Cognome e Nome: _____ **Matricola:** _____ **Mod 2:** ☐ **Mod 1+2:** ☐

Architettura degli Elaboratori (modulo II)

(Compito 1 Settembre 2021)

Esercizio 1

Individuare le dipendenze RAW (Read After Write) nel seguente codice MIPS:

```
        loop:
1.         lw    $2, 25($27)
2.         add   $1, $2, $3
3.         or    $4, $2, $6
4.         add   $27, $6, $27
5.         or    $8, $4, $28
6.         beq   $2, $0, loop
```

Considerare una CPU MIPS a 5 stadi **senza** forwarding, e **con** register file speciale (che nella prima parte del ciclo scrive e nella seconda parte del ciclo legge i registri). La CPU in questione **non** ha hazard detection unit, per cui la gestione delle dipendenze è affidata al software (necessario inserire opportune `nop`).
Il branch viene anch'esso risolto a software con il delayed branch; ovvero, in assenza di ottimizzazioni, inserendo una istruzione `nop` di seguito al branch.
Inserire le `nop` opportune nel codice affinché le dipendenze sui dati (RAW) siano rispettate, e verificare il funzionamento del codice tramite un diagramma temporale di funzionamento, specificando come le dipendenze sono risolte.
Ottimizzare il codice, spostando le istruzioni in modo opportuno, e producendo il nuovo diagramma temporale di esecuzione.

Soluzione

Le dipendenze RAW sono le seguenti:

```
1->2 ($2)
1->3 ($2)
1->6 ($2)
3->5 ($4)
```

Il codice con le `nop` è il seguente:

```
        loop:
1.          lw    $2, 25($27)
            nop
            nop
2.          add   $1, $2, $3
3.          or    $4, $2, $6
4.          add   $27, $6, $27
            nop
5.          or    $8, $4, $28
6.          beq   $2, $0, loop
            nop
```

Il corrispondente diagramma è il seguente:

```
         1    2    3    4    5    6    7    8    9    10   11   12   13   14
1.     | IF | ID | EX | ME | WB |
nop         | IF | ID | EX | ME | WB |
nop              | IF | ID | EX | ME | WB |
2.                    | IF | ID | EX | ME | WB |
3.                         | IF | ID | EX | ME | WB |
4.                              | IF | ID | EX | ME | WB |
nop                                  | IF | ID | EX | ME | WB |
5.                                        | IF | ID | EX | ME | WB |
6.                                             | IF | ID | EX | ME | WB |
nop                                                 | IF | ID | EX | ME | WB |
```

La dipendenza `1->2` è risolta dal register file speciale al ciclo 5. La dipendenza `3->5` è risolta dal register file speciale al ciclo 9. Le altre dipendenze sono risolte dal fatto che l'ID delle istruzioni 3 e 6 avvengono tutte dopo il ciclo 5, durante il quale si verifica il WB dell'istruzione 1.

Il codice ottimizzato è il seguente:

```
        loop:
1.         lw    $2, 25($27)
4.         add   $27, $6, $27
3.         or    $4, $2, $6
2.         add   $1, $2, $3
6.         beq   $2, $0, loop
5.         or    $8, $4, $28
```

Si noti che tale modifica non cambia la semantica del programma, in quanto continua a valere l'*ordine di esecuzione* imposto dalle dipendenze tra le istruzioni.
In altri termini, sono preservati gli ordinamenti parziali: `1->2, 1->3, 1->6, 3->5`.
Il nuovo diagramma è il seguente, dove è ancora necessario inserire una `nop` per rispettare la dipendenza `1->2`:

```
         1    2    3    4    5    6    7    8    9    10   11
1.     | IF | ID | EX | ME | WB |
4.          | IF | ID | EX | ME | WB |
nop              | IF | ID | EX | ME | WB |
3.                    | IF | ID | EX | ME | WB |
2.                         | IF | ID | EX | ME | WB |
6.                              | IF | ID | EX | ME | WB |
5.                                   | IF | ID | EX | ME | WB |
```

La dipendenza `1->3` è risolta dal register file speciale al ciclo 5. Le altre dipendenze `1->2`, `1->6` sono risolte dal fatto che l'ID delle istruzioni 2 e 6 avvengono tutte dopo il ciclo 5, durante il quale si verifica il WB dell'istruzione 1. Infine la dipendenza `3->5` viene risolta grazie al register file speciale.
Si noti che poiché l'istruzione 6 (`beq`) non dipende dall'istruzione 5, quest'ultima viene spostata nel delayed slot del branch.

Esercizio 2

Dato un sistema di memoria virtuale paginata con pagine da 4 KB, considerare una TLB a 8 vie, con 32 set, di dimensione totale: 1 KB.
Ogni entry della TLB contiene 2 bit aggiuntivi: (V,D), mentre l'indirizzo fisico è di 27 b.
Rispondere alle seguenti domande:

1. calcolare la dimensione di TAG e PPN, che fanno parte di ciascuna entry della TLB: *(V,D,TAG,PPN)*
2. calcolare la dimensione dell'indirizzo virtuale.

Il sistema di memoria precedente include anche una cache di primo livello, associativa a 4 vie, con blocchi da 16 B, e dimensione della cache (solo parte dati) di 32 KB.

1. Calcolare la TAG della cache.

Soluzione

Size_entryTLB = $\frac{1KB}{n_{vie} * n_{set}} = \frac{2_{10}}{2^3 * 2^5} = 4$ B.
OFFSET = $\log_2 sz_{page} = \log_2 2^{12} = 12$ b.
PPN = IND_FISICO - OFFSET = 27 - 12 = 15 b.
$\text{INDEX}_{TLB} = \log_2 n_{set} = \log_2 2^5 = 5$ b.
TAG_{TLB} = sz_entry - 2 - PPN = 32 - 2 - 15 = 15 b.
VPN = TAG + INDEX = 5 + 15 = 20 b
IND_VIRTUALE = VPN + OFFSET = 20 + 12 = 32 b.

n_blocchi = $\frac{32KB}{16B} = 2^{11} = 2048$
n_set = n_blocchi / n_vie = $211/2^2 = 2^9$
OFFSET = $\log_2 16 = 4$ b
INDEX = $\log_2 2^9 = 9$ b
TAG = IND_FISICO - INDEX - OFFSET = 27 - 9 - 4 = 14 b.

Esercizio 3

Tradurre in assembly MIPS la seguente funzione `low2upp` che prende in input una stringa `s` di lunghezza `l`, e sostituisce tutte le lettere minuscole in maiuscole.
Nota che 97 è il codice ASCII di `'a'` e 122 il codice di `'z'`, mentre 32 è la costante da sottrarre per trasformare minuscole in maiuscole.

```
void low2upp(char *s, int l) {
   int i;
   char v;

   for (i=0; i < l; i++) {
      v = s[i];
      if (v >= 97 && v <= 122)
         s[i] = v - 32;
   }
}
```

1. Tradurre in C con `if-goto` e `goto`.
2. Tradurre in assembly MIPS, usando `$s0` per la variabile `char v` (in verità, il byte meno significativo del registro).
   E' possibile usare pseudo-istruzioni come `blt`, `ble`, `bgt` e `bge`.

Soluzione

Il codice con `if-goto`:

```
void low2upp(char *s, int l) {
   int i;
   char v;

   i=0;
init_for:
   if (i >= l) goto exit_for;

   v = s[i];
   if (v < 97) goto exit_if;
   if (v > 122) goto exit_if;
   s[i] = v - 32;
exit_if:
   i++;
   goto init_for;

exit_for:
   return;
}
```

Traduzione MIPS:

```
low2upp:
   # s in $a0 e l in $a1
   addi $sp, $sp, -4
   sw $s0, 0($sp)

   # i in $t0, v in $s0

   li $t0, 0       # i=0;
init_for:
   bge $t0, $a1, exit_for #  if (i >= l) goto exit_for;

   add $t1, $a0, $t0
   lb $s0, 0($t1)         #   v = s[i];
   blt $s0, 97, exit_if   #   if (v < 97) goto exit_if;
   bgt $s0, 122, exit_if  #   if (v > 122) goto exit_if;
   addi $t2, $s0, -32
   sb  $t2, 0($t1)        #   s[i] = v - 32;
exit_if:
   add $t0, $t0, 1        #   i++;
   j init_for             #   goto init_for;

exit_for:
   lw $s0, 0($sp)
   addi $sp, $sp, 4
   jr $ra                 #   return;
```

Domande

1. Dopo un page fault, si può verificare un TLB miss (motivare)?
2. L'I/O si può programmare sia con il polling e sia tramite interrupt. Quando conviene l'uno o l'altro metodo? Distinguere tra dispositivi lenti o veloci, e per i veloci i casi in cui i dispositivi trasferiscono in continuazione o raramente.
3. Nel processore MIPS esiste l'indirizzamento PC-relative, impiegato nelle istruzioni di branch come `beq`. In cosa consiste? Quali sono le motivazioni che hanno portato i progettisti ad adottarlo nel progetto del set di istruzioni MIPS?
4. Discutere a grandi linee cosa sono i file oggetto e le librerie, e come questi possono, staticamente o dinamicamente, essele collegati (linking) alle librerie.