decodifica, esecuzione) della seguente istruzione assembler (Intel 80386 AT&T), assumendo che la CPU abbia un solo BUS, che l'istruzione sia composta da una sola parola, che (%Exx) rappresenti un metodo di indirizzamento indiretto a registro, che $4 sia un indirizzamento immediato e che l'indirizzo di salto della procedura sia assoluto (usare solamente le righe necessarie e commentare ogni istruzione):

CALL (%EAX + $4)

Soluzione:

1.PC_{OUT}, MAR_{IN}, READ, SELECT[4], ADD, Z_{IN}
2.WMFC, Z_{OUT}, PC_{IN}
3.MDR_{OUT}, IR_{IN}

} Fecth

4.ESP_{OUT}, SELECT[4], SUB, Z_{IN}
5.Z_{OUT}, ESP_{IN},MAR_{IN}
6.PC_{OUT}, MDR_{IN}, WRITE
7.WMFC, EAX_{OUT},V_{IN}

} Salvataggio Stack

8. $OFFSET(IR)_{OUT}$,SELECT[V], ADD, Z_{IN}
9.Z_{OUT},MAR_{IN}, READ
10.WMFC
11.MDR_{OUT}, PC_{IN}, END

Commenti:

Fetch:
1. Viene caricato, letto tramite MAR ed incrementato il PC (nell'architettura Intel a 32 bit, per fare ciò bisogna incrementare di 4)
2. Viene messo in PC il valore PC+4
3. Escono i dati da MDR che vanno messi nel registro delle istruzioni (IR)

Salvataggio dello stack:
4. Viene decrementato il valore dell'ESP (stack pointer) per puntare ad una cella vuota dello Stack, inizio dell'operazione di PUSH nella pila di PC.
5. ESP viene caricato nella MAR per la fase di scrittura.
6. Il PC passa nel registro dei dati MDR e viene eseguita l'operazione di scrittura, salvando in memoria (nella cella nuova dello Stack) l'indirizzo attuale di PC.
7. Attesa scrittura nello Stack, mentre il valore di EAX viene caricato nel registro ausiliario V (la seconda operazione non fa parte del salvataggio dello stack, ma è stata scritta sulla stessa riga per "risparmiare" un ciclo, in quanto il BUS in quel momento non era occupato).

8. L'offset di IR che contiene il valore 4 (poiché è una costante) viene sommato a EAX
9. Il risultato della somma passa al registro degli indirizzi MAR, in quanto il metodo di indirizzamento è indiretto.
10. Attesa lettura
11. Il valore letto passa nel PC,in quanto il salto è assoluto. Termine struzione

Memorie Ram, Cache e Virtuali: regole ed accorgimenti

Per questo argomento, esistono vari tipi di esercizio. Il più semplice è quello di trovare la dimensione dell'indirizzo della Ram e della Cache.

Per questo tipo di esercizio, è utile imparare delle piccole formule. Questi infatti si svolgono sempre allo stesso modo, ossia bisogna identificare la dimensione dei campi che compongono l'indirizzo.

I campi sono: PAROLA, BLOCCO, ETICHETTA.

Innanzitutto bisogna identificare la **dimensione dell'indirizzo**; per fare ciò, bisogna semplicemente prendere il numero di parole nella Ram e scomporlo in potenza di 2. Si otterrà perciò un numero 2^n, il cui "n" sarà il numero corrispondente alla dimensione dell'indirizzo.

Sapere quanto valgono, in potenze di 2, 1KB, 1MB, 1GB ecc.. può agevolare l'operazione:

1KB = 1024 bit = 2^{10} 1MB = 1024 kb = 2^{20}

1GB = 1024 mb = 2^{30} 1TB = 1024 gb = 2^{40}

…

Se quindi il numero di parole nella Ram è 4M parole:

$4 = 2^2$

$1MB = 2^{20}$

$2^{20}*2^2 = 2^{22}$ 22 bit (Dimensione dell'indirizzo)

Successivamente bisogna scoprire la **dimensione del campo PAROLA**; per fare ciò, bisogna considerare il numero di parole per blocco e trasformarlo in potenza di 2. Si ottiene perciò il numero 2^n, il cui "n" è il numero corrispondente alla dimensione del campo PAROLA.

Per il **campo BLOCCO**, è necessario ottenere la dimensione della Cache (a volte indicata con "# Cache", altre con "dim. Cache"); per far ciò, occorre dividere il numero di parole nella Cache per il numero di parole per blocco.

Infine bisogna ottenere il "Cache Set", dividendo la dimensione della Cache per il tipo di set (dato nella consegna, come per esempio "4 set associativa", il 4 è il numero di set da utilizzare

convertendolo in potenza di 2).

Il tutto si potrebbe riassumere attraverso delle “formule”:

Blocco = dim. Cache / Tipo Set

↓

dim. Cache = parole Cache / parole Blocco

Per ottenere infine il **campo ETICHETTA**, occorre ottenere la dimensione della Ram, dividendo il numero di parole nella Ram per il numero di parole del blocco; dopo ciò, si divide il numero appena ottenuto per il “Cache Set”.

Come prima, possiamo riassumere il tutto attraverso delle “formule”:

Etichetta = dim. Ram / Cache Set

↓

dim. Ram = parole Ram / parole Blocco

Per completare l'esercizio, occorre disegnare l'indirizzo ottenuto e verificare se la somma delle dimensioni dei campi restituisce un risultato uguale alla dimensione dell'indirizzo.

Il disegno avrà una forma del genere:

ETICHETTA	***BLOCCO***	***PAROLA***
Bit	Bit	bit

bit Etichetta + bit Blocco + bit Parola = bit dim. Indirizzo.

In alcuni esercizi può capitare che vengano chieste le dimensioni dell’indirizzo della Cache.

Per trovare tali dimensioni basta ridurre in potenza di 2 le parole della Cache e sottrarre i bit dei campi Blocco e Parola (questi due campi rimangono invariati, si va a modificare solo il campo Etichetta). Il risultato sarà la “nuova” dimensione del campo Etichetta.

Es.

Cache 4-set associativa
8k parole

ETICHETTA	*BLOCCO*	*PAROLA*
11 bit	5 bit	6 bit

$8k = 2^3 * 2^{10} = 2^{13}$ 13 bit per l'indirizzamento

13 – 5 (bit Blocco) - 6 (bit Parola) = 2 bit (campo Etichetta)

ETICHETTA	*BLOCCO*	*PAROLA*
2 bit	5 bit	6 bit

N.B. La Cache può essere anche essere:

- **accesso diretto**: in questo caso non teniamo conto del Cache Set perché vale 1 ($2^0=1$)
- **completamente associativa**: non esiste il campo Blocco e per calcolare il campo Etichetta basta sottrarre dai bit dell'indirizzamento i bit del campo Parole.

Un altro tipo di esercizio è quello di considerare quanti "Cache miss" avvengono durante l'esecuzione di un programma, in base alla grandezza della Cache o della Ram (in assenza di Cache). Di solito questi esercizi sono particolari per ogni situazione, però in pratica basta calcolare quanto viene occupata la Cache o la Ram caricando in esse il codice del programma e i dati e calcolare quanti e quali dati devono essere sostituiti ciclicamente.

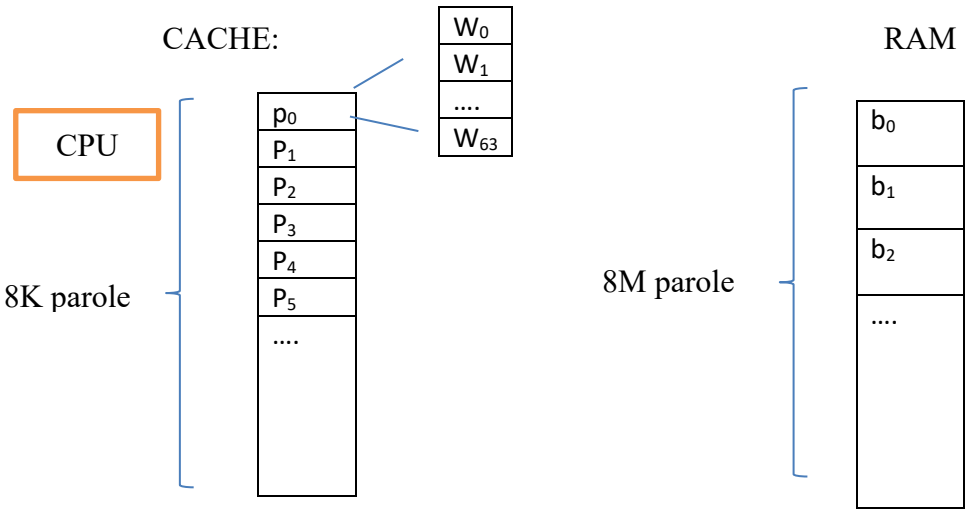
La stessa cosa riguarda le memorie virtuali, in cui o viene chiesto di determinare i bit che corrispondono alla lunghezza dell'indirizzo logico oppure di calcolare quante pagine devono essere sovrascritte da nuovi dati ciclicamente.

Esercizio 1 *Prima parte*

Si consideri una CPU dotata di memoria cache 4-associativa di 8K parole con 64 parole per blocco. Questa CPU è collegata ad una memoria RAM da 8M parole.

Definire le dimensioni dell'indirizzo necessario a indirizzare tutta la memoria RAM e definire le dimensioni dei campi PAROLA, BLOCCO ed ETICHETTA in cui questo indirizzo può essere suddiviso. Motivare la risposta con un opportuno schema.

Soluzione:



Cache	4-set associativa
	8k parole
Blocco	64 parole
Ram	8M parole

Dimensioni dell'*indirizzo* necessario ad indirizzare tutta la memoria Ram:
corrisponde alla potenza della dimensione (in base 2) della Ram, quindi se si prende il numero 2^n , n è il numero interessato.

Dimensioni dell'indirizzo necessario ad indirizzare tutta la memoria Ram:

dim. Ram 8M parole = 2^{23} 23 bit per l'indirizzo

Dimensione del campo parola:

64 parole = 2^6 6 bit necessari per costruire il campo Parola

Dimensione del campo blocco:

Blocco = dim. Cache / Tipo Set

dim. Cache = parole Cache / parole Blocco

dim. Cache = 8 K parole / 64 * 4 parole

Blocco = $2^{13} / 2^8 = 2^5$ 5 bit necessari a costruire il campo Blocco

Dimensione del campo etichetta:

Etichetta = dim. Ram / Cache Set

dim. Ram = parole Ram / parole Blocco

dim. Ram = 8 M parole / 64 parole $= 2^{17}$

Etichetta = $2^{17} / {}^{25} = 2^{12}$ 12 bit necessari a costruire il campo Etichetta

ETICHETTA	BLOCCO	PAROLA
12 Bit	5 Bit	6 bit

12 + 5 + 6 = 23 Sommati tutti i campi danno le dimensioni dell'indirizzamento

Esercizio 1 *Seconda parte*

Si assuma che la cache appena descritta sia utilizzata per i dati, che sia inizialmente vuota e che utilizzi un algoritmo di sostituzione dei blocchi di tipo LRU (sostituzione dell'elemento meno utilizzato di recente). La CPU esegue un programma che accede in sequenza a tutti gli elementi di un array di 8320 parole (ogni elemento ha le dimensioni di una parola) che è memorizzato a partire dall'indirizzo 0. Questa operazione di scansione è effettuata all'interno di un ciclo che viene eseguito 5 volte. Si assuma che il tempo di accesso alla cache sia di 1T e che il tempo di accesso alla memoria sia di 10T (entrambi i tempi si riferiscono alla lettura di una parola). Calcolare il rapporto (fattore di miglioramento) tra il sistema in presenza di cache e in assenza di cache per l'esecuzione di questo programma.

Soluzione:

Cache 4-set associativa
8k parole
tempo di accesso in lettura pari ad 1T

Blocco 64 parole
Ram 8M parole
tempo di accesso in lettura pari a 10T

Algoritmo LRU

Array 8320 parole

Essendo la cache 4-set associativa, essa ha 2^2 blocchi per set, quindi 4 blocchi per set.
La cache contiene 8k parole, quindi 8192 parole e 32 set (8192/64/4) che chiameremo set0,set1,...,set31.
L'array contiene 8320 parole e siccome in cache ne possono risiedere massimo 8192, si ha che 128 parole (corrispondenti a 2 blocchi) restano fuori dalla cache e dovranno essere caricate sovrascrivendo altri blocchi.

La scansione dell'array viene effettuata all'interno di un ciclo che viene eseguito 5 volte. Il tempo di trasferimento dalla cache alla CPU dipende soltanto dalla dimensione dell'array e della velocità di accesso, quindi è indipendente dalla dimensione della cache:

Accessi in lettura senza cache :

$$8320 \text{ parole} \times 5 = 41600 \text{ parole}$$

Tempo accessi totale senza cache:

$$T_{nocache} = 41600 * 10t = 416000t$$

Calcoliamo ora il numero di accessi alla Ram per caricare la cache.

1^ ciclo 130 blocchi (8320 parole / 64 parole) dalla Ram alla cache (la cache è inizialmente vuota). Utilizzando l'algoritmo LRU i blocchi 128 e 129 sostituiscono rispettivamente i blocchi 0 e 1 che occupano il set0 e il set1. In totale ci saranno 130 miss.

2^ - 5^ ciclo I blocchi 0 e 1 dovranno essere ricaricati dalla Ram, quindi sostituiranno i blocchi 8 e 9 del set0 rispettivamente set1. In questo modo tutti i blocchi del set0 e set1 verranno sostituiti con blocchi prelevati dalla Ram. In totale ci saranno 8 miss più i 2 miss dovuti ai blocchi 128 e 129, ovvero 10 miss per ciclo.

Tot = 130 + 4*10 = 170 miss per i cicli di lettura

- Tot blocchi letti da ram (con miss) : 170
- Tot parole lette da ram: 170 blocchi * 64 parole = 10880 parole
- Tot parole lette da cache (con hit) = tot letture complessive – tot parole lette da ram =

(130*64*5) – 10880 = 30720 parole

Il tempo di esecuzione con cache ` e:

T_{cache} = tot parole lette da ram * 10t + tot parole lette da cache * 1t =

10880 * 10t + 30720 *1t = 139520t

Lo speed up è il rapporto tra il tempo di esecuzione del programma senza cache e in presenza di cache, quindi si ha che:

$T_{nocache}/T_{cache}$ = 416000t / 139520t = 2.98

Il che implica che in questo caso la cache velocizza di quasi 3 volte la CPU.

Esercizio 2

Un programma consiste in un totale di 300 istruzioni e contiene un ciclo di 50 istruzioni che è eseguito 15 volte. Il processore contiene una cache. Prelevare ed eseguire un' istruzione che si trova in memoria principale richiede 20 unità di tempo. Se l'istruzione si trova nella cache, prelevare ed eseguire l'istruzione richiede solo 2 unità di tempo. Ignorando gli accessi dovuti al prelievo degli operandi, calcolare il rapporto tra il tempo di esecuzione del programma senza la cache e il tempo di esecuzione con la cache. Questo rapporto è chiamato accelerazione (speedup) dovuta all'uso della cache. Assumere che la cache sia inizialmente vuota, che sia sufficientemente grande da contenere il ciclo e che il programma inizi con tutte le istruzioni presenti in memoria principale.

SOLUZIONE:

Il tempo di esecuzione senza cache ` e:

$$T = 250\times20 + 50\times20\times15 = 20.000$$

Il tempo di esecuzione con cache ` e:

$$T_{cache} = 300\times20 + 50\times2\times14 = 7.400$$

Quindi l'accelerazione (speedup) ` e

$$T/T_{cache} = 2,7$$

Esercizio 3

a) Supporre che il tempo di esecuzione per un programma sia proporzionale al tempo di prelievo delle istruzioni. Assumere che prelevare un'istruzione dalla cache richieda 1 unità di tempo, ma che prelevarla dalla memoria principale richieda 10 unità di tempo. Inoltre assumere che un'istruzione richiesta sia trovata nella cache con probabilità 0.96. Infine assumere che, se un'istruzione non è trovata nella cache, essa debba essere prima prelevata dalla memoria principale e inserita nella cache, poi prelevata dalla cache per essere eseguita. Calcolare il rapporto tra il tempo di esecuzione di un programma senza cache e il tempo di esecuzione con cache. Questo rapporto è chiamato accelerazione (speedup) dovuta alla presenza della cache.
b) Se la dimensione della cache è raddoppiata, assumere che la probabilità di non trovare una istruzione richiesta sia dimezzata. Ripetere la parte a per una cache con dimensione doppia.

SOLUZIONE:

(a) Il tempo di prelievo di un'istruzione senza cache `e di 10 unità di tempo. Il tempo medio di prelievo con cache ` e

$$0,96\times1 + (0,04\times(10 + 1)) = 1,4$$

Quindi

Accelerazione $10/1,4 = 7,1$

(b) Raddoppiando la dimensione della cache, il tempo medio di prelievo con cache diventa

$$0,98\times1 + (0,02\times(10 + 1)) = 1,2$$

Quindi

$$\text{accelerazione} = 10/1,2 = 8,3$$

Esercizio 4 *Prima parte*

Si consideri un sistema a memoria virtuale con spazio logico di 4G parole, una memoria fisica di 32M parole e dimensione delle pagine di 16K parole.

Determinare il numero di bit che definiscono:

Lunghezza indirizzo logico:
Di cui per Num. Pag. Logica
per lo spiazzamento

Lunghezza indirizzo fisico:
di cui per Num. Pag. Fisica
per lo spiazzamento

Soluzione:

Spazio logico 4G parole → $2^2 * 2^{30} = 2^{32} =$ **32** (lunghezza indirizzo logico)
Memoria fisica 32M parole → $2^5 * 2^{20} = 2^{25} =$ **25** (lunghezza indirizzo fisico)
Dimensione pagine 16k parole → $2^4 * 2^{10} = 2^{14} =$ **14** (lunghezza spiazzamento)

per calcolare il numero pagina logica (indirizzo logico): 32 – 14 = **18**
per calcolare il numero pagina logica (indirizzo fisico): 25 – 14 = **11**

Lunghezza indirizzo logico:	32
Di cui per Num. Pag. Logica:	18
Per lo spiazzamento:	14
Lunghezza indirizzo fisico:	25
Di cui per Num. Pag. Logica:	11
Per lo spiazzamento:	14

Esercizio 4 *Seconda parte*

Nella seguente tabella sono riportati alcuni valori del parametro R (numero di pagine residenti per processo); sapendo che il Sistema Operativo occupa permanentemente 448 pagine e che sono stati creati 22 processi, indicare per ogni valore di R il numero di processi in stato di "fuori memoria" ossia che non possono avere tutte le pagine in memoria.

Soluzione:

Dato **R** il numero di pagine residenti per processo, si ha:

Sistema Operativo: occupa 448 pagine permanentemente

Numero Processi: 22

dim. Memoria fisica / dim. Pagina = Num. totale di pagine

$2^{25} / 2^{14} = 2^{11} = 2048$ pagine

2048 - 448 = 1600 pagine libere

1600 / 40 = 40 => 22 – 40 = 22 < 40 Non ci sono processi fuori memoria
1600 / 80 = 20 => 22 – 20 = 2 processi fuori memoria
1600 / 160 = 10 => 22 – 10 = 12 processi fuori memoria
1600 / 320 = 5 => 22 – 5 = 17 processi fuori memoria
1600 / 800 = 2 => 22 – 2 = 20 processi furoi memoria

R	**40**	**80**	**160**	**320**	**800**
Numero di processi fuori memoria	0	2	12	17	20

Esercizio 4 *Terza parte*

Specificare come è stato calcolato il valore per R = 160:

Soluzione:

I processi in totale sono 22 e il numero di pagine residenti per processo 160, quindi il numero di pagine necessarie per far rimanere i processi in memoria sarebbe 22 * 160 = 3520.
Il numero di pagine disponibili è però 1600.
E' quindi evidente che i processi i quali possono restare attivi in memoria sono 10 (in quanto 10 * 160 = 1600) e per cui quelli fuori memoria sono 12 (22 - 10 = 12).

Esercizio 5

Si consideri una memoria cache 4-set associativa della dimensione di 32 Kbyte con 1024 byte per blocco. La cache è collegata ad una memoria di 1Mbyte indirizzabile per byte. Definire le dimensioni ed il significato delle parti dell'indirizzo della cache e dell'indirizzo della RAM.

Soluzione:

Cache 4-set associativa
32 kbyte

Blocco 1 kbyte

Ram 1 Mbyte

Dimensioni dell'*indirizzo* necessario ad indirizzare tutta la memoria Ram:

dim. Ram 1M parole = 2^{20} 20 bit per l'indirizzo

Dimensione del campo *parola*:

1 KB = 2^{10} 10 bit necessari per costruire il campo Parola

Dimensione del campo *blocco:*

Blocco = dim. Cache / Tipo Set
↓
dim. Cache = parole Cache / parole Blocco

dim. Cache = 32KB / 1 KB
$2^5 * 2^{10} / 2^{10} = 2^{15} / 2^{10} = 2^5$
Blocco = $2^5 / 2^2$ (4-set associativa) = 2^3 3 bit necessari a costruire il campo Blocco

Dimensione del campo *etichetta*:

Etichetta = dim. Ram / Cache Set
↓
dim. Ram = parole Ram / parole Blocco

dim. Ram = 1MB / 1KB
$2^{20} / 2^{10} = 2^{10}$
Etichetta = $2^{10} / 2^3 = 2^7$ 7 bit necessari a costruire il campo Etichetta

ETICHETTA	***BLOCCO***	***PAROLA***
7 bit	3 bit	10 bit

7 + 3 + 10 = 20 Sommati i campi danno le dimensioni dell'indirizzamento.

Dimensioni dell'indirizzo per la Cache:

32 kbyte = $2^5 * 2^{10} / 2^{10} = 2^{15} / 2^{10} = 2^5$ quindi si hanno 15 bit per l'indirizzo

Le dimensioni dei campi *parola* e blocco restano uguali, quindi si riduce la dimensione del campo *etichetta* che diventano:
15 - 10 - 3 = 2 bit

ETICHETTA	***BLOCCO***	***PAROLA***
2 bit	3 bit	10 bit

infatti 10 + 3 + 2 = 15

Esercizio 6

Si consideri un programma che accede in sequenza a tutti gli elementi di un array di 10000 parole (ogni elemento ha dimensione di una parola). Questa operazione di scansione è effettuata all'interno di un ciclo che viene eseguito 10 volte.

Si assuma che le dimensioni di una pagina siano di 1k parole. Il programma occupa due pagine ed il suo *working set* è composto da 10 pagine.

Quanti page fault avvengono durante l'esecuzione del programma considerando che nessuna pagina del programma è residente in memoria al momento della sua esecuzione?

Soluzione:

l codice occupa due pagine.

Parte di dati: 10000 parole (dati) / 1024 parole (dimensione pagina) = 9.76
Quindi servono 10 pagine per caricare tutto il programma.

Codice 1	Codice 2	Dati 1	Dati 2	Dati 3	Dati 4	Dati 5	Dati 6	Dati 7	Dati 8

1^ ciclo: Considerando che la memoria non contiene nessuna pagina del programma,avvengono 12 page fault (2 per il caricamento del programma e 10 per il caricamento dell'array. La mappa della memoria alla fine della scansione è:

Codice 1	Codice 2	Dati 9	Dati 10	Dati 3	Dati 4	Dati 5	Dati 6	Dati 7	Dati 8

2^ - 10^ ciclo: utilizzando l'algoritmo LRU: Le pagine Dati 9 e Dati 10 hanno sostituito le pagine Dati 1 e Dati 2 nel ciclo precedente, pagine che sono necessarie nel secondo ciclo di scansione. Per caricare la memoria, utilizzando LRU, la pagina Dati 1 sostituisce la pagina Dati 3 e la pagina Dati 2 sostituisce la pagina Dati 4. In questo modo non si avrà mai una pagina Dati utile nella memoria.

2^ - 10^ ciclo: utilizzando l'algoritmo FIFO: Nel ciclo di scansione 1^ la pagina Dati 9 ha sostituito la pagina Dati 8 e poi è stata sostituita dalla pagina Dati 10. La mappa della memoria, prima del secondo ciclo, è la seguente:

Codice 1	Codice 2	Dati 1	Dati 2	Dati 3	Dati 4	Dati 5	Dati 6	Dati 7	Dati 10

Con l'algoritmo FIFO l'unica pagina che viene ripetutamente cambiata è l'ultima pagina, in cui verranno caricate rispettivamente le pagine Dati 8,Dati 9, Dati 10.

Concludendo,il numero di page fault è:

1^ ciclo: **12 page fault**, indipendentemente dall'algoritmo utilizzato.
2^-10^ ciclo utilizzando LRU: 10 page_fault * 9 cicli = **90 page fault**.
2^-10^ ciclo utilizzando FIFO: 3 page_fault * 9 cicli = **27 page fault.**

Possiamo dire che,nel caso in cui la dimensione dei dati eccede leggermente la dimensione della memoria allocata, l'algoritmo vincente è FIFO mentre l'algoritmo LRU falisce sempre.

Esercizio 7

Si consideri una memoria cache 4-set associativa della dimensione di 16 Kbyte con 512 byte per blocco. La cache è collegata ad una memoria di 2Mbyte indirizzabile per byte.
Definire le dimensioni ed il significato delle parti dell'indirizzo della cache e dell'indirizzo della RAM.

Soluzione:

Cache 4-set associativa
16 kbyte

Blocco 512 byte

Ram 2 Mbyte

Dimensioni dell'*indirizzo* necessario ad indirizzare tutta la memoria Ram:

dim. Ram 2M parole = $2 * 2^{20} = 2^{21}$ 21 bit per l'indirizzo

Dimensione del campo *parola*:

512 byte = 2^9 9 bit necessari per costruire il campo Parola

Dimensione del campo *blocco:*

Blocco = dim. Cache / Tipo Set
dim. Cache = parole Cache / parole Blocco
dim. Cache = 16KB / 512 byte
$2^4 * 2^{10} / 2^9 = 2^{14} / 2^9 = 2^5$
Blocco = $2^5 / 2^2$ (4-set associativa) = 2^3 3 bit necessari a costruire il campo Blocco

Dimensione del campo *etichetta*:

Etichetta = dim. Ram / Cache Set
dim. Ram = parole Ram / parole Blocco
dim. Ram = 2MB / 512 byte
$2 * 2^{20} / 2^9 = 2^{12}$
Etichetta = $2^{12} / 2^3 = 2^9$ 9 bit necessari a costruire il campo Etichetta

ETICHETTA	***BLOCCO***	***PAROLA***
9 bit	3 bit	9 bit

9 + 3 + 9 = 21 Sommati i campi danno le dimensioni dell'indirizzamento.

Dimensioni dell'indirizzo per la Cache:

16 kbyte = $2^4 * 2^{10} = 2^{14}$ quindi si hanno 14 bit per l'indirizzo

Le dimensioni dei campi *parola* e *blocco* restano uguali, quindi si riduce la dimensione del campo *etichetta* che diventano:
14 - 9 - 3 = 2 bit

ETICHETTA	***BLOCCO***	***PAROLA***
2 bit	3 bit	9 bit

Esercizio 8

Si consideri una CPU dotata di memoria cache di 128K byte con 64 byte per blocco. Questa CPU è collegata ad una memoria RAM da 16M byte indirizzabile per byte. Definire le dimensioni dell'indirizzo necessario a indirizzare tutta la memoria RAM e definire le dimensioni dei campi PAROLA, BLOCCO (o SET) ed ETICHETTA in cui questo indirizzo può essere suddiviso. Definire queste dimensioni per una memoria cache di tipo:

- ad accesso diretto
- 2-set associativa
- completamente associativa

Soluzione:

Cache accesso diretto
2-set associativa
completamente associativa
128 kbyte

Blocco 64 byte

Ram 16 Mbyte

Accesso diretto

Dimensioni dell'*indirizzo* necessario ad indirizzare tutta la memoria Ram:

dim. Ram 16M parole = $2^4 * 2^{20} = 2^{24}$ 24 bit per l'indirizzo

Dimensione del campo *parola*:

64 byte = 2^6 6 bit necessari per costruire il campo Parola

Dimensione del campo *blocco:*

Blocco = dim. Cache / Tipo Set
dim. Cache = parole Cache / parole Blocco
dim. Cache = 128KB / 64 byte
$2^7 * 2^{10} / 2^6 = 2^{17} / 2^6 = 2^{11}$
Siccome si tratta di accesso diretto possiamo tralasciare il set cache in quanto sarebbe $2^0=1$
Quindi: 11 bit necessari a costruire il campo Blocco

Dimensione del campo *etichetta*:

Etichetta = dim. Ram / Cache Set
dim. Ram = parole Ram / parole Blocco
dim. Ram = 16MB / 64 byte
$2^4 * 2^{20} / 2^6 = 2^{24} / 2^6 = 2^{18}$
Etichetta = $2^{18} / 2^{11} = 2^7$ 7 bit necessari a costruire il campo Etichetta

ETICHETTA	*BLOCCO*	*PAROLA*
7 bit	11 bit	6 bit

7 + 11 + 6 = 24 Sommati i campi danno le dimensioni dell'indirizzamento.

2-set Associativa

Dimensione del campo *parola*:

64 byte = 2^6 6 bit necessari per costruire il campo Parola

Dimensione del campo *blocco:*

Blocco = dim. Cache / Tipo Set
dim. Cache = parole Cache / parole Blocco
dim. Cache = 128KB / 64 byte
$2^7 * 2^{10} / 2^6 = 2^{17} / 2^6 = 2^{11}$
Blocco = 2^{11} / 2 (2-set associativa) = 2^{10} 10 bit necessari a costruire il campo Blocco

Dimensione del campo *etichetta*:

Etichetta = dim. Ram / Cache Set
dim. Ram = parole Ram / parole Blocco
dim. Ram = 16MB / 64 byte
$2^4 * 2^{20} / 2^6 = 2^{24} / 2^6 = 2^{18}$
Etichetta = $2^{18} / 2^{10} = 2^8$ 8 bit necessari a costruire il campo Etichetta

ETICHETTA	*BLOCCO*	*PAROLA*
8 bit	10 bit	6 bit

8 + 10 + 6 = 24 Sommati i campi danno le dimensioni dell'indirizzamento.

Completamente Associativa

Dimensione del campo *parola*:

64 byte = 2^6 6 bit necessari per costruire il campo Parola

Il campo *BLOCCO* non esiste perché è una memoria completamente associativa.

Dimensione del campo *etichetta*:

bit indirizzo della Ram - bit parola = Dimensione del campo etichetta

24 - 6 = 18

ETICHETTA	*PAROLA*
18 bit	6 bit

18 + 6 = 24 Sommati i campi danno le dimensioni dell'indirizzamento.

Esercizio 9

Si consideri una memoria cache completamente associativa della dimensione di 64 Kbyte con 1024 byte per blocco. La cache è collegata ad una memoria di 16 Mbyte.
Definire le dimensioni ed il significato delle parti dell'indirizzo della cache e dell'indirizzo della RAM.

Soluzione:

Cache completamente associativa
64 kbyte

Blocco 1024 byte

Ram 16 Mbyte

Dimensioni dell'*indirizzo* necessario ad indirizzare tutta la memoria Ram:

dim. Ram 16M parole = $2^4 * 2^{20} = 2^{24}$ 24 bit per l'indirizzo

Dimensione del campo *parola*:

1024 byte = 2^{10} 10 bit necessari per costruire il campo Parola

Dimensione del campo *etichetta*:
bit indirizzo della Ram - bit parola = Dimensione del campo etichetta

24 - 10 = 14

ETICHETTA	***PAROLA***
14 bit	10 bit

14 + 10 = 24 Sommati i campi danno le dimensioni dell'indirizzamento.

Dimensioni dell'indirizzo per la *Cache*:

64 kbyte = 2^{16} quindi si hanno 16 bit per l'indirizzo

Le dimensioni dei campi *parola* e *blocco* restano uguali, quindi si riduce la dimensione del campo *etichetta* che diventano:
16 – 10 = 6 bit

ETICHETTA	***PAROLA***
6 bit	10 bit

Esercizio 10

Si assuma che un computer con memoria virtuale, strutturata a pagine di 4Kbyte, sia dotato di 64Mbyte di memoria fisica.
Si consideri un programma con un codice di 7.2Kbyte che accede ciclicamente in sequenza a tutti gli elementi di un array di 1000 record in cui ogni campo è composto da 2 numeri interi.
Quale deve essere la dimensione del *working set* perché si abbiano dei *page fault* solamente nella fase di caricamento del programma con esecuzione del primo ciclo di accesso agli elementi dell'array?
Quanti *page fault* si avrebbero durante l'esecuzione del programma ipotizzando che il *working set* abbia una pagina meno di quanto definito nel punto precedente e che il ciclo di accesso venga ripetuto 10 volte?

Soluzione:

Dim pagina	4 kbyte
memoria fisica	64 Mbyte
codice	7,2 kbyte
array	1000 record di cui ogni campo contiene 2 numeri interi

Il codice occupa due pagine.

Parte di dati: 2 (numeri) * 4 (byte) * 1000 (record) = 8000 byte
8000 byte : 4 kbyte = 1,95

La parte di dati occupa quindi 2 pagine.

Codice 1	Codice 2	Dati 1	Dati 2

La dimensione del Working Set perché si abbiano dei page fault solamente nella fase di caricamento del programma con esecuzione del primo ciclo di accesso agli elementi dell'array è di 4 pagine.

Ipotizzando che il working set abbia una pagina in meno di quanto definito nel punto precedente e che il ciclo di accesso venga ripetuto 10 volte, il numero di page fault che si avrebbero durante l'esecuzione del programma è 22, infatti:

Codice 1	Codice 2	Dati 1

1^ ciclo:	4 page fault
2^ - 10^ ciclo:	2 page fault * 9 cicli = 18 page fault
totale:	4 + 18 = 22 page fault

Esercizio 11

Si consideri una memoria cache 2-set associativa della dimensione di 64 Kbyte con 1024 byte per blocco. La cache è collegata ad una memoria di 4Mbyte indirizzabile per byte. Definire le dimensioni ed il significato delle parti dell'indirizzo della cache e dell'indirizzo della RAM.

Soluzione:

Cache 2-set associativa
64 kbyte

Blocco 1 Kbyte

Ram 4 Mbyte

Dimensioni dell'*indirizzo* necessario ad indirizzare tutta la memoria Ram:

dim. Ram 4M parole = $2^2 * 2^{20} = 2^{22}$ 22 bit per l'indirizzo

Dimensione del campo *parola*:

64 byte = 2^6 6 bit necessari per costruire il campo Parola

Dimensione del campo *blocco:*

Blocco = dim. Cache / Tipo Set
dim. Cache = parole Cache / parole Blocco
dim. Cache = 64Kbyte / 1 Kbyte
$2^6 * 2^{10} / 2^{10} = 2^{16} / 2^{10} = 2^6$
Blocco = $2^6 / 2$ (2-set associativa) = 2^5 5 bit necessari a costruire il campo Blocco

Dimensione del campo *etichetta*:

Etichetta = dim. Ram / Cache Set
dim. Ram = parole Ram / parole Blocco
dim. Ram = 4Mbyte / 1 Kbyte
$2^2 * 2^{20} / 2^{10} = 2^{22} / 2^{10} = 2^{12}$
Etichetta = $2^{12} / 2^5 = 2^7$ 7 bit necessari a costruire il campo Etichetta

ETICHETTA	***BLOCCO***	***PAROLA***
7 bit	5 bit	10 bit

7 + 5 + 10 = 22 Sommati i campi danno le dimensioni dell'indirizzamento.

Dimensioni dell'indirizzo per la *Cache*:

64 Kbyte = $2^6 * 2^{10} = 2^{16}$ quindi sono necessari 16 bit per l'indirizzo

Le dimensioni dei campi *parola* e *set* restano uguali, quindi si riduce la dimensione del campo *etichetta* che diventano:

16 - 5 -10 = 1 bit

ETICHETTA	***BLOCCO***	***PAROLA***
1 bit	5 bit	10 bit

Esercizio 12

Si consideri una memoria cache 4-set associativa della dimensione di 32 Kbyte con 1024 byte per blocco. La cache è collegata ad una memoria di 4Mbyte indirizzabile per byte.
Definire le dimensioni ed il significato delle parti dell'indirizzo della cache e dell'indirizzo della RAM.

Soluzione:

Cache 4-set associativa
32 kbyte

Blocco 1024 byte

Ram 4 Mbyte

Dimensioni dell'*indirizzo* necessario ad indirizzare tutta la memoria Ram:

dim. Ram 4M parole = $2^2 * 2^{20} = 2^{22}$ 22 bit per l'indirizzo

Dimensione del campo *parola*:

1024 byte per blocco
$1024 = 2^{10}$ 10 bit necessari per costruire il campo Parola

Dimensione del campo *blocco:*

Blocco = dim. Cache / Tipo Set
dim. Cache = parole Cache / parole Blocco
dim. Cache = 32 Kbyte / 1 Kbyte
$2^5 * 2^{10} / 2^{10} = 2^{15} / 2^{10} = 2^5$
Blocco = $2^5 / 2^2$ (4-set associativa) = 2^3 3 bit necessari a costruire il campo Blocco

Dimensione del campo *etichetta*:

Etichetta = dim. Ram / Cache Set
dim. Ram = parole Ram / parole Blocco
dim. Ram = 4M parole / 1 Kbyte
$4M / 1Kbyte = 2^{22} / 2^{10} = 2^{12}$
Etichetta = $2^{12} / 2^3 = 2^9$ 9 bit necessari a costruire il campo Etichetta

ETICHETTA	*BLOCCO*	*PAROLA*
9 bit	3 bit	10 bit

9 + 3 + 10 = 22 Sommati i campi danno le dimensioni dell'indirizzamento.

Dimensioni dell'indirizzo per la *Cache*:

32 Kbyte = $2^5 * 2^{10} = 2^{15}$

Le dimensioni dei campi *parola* e *set* restano uguali, quindi si riduce la dimensione del campo *etichetta* che diventano:

15 - 3 -10 = 2 bit

ETICHETTA	***BLOCCO***	***PAROLA***
2 bit	3 bit	10 bit

Esercizio 13

Si consideri una CPU dotata di memoria cache 2-associativa di 8K parole con 64 parole per blocco. Questa CPU è collegata ad una memoria RAM da 8M parole. Definire le dimensioni dell'indirizzo necessario a indirizzare tutta la memoria RAM e definire le dimensioni dei campi PAROLA, BLOCCO ed ETICHETTA in cui questo indirizzo può essere suddiviso. Motivare la risposta con un opportuno schema.

Soluzione:

Cache 2-set associativa
8k parole

Blocco 64 parole

Ram 8M parole

Dimensioni dell'*indirizzo* necessario ad indirizzare tutta la memoria Ram:

dim. Ram 8M parole = $2^3 * 2^{20} = 2^{23}$ 23 bit per l'indirizzo

Dimensione del campo *parola*:

64 parole per blocco
$64 = 2^6$ 6 bit necessari per costruire il campo Parola

Dimensione del campo *blocco:*

Blocco = dim. Cache / Tipo Set
dim. Cache = parole Cache / parole Blocco
dim. Cache = 8 Kbyte / 64
$2^3 * 2^{10} / 2^6 = 2^{13} / 2^6 = 2^7$
Blocco = 2^7 / 2 (2-set associativa) = 2^6 6 bit necessari a costruire il campo Blocco

Dimensione del campo *etichetta*:

Etichetta = dim. Ram / Cache Set
dim. Ram = parole Ram / parole Blocco
dim. Ram = 8M / 64
$8M / 64 = 2^3 * 2^{20} / 2^6 = 2^{23} / 2^6 = 2^{17}$
Etichetta = $2^{17} / 2^6 = 2^{11}$ 11 bit necessari a costruire il campo Etichetta

ETICHETTA	***BLOCCO***	***PAROLA***
11 bit	6 bit	6 bit

11 + 6 + 6 = 23 Sommati i campi danno le dimensioni dell'indirizzamento

Esercizio 14

NB: questo esercizio non può essere svolto con la notazione moderna in quanto il numero del set non è esprimibile in potenza di 2. Occorre perciò considerare il numero del set come $2^{n\,set}$ *nei calcoli, non come n-set (come per la risoluzione degli altri esercizi).*

Si consideri una memoria cache 3-set associativa della dimensione di 64 Kbyte con 1024 byte per blocco. La cache è collegata ad una memoria di 4 Mbyte indirizzabile per byte. Definire le dimensioni ed il significato delle parti dell'indirizzo della cache e dell'indirizzo della RAM.

Soluzione:

Cache 3-set associativa
64 kbyte

Blocco 1024 byte

Ram 4 Mbyte

Dimensioni dell'*indirizzo* necessario ad indirizzare tutta la memoria Ram:

dim. Ram 4M parole = $2^2 * 2^{20} = 2^{22}$ 22 bit per l'indirizzo

Dimensione del campo *parola*:

1 Kbyte
1024 byte = 2^{10} 10 bit necessari per costruire il campo Parola

Dimensione del campo *blocco:*

Blocco = dim. Cache / Tipo Set
dim. Cache = parole Cache / parole Blocco
dim. Cache = 64 Kbyte / 1 Kbyte
$2^6 * 2^{10} / 2^{10} = 2^{16} / 2^{10} = 2^6$
Questo risultato va ora diviso per $2^{n\,set}$ quindi:
Blocco = $2^6 / 2^3$ (3-set associativa) = 2^3 3 bit necessari a costruire il campo Blocco

Dimensione del campo *etichetta*:

bit indirizzo della Ram - bit parola - bit set = Dimensione del campo etichetta
quindi
22 - 10 - 3 = 9

ETICHETTA	***BLOCCO***	***PAROLA***
10 bit	3 bit	9 bit

10 + 3 + 9 = 22 Sommati i campi danno le dimensioni dell'indirizzamento

Dimensioni dell'indirizzo per la *Cache*:

64 Kbyte = $2^6 * 2^{10} = 21^6$ quindi si hanno 16 bit per l'indirizzo

Le dimensioni dei campi *parola* e *set* restano uguali, quindi si riduce la dimensione del campo *etichetta* che diventano:

16 - 3 -10 = 3 bit

ETICHETTA	***BLOCCO***	***PAROLA***
3 bit	3 bit	10 bit

Esercizio 15

NB: questo esercizio non può essere svolto con la notazione moderna in quanto il numero del set non è esprimibile in potenza di 2. Occorre perciò considerare il numero del set come $2^{n\ set}$ nei calcoli, non come n-set(come per la risoluzione degli altri esercizi).

Si consideri una CPU dotata di memoria cache 3-set associativa di 8K parole con 64 parole per blocco. Questa CPU è collegata ad una memoria RAM da 8M parole. Definire le dimensioni dell'indirizzo necessario a indirizzare tutta la memoria RAM e definire le dimensioni dei campi PAROLA, BLOCCO ed ETICHETTA in cui questo indirizzo può essere suddiviso. Motivare la risposta con un opportuno schema.

Soluzione:

Cache 3-set associativa
8k parole

Blocco 64 parole

Ram 8M parole

Dimensioni dell'*indirizzo* necessario ad indirizzare tutta la memoria Ram:

dim. Ram 8M parole = $2^3 * 2^{20} = 2^{23}$ 23 bit per l'indirizzo

Dimensione del campo *parola*:

64 parole per blocco
$64 = 2^6$ 6 bit necessari per costruire il campo Parola

Dimensione del campo *blocco:*

Blocco = dim. Cache / Tipo Set
dim. Cache = parole Cache / parole Blocco
dim. Cache = 8 Kbyte / 64
$2^3 * 2^{10} / 2^6 = 2^{13} / 2^6 = 2^7$
Questo risultato va ora diviso per $2^{n\ set}$ quindi:
Blocco = $2^7 / 2^3$ (3-set associativa) = 2^4 4 bit necessari a costruire il campo Blocco

Dimensione del campo *etichetta*:
bit indirizzo della Ram - bit parola - bit set = Dimensione del campo etichetta
quindi

23 - 4 - 6 = 13

ETICHETTA	***BLOCCO***	***PAROLA***
13 bit	4 bit	6 bit

13 + 4 + 6 = 23 Sommati i campi danno le dimensioni dell'indirizzamento

Esercizio 16

Disponiamo di un elaboratore con cache separata per istruzioni e dati, il cui costo di accesso è 1 ciclo macchina per entrambe. Il costo di accesso (esclusivamente) alla memoria RAM invece è di 19 cicli macchina. Il tasso di cache hit stimato per le istruzioni è 90%, mentre il tasso di cache hit stimato per i dati è 80%. Si consideri ora un programma composto per il 20% da istruzioni di lettura dalla memoria e per lo 0% da istruzioni di scrittura in memoria. Quale risulta essere per un simile programma lo speedup dovuto alla presenza della cache rispetto ad un accesso diretto alla memoria?

Calcolo il tempo di esecuzione del programma nell'elaboratore privo di cache:

t_no_cache = t_istruzione_no_cache + 20/100 * t_dato_no_cache = 19 + 20/100 * 19 = 6/5 * 19

in quanto dobbiamo sempre prelevare l'istruzione dalla memoria, mentre per il 20% delle volte dobbiamo in aggiunta prelevare il dato dalla memoria.

Calcolo il tempo di esecuzione del programma nell'elaboratore con cache:

t_cache = t_istruzione_cache + 20/100 * t_dato_cache

t_istruzione_cache = 90/100 * 1 + 10/100 * (19 + 1 +1) = 9/10 + 21/10 = 3

in quanto in caso di hit (90% per le istruzioni) è richiesto un solo ciclo, ma in caso di miss dobbiamo accedere alla cache per tentare di prelevare l'istruzione (1), accedere alla memoria per prelevare l'istruzione (19) e di nuovo accedere alla cache (1) per prelevare l'istruzione con successo.

t_dato_cache = 80/100 * 1 + 20/100 * (19 + 1 + 1) = 8/10 + 42/10 = 5

Il caso del dato è identico al caso delle istruzioni ma con hit rate 80%.

Segue che t_cache = 3 + 2/10 * 5 = 4

Lo speed up è il rapporto tra il tempo di esecuzione del programma senza cache e in presenza di cache, quindi si ha che:

Speed up = 6/5 * 19 / 4 = 57/10 = 5.7

Il che implica che in questo caso la cache velocizza di quasi sei volte la CPU.

Pipeline: regole ed accorgimenti

Questo tipo di esercizi consiste nell'elencare le varie fasi di Fetch, Decode, Execute, Memory e Write di una istruzione Assembly in base alle situazioni proposte e in base alle dipendenze di altre istruzioni. Nella migliore delle ipotesi, un'istruzione parte con la sequenza F, D, E, M, W, la successiva parte invece con la fase di Fetch durante l'ultimo ciclo della fase di Decode dell'istruzione precedente (in questo caso la fase di Decode è unica).

Graficamente:

Istruzione 1	F	D	E	M	W	
Istruzione 2		F	D	E	M	W

Qualora invece un'istruzione dipenda da quella precedente, la sua fase di Fetch inizia all'istante dello stato di Decode della funzione precedente, la fase di Decode perdura per tutto il tempo degli altri stadi dell'istruzione precedente (quindi Decode, Execute, Memory, Write) più una fase aggiuntiva dopo la Write. La fase di Execute inizia dopo la fase aggiuntiva di Decode.

Graficamente:

Istruzione 1	F	D	E	M	W				
Istruzione 2		F	D	D	D	D	E	M	W

Qualora invece ci siano 3 istruzioni e l'ultima dipendesse dalla prima ma non dalla seconda, lo stato in più descritto precedentemente deve essere "in più" rispetto alla prima istruzione (cioè quella da cui dipende l'istruzione considerata).

Graficamente:

Istruzione 1	F	D	E	M	W				
Istruzione 2		F	D	E	M	W			
Istruzione 3			F	D	D	D	E	M	W

Innanzitutto, è buona cosa leggere bene ogni istruzione elencata per capire le dipendenze. Per capire quali sono le istruzioni dipendenti tra loro, basta vedere quali registri vengono utilizzati: se due istruzioni usano lo stesso registro, bisogna sicuramente aspettare la fase di Write per iniziare la fase di Execute (come già descritto prima).

Negli esercizi seguenti sono stati inseriti i flag #yes e #no per indicare se il salto avviene oppure no. Si può provare a modificarli.
#yes → indica che il salto avviene
#no → indica che il salto non avviene

Esercizio 1

Si consideri una CPU con una pipeline a 5 stadi (F, D, E, M, W). Si riporti nel seguente diagramma, per ogni istruzione, lo stadio della pipeline coinvolto in ogni istante di clock. Si ipotizzi la pipeline vuota al tempo 1. Si ipotizzi che il salto avvenga.

Soluzione

CLOCK/ISTRUZIONE	1	2	3	4	5	6	7	8	9	10	11
`loop: addl %eax, %ebx`	F	D	E	M	W			F	D	D	D
`movl $4, %ecx`		F	D	E	M	W					
`subl %ebx, %ecx`			F	D	D	D	D	E	M	W	
`jz loop #yes`							F	D	E	M	W

Le prime due istruzioni vengono eseguite normalmente, poiché non ci sono dipendenze. La terza istruzione dipende dalla seconda perché hanno un registro in comune, dunque è necessario che l'istruzione `subl` prolunghi la fase di decode fino all'ultimo ciclo di `movl` più uno. Infine, la quarta istruzione non dipende dalle altre e può essere eseguita normalmente. Dato che ipotizzo che il salto avvienga, all'ottavo ciclo di clock ricomincia la prima istruzione.

Esercizio 2

Si consideri una CPU con una pipeline a 5 stadi (F, D, E, M, W). Si riporti nel seguente diagramma, per ogni istruzione, lo stadio della pipeline coinvolto in ogni istante di clock. Si ipotizzi la pipeline vuota al tempo 1.

Soluzione

CLOCK/ISTRUZIONE	1	2	3	4	5	6	7	8	9	10	11	12	13
`inizio: mull %ebx`	F	D	E	M	W								
`movl %ecx, %edx`		F	D	E	M	W							
`cmpl %ebx, 0x86FF`			F	D	D	D	E	M	W				
`jnz inizio #no`						F	D	D	D	D	E	M	W

Le prime due istruzioni vengono eseguite normalmente, poiché non ci sono dipendenze. La terza istruzione dipende dalla prima perché hanno un registro in comune, dunque è necessario che l'istruzione `cmpl` prolunghi la fase di decode fino all'ultimo ciclo di `mull` più uno. Infine, la quarta istruzione dipende dalla terza perché il salto deve attendere che si sappia se la comparazione è vera o falsa, affinché si possa capire se il salto deve avvenire oppure no. Di conseguenza anche qui l'istruzione `jnz` deve prolungare la fase di decode fino all'ultimo ciclo di `cmpl` più uno.

Esercizio 3

Si consideri una CPU con una pipeline a 5 stadi (F, D, E, M, W). Si riporti nel seguente diagramma, per ogni istruzione, lo stadio della pipeline coinvolto in ogni istante di clock. Si ipotizzi la pipeline vuota al tempo 1.

Soluzione

CLOCK/ISTRUZIONE	1	2	3	4	5	6	7	8	9	10	11
`START: cmpl %eax, %ebx`	F	D	E	M	W						
`jz START #no`		F	D	D	D	D	E	M	W		
`subl %ebx, %ecx`						F	D	E	M	W	
`movl %edx, %eax`							F	D	E	M	W

La prima istruzione viene eseguita normalmente. La seconda ha una dipendenza con la prima perché il salto deve conoscere il risultato della comparazione. Poiché il salto non avviene, posso eseguire tranquillamente la terza istruzione. Inoltre, quest'ultima dipende dalla prima, ma non è necessaria alcuna attesa poiché la prima istruzione finisce prima che la terza cominci. Infine, la quarta istruzione non ha dipendenze.

Esercizio 4

Si consideri una CPU con una pipeline a 5 stadi (F, D, E, M, W). Si riporti nel seguente diagramma, per ogni istruzione, lo stadio della pipeline coinvolto in ogni istante di clock. Si ipotizzi la pipeline vuota al tempo 1.

Soluzione

CLOCK/ISTRUZIONE	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`loop: addl %eax, %ebx`	F	D	E	M	W										
`movl %edx, %ecx`		F	D	E	M	W									
`subl %ebx, %ecx`			F	D	D	D	D	E	M	W					
`jz loop #no`							F	D	D	D	D	E	M	W	
`movl %ecx, %edx`											F	D	E	M	W

Le prime due istruzioni vengono eseguite normalmente, poiché non ci sono dipendenze. La terza dipende sia dalla prima che dalla seconda perché ha un registro in comune con ciascuna di esse, quindi è necessario prolungare la fase di decode di `subl` fino all'ultimo ciclo di `movl` più uno. La quarta istruzione non ha dipendenze, ma dato che non faccio alcuna previsione deve aspettare che la terza

istruzione termini quindi prolunga la fase di decode del salto fino all'ultimo ciclo di `subl` più uno. Infine, dato che il salto non avviene, la quinta viene eseguita normalmente.

Esercizio 5

Si consideri una CPU con una pipeline a 5 stadi (F, D, E, M, W). Si riporti nel seguente diagramma, per ogni istruzione, lo stadio della pipeline coinvolto in ogni istante di clock. Si ipotizzi che la pipeline sia vuota al tempo 1 e che `jz` faccia riferimento all'istruzione `subl`.

Soluzione

CLOCK/ISTRUZIONE	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
`loop: addl %eax, %ebx`	F	D	E	M	W												
`movl %ebx, %ecx`		F	D	D	D	D	E	M	W								
`subl %eax, %ecx`						F	D	D	D	D	E	M	W				
`jz loop #no`										F	D	D	D	D	E	M	W

La prima istruzione viene eseguita normalmente. La seconda ha una dipendenza con la prima, dunque è necessario che l'istruzione `movl` prolunghi la fase di decode fino all'ultimo ciclo di `addl` più uno. La terza istruzione ha una dipendenza con la seconda, dunque è necessario che anche l'istruzione `subl` prolunghi la fase di decode fino all'ultimo ciclo di `movl` più uno. Infine, il salto fa riferimento all'istruzione `subl`, quindi la quarta istruzione dipende dalla terza e si deve prolungare con uguale criterio la fase di decode.

Esercizio 6

Si consideri una CPU con una pipeline a 5 stadi (F, D, E, M, W). Si riporti nel seguente diagramma, per ogni istruzione, lo stadio della pipeline coinvolto in ogni istante di clock. Si ipotizzi la pipeline vuota al tempo 1 e si facciano le opportune ipotesi sul salto condizionale.

Soluzione

Suppongo che il salto non avvenga.

CLOCK/ISTRUZIONE	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`init: movl %ecx, %edx`	F	D	E	M	W										
`addl $4, %ebx`		F	D	E	M	W									
`cmpl 0x319FA, %ebx`			F	D	D	D	D	E	M	W					
`jnz init #no`							F	D	D	D	D	E	M	W	
`addl %eax, %ecx`											F	D	E	M	W

Le prime due istruzioni vengono eseguite normalmente, poiché non ci sono dipendenze. La terza dipende dalla seconda, quindi è necessario prolungare la fase di decode di `cmpl` fino all'ultimo ciclo di `addl` più uno. La quarta istruzione dipende dalla terza e quindi anche qui si deve prolungare allo stesso modo la fase di decode. Infine, poiché ipotizzo che il salto non avvenga, posso passare alla quinta istruzione, la quale non ha dipendenze e viene eseguita normalmente.

Suppongo che il salto avvenga.

CLOCK/ISTRUZIONE	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
`init: movl %ecx, %edx`	F	D	E	M	W						F	D	E	M	W			
`addl $4, %ebx`		F	D	E	M	W						F	D	E	M	W		
`cmpl 0x319FA, %ebx`			F	D	D	D	D	E	M	W			F	D	D	D	D	E
`jnz init #yes`							F	D	D	D	D	E	M	W			F	D
`addl %eax, %ecx`																		

Le prime due istruzioni vengono eseguite normalmente, poiché non ci sono dipendenze. La terza dipende dalla seconda perché hanno un registro in comune, quindi è necessario prolungare la fase di decode di `cmpl` fino all'ultimo ciclo di `addl` più uno. La quarta istruzione dipende dalla terza e quindi anche qui si deve prolungare allo stesso modo la fase di decode. Infine, poiché ipotizzo che il salto avvenga, la quinta istruzione non viene eseguita, mentre ricominciano le istruzioni dell'etichetta `init`.

Domande di teoria

Domanda 1

Una CPU con una pipeline a 2 stadi viene sostituita con una CPU con una pipeline a 4 stadi. Se il tempo totale di esecuzione di una singola istruzione è rimasto invariato, qual è il minimo ed il massimo incremento delle prestazioni che si può attendere?

Soluzione:

Una pipeline ad n stadi è in grado di aumentare le prestazioni idealmente di n volte e di raggiungere lo scopo di riuscire ad eseguire una istruzione a ciclo di clock. Quindi se da una pipeline a due stadi si passa ad una a quattro stadi, essa può aumentare idealmente le prestazioni di due volte. Il minimo incremento delle prestazioni che si può ottenere è di una volta.

L'aumento degli stadi nella pipeline aumenta anche la probabilità di stallo della CPU. In condizione di istruzione di salto, nel caso in cui l'algoritmo di predizione dei salti fallisca, dato che più istruzioni vengono eseguite in parallelo, si annulla il vantaggio della pipeline con la perdita delle istruzioni fino a quel momento erroneamente elaborate, è quindi possibile che non si ottenga alcun incremento in quanto aumentando gli stadi della pipeline aumentano le condizioni di criticità, quindi si potrebbero formare delle "bolle" che manderebbero in stallo la CPU.

Domanda 2

Descrivere il meccanismo e l'utilità della *predizione dei salti*.

Soluzione:

I salti interrompono il flusso della pipeline e per minimizzare il numero di interruzioni è necessario uno schema di *brach prediction*.
I salti sono molto frequenti, in media uno ogni 6 istruzioni; nei processori superscalari che eseguono anche 4 istruzioni per ciclo di clock, la *predizione dei salti* è molto importante.
Molti schemi di *brach prediction* hanno un algoritmo che tiene traccia del comportamento di un certo salto l'ultima volta che è stato eseguito.
Infatti sese è stato eseguito un salto nella precedente istruzione, si farà l'ipotesi che il salto avvenga nuovamente.
La pipeline ora contiene un'istruzione di salto condizionato e altre istruzioni successive, ma non si sa ancora se tali istruzioni serviranno o meno. Se il salto verrà eseguito, l'esecuzione del programma potrà continuare tranquillamente, se invece non verrà eseguito, tutte le istruzioni caricate nella pipeline dovranno essere eliminate.

La *predizione dei salti* si divide in due tipi: quella statica e quella dinamica.

Predizione STATICA (*compile-time*): è realizzata dal compilatore.
La parola di codice operativo dell'istruzione indica all'unità di prelievo delle istruzioni se deve prevedere che il salto sia

effettuato o meno. Il risultato della predizione è lo stesso ogni volta che si incontra una data istruzione di salto.

Predizione DINAMICA(*run-time*): L'hardware del processore determina la probabilità che un salto venga eseguito ogni volta che si incontra una certa istruzione. Ciò può essere ottenuto mantenendo traccia del risultato della decisione di salto l'ultima volta che tale istruzione è stata eseguita e ipotizzando che la decisione nella presente istanza possa essere la stessa.

Domanda 3

• Quali sono le motivazioni che fanno preferire la realizzazione di unità di controllo cablate rispetto a quelle microprogrammate.
• Definire lo schema di un controllore cablato indicando i segnali utilizzati e la funzione dei blocchi presenti.

Soluzione:

Le unità di controllo cablate sono pilotate da un segnale di clock. Ogni stato del contatore corrisponde ad uno dei passi dell'istruzione in esecuzione. Il tempo di clock minimo è il t_{max} di propagazione della rete combinatoria nel caso peggiore, quindi le unità di controllo cablate vantano una buona velocità.

Le unità di controllo microprogrammate sono invece molto più lente, per prelevare le istruzioni successive da eseguire devono continuamente accedere alla memoria.
Schema del controllore cablato:

I segnali di controllo richiesti sono univocamente determinati da:

- contenuto del contatore dei passi di controllo
- contenuto del registro delle istruzioni
- contenuto del codice di condizione e di altri flag di stato che rappresentano gli stati di diverse parti della CPU e delle linee di controllo legate ad esse.

Inoltre per ogni istruzione caricata in IR, una sola delle linee di uscita del decodificatore delle istruzioni è posta a a 1, le altre sono poste a 0.

Domanda 4

Elencare le ottimizzazioni che devono essere eseguite sull'architettura di un calcolatore per raggiungere l'obiettivo di avere, per la maggioranza delle istruzioni, CPI = 1

Soluzione:

Ottimizzazioni per ottenere $CPI_{medio} = 1$

1. Architettura a tre bus
2. Incremento PC separato
3. Cache
4. No operazioni su memoria

5. Cache dati separata dalla cache del programma
6. Alu in cui tutte le operazioni vengono eseguite in un ciclo di clock
7. Pipeline

Domanda 5

Si descriva il meccanismo di interrupt.

Soluzione:

Un interrupt è un meccanismo che permette, ad un avento esterno al calcolatore, di provocare il trasferimento del controllo da un programma ad un altro, attraverso la ISR (Interrupt Service Routine) seguendo il "Principio di trasparenza", ossia il programma o il processo in esecuzione non deve ricevere interruzioni o modifiche dall'interrupt, lasciando intatto il suo flusso di esecuzione.

Quando il processore riceve un interrupt, prima termina l'esecuzione in corso (per il "principio di trasparenza", un processo non deve essere interrotto, perciò si attende prima che la prossima fase di Fetch abbia luogo), poi carica nel PC l'indirizzo della prima istruzione della procedura di risposta interrupt.

2 Quando si verifica un interrupt il contenuto corrente del PC (che punta all'istruzione i+1) dev'essere memorizzato in una locazione temporanea.
Devono essere salvate anche tutte le informazioni che possono essere modificate durante la risposta interrupt.

Terminato l'interrupt, per il "principio di trasparenza", devono essere ripristinati tutti i registri modificati durante la fase di interrupt.

Domanda 6

Quali sono le motivazioni che spingono i microprocessori moderni a essere dotati di una pipeline.

Soluzione:

Lo scopo di adottare le pipeline nei processori moderni è quello di ridurre il più possibile il CPI_{medio} e di ottenere $CPI_{medio} = 1$.
Le pipeline sono delle strutture interne alla CPU; esse sono dotate di più stadi (come ad esempio Fetch, Decode, Execute e WriteBack) e permettono di ridurre il CPI aumentando l'IPS.
Per avere dei netti miglioramenti in quantità di tempo è indispensabile cercare di fare entrare la CPU in stallo il meno possibile, di tenerla quindi sempre a regime.
Per fare questo, un meccanismo indispensabile è la *predizione dei salti* (*branch prediction*).
Teoricamente se una pipeline ha *n* stadi, aumenta l'efficienza di *n* volte.
In realtà non è così poiché aumenta la probabilità di dipendenza fra i dati, motivo per cui i processori moderni cercano di avere molti stadi ma non eccessivi, ed un sistema di *branch prediction* il più affidabile possibile.

Domanda 7

Quali sono i vantaggi e gli svantaggi delle memorie completamente associative rispetto alle memoria non associative.

Soluzione:

Memorie *completamente associative*:

Vantaggi	***Svantaggi***
Non ci sono blocchi di Cache predefiniti ove memorizzare i blocchi di Ram, quindi la Cache è utilizzabile per intero ed i conflitti di blocco compaiono solo quando la Cache è piena. Questo implica massima libertà nella scelta di dove posizionare in Cache il blocco di Ram.	E' necessario un algoritmo di ricerca per trovare i blocchi ricercati all'interno della Cache.
Lo spazio della Cache è usato in modo molto efficiente.	Costo elevato.

Memorie *non associative*:

Vantaggi	***Svantaggi***
Facile da realizzarsi.	Non molto flessibile.
Unico blocco in Cache ove memorizzare uno specifico blocco della Ram, quindi è sufficiente confrontare i bit più significativi dell'indirizzo con l'etichetta associata al blocco della Cache per sapere se il blocco è presente in Cache oppure no.	

Domanda 8

Quali sono i vantaggi di una *cache* a indirizzamento diretto?

Soluzione:

Nella cache ad accesso diretto il set è composto da un solo blocco e ogni blocco della RAM ha un unico posto in cache ove essere memorizzato.
I bit più significativi dell'indirizzo vengono confrontati con l'etichetta associata al blocco della Cache: se coincidono, la parola cercata si trova in quel blocco della Cache, altrimenti non c'è altra posizione ove cercarla e bisogna caricarla dalla Ram.
Il metodo di indirizzamento diretto è facile da realizzarsi (anche se non è molto flessibile), inoltre, qualora un programma facesse riferimento sempre e solo a specifiche zone di memoria della RAM, l'intera RAM non verrebbe utilizzata, rallentando notevolmente l'esecuzione del programma.