Übungsblatt 6

6.1 Pipelining: CPI-Wert

Bei der Ausführung einer Anwendung auf einer MIPS-Pipeline-Architektur mit „Branch Prediction“ wurde der in der Tabelle angeführte Befehlsmix bestimmt. Berechnen Sie den durchschnittlichen CPI-Wert für diese Anwendung unter folgenden Annahmen:

- 75% aller Load-Befehle werden unmittelbar von Befehlen gefolgt, die den geladenen Wert verwenden; in diesem Fall wird nach dem Load-Befehl ein Leerzyklus in die Pipeline eingefügt.
- 10% aller Branch-Befehle werden falsch vorhergesagt; dies verursacht einen Leerzyklus in der Pipeline.
- Jump-Befehle verursachen 2 Leerzyklen in der Pipeline.
- Sonst treten keine weiteren Leerzyklen in der Pipeline auf.

Befehlsgruppe	Häufigkeit
Load	25%
Store	10%
R-Format	40%
Branch	20%
Jump	5%

6.2 Bedingte Sprünge

Durch eine Erweiterung der MIPS-Beispiel-Pipeline (vgl. VO-Folie 3-102 ff.), kann die Latenz von bedingten Sprüngen verringert werden. Insbesondere können durch zusätzliche Hardware die Auswertung der Sprungbedingung und die Berechnung der Sprungzieladresse bereits in der zweiten Pipelinestufe erfolgen.

(a) Welche positive Auswirkung hat die genannte Hardware-Erweiterung auf die Latenz von Branch-Befehlen? Welchen negativen Effekt gibt es für die Latenz von Datenabhängigkeiten zu vorausgehenden ALU- bzw. Load-Befehlen? Wie groß sind diese Latenzen (in Takten)?

(b) Wie viele Takte benötigt die Ausführung des folgenden MIPS-Code-Fragments auf der gegebenen Pipeline? (Ergebnis auf 0,1% genau)

```
          xor   $s0, $zero, $zero
          addi  $t0, $a0, 600
outer:    addi  $t1, $a1, 200
inner:    addi  $s0, $s0, 1
          addi  $t1, $t1, -1
          bne   $t1, $a1, inner
          addi  $t0, $t0, -2
          bne   $t0, $a0, outer
```

6.3 Branch Prediction

Das MIPS-Code-Fragment aus Ü 6.2 wird auf der dort beschriebenen Beispiel-Pipeline mit Unterstützung für Branch Prediction ausgeführt. Die Pipeline besitzt eine Branch History Tabelle (BHT) und einen Branch Target Buffer (BTB)[1], die beide in der ersten Pipelinestufe ausgelesen werden. Das PC-Register enthält somit am Beginn der zweiten Pipelinestufe die vorhergesagte Sprungadresse (oder PC+4, falls BHT oder BTB keine gültigen Einträge für die aktuelle Befehlsadresse enthalten). Beantworten Sie folgende Fragen:

a) Wie viele Taktzyklen werden benötigt, wenn eine *1-Bit BHT* implementiert ist? (Ergebnis auf 0,1% genau)

b) Wie viele Taktzyklen werden benötigt, wenn eine *2-Bit BHT* laut VO-Folie 3-109 implementiert ist? (Ergebnis auf 0,1% genau)

c) Verwenden Sie den MARS BHT Simulator (siehe Howto im Moodle), um die Anzahl der richtig bzw. falsch vorhergesagten Sprünge in a) und b) zu verifizieren.

6.4 Pipelining: Bedingter move Befehl

(a) Geben Sie eine möglichst kurze Sequenz von MIPS-Assembler-Befehlen an, welche das Maximum der Werte der Register `$t0` und `$t1` im Register `$t2` ablegt. Verwenden Sie dabei keine Pseudo-Befehle. Die Befehlssequenz soll auf der Beispiel-Pipeline mit Branch Prediction laut Ü 6.3 ausgeführt werden. Nehmen Sie an, dass das Maximum mit der gleichen Wahrscheinlichkeit in $t0 bzw. in $t1 zu finden ist und dass die Branch Prediction in der Hälfte aller Fälle richtig ist. Wie viele Takte benötigt die Ausführung dieser Befehlssequenz im Mittel?

(b) Angenommen, der MIPS-Befehlssatz sei um einen Befehl `movz $rd, $rs, $rt` (kein Pseudo-Befehl) erweitert worden, der den Wert des Registers `$rs` nach `$rd` kopiert, falls `$rt` den Wert 0 hat; anderenfalls hat dieser Befehl keinen Effekt. Benutzen Sie diesen Befehl, um eine effizientere Lösung für a) ohne Verwendung von Verzweigungs- oder Sprungbefehlen anzugeben. Um welchen Prozentsatz sinkt die Ausführungszeit der Befehlssequenz auf der Beispiel-Pipeline gegenüber a)? Wodurch wird die Leistungssteigerung verursacht?

[1] Der Branch Target Buffer oder *Sprungzielpuffer* ist eine kleine, schnelle Speichereinheit (Cache), die zur Adresse eines Sprungbefehls die zuletzt verwendete Sprungzieladresse speichert.

6.5 Loop Unrolling

Gegeben sei folgendes MIPS-Assemblercodefragment, das auf einem Pipelined-Prozessor mit „Delayed Branching“ (1 Takt Branch Delay) ausgeführt wird. Die Latenzen zwischen abhängigen Befehlen sind in untenstehender Tabelle angegeben (FP = floating point).

```
      # initialize c as $f0, d as $f2 – not shown
loop: l.d   $f4, 0($t0)       # load x[i]
      sub.d $f6, $f4, $f0     # x[i] - c
      l.d   $f8, 0($t1)       # load y[i]
      mul.d $f10, $f6, $f8    # (x[i] – c) * y[i]
      add.d $f12, $f10, $f2   # (x[i] - c) * y[i] + d
      s.d   $f12, 0($t2)      # store result element z[i]
      addi  $t2, $t2, -8
      addi  $t1, $t1, -8
      addi  $t0, $t0, -8
      bne   $t0, $t4, loop
      nop
```

(a) Identifizieren sie alle Daten- und Kontrollabhängigkeiten, die Leerzyklen (Stalls) verursachen. Wie viele Takte werden für ein Ergebniselement (durch s.d gespeicherter Wert) benötigt?

(b) Optimieren Sie den Code durch Umordnen von Befehlen so, dass er auf dem gegebenen Prozessor möglichst schnell ausgeführt wird. Wie viele Takte werden für die Verarbeitung eines Ergebniselements (z[i]) durchschnittlich benötigt?

(c) Rollen Sie die Schleife zweimal ab (zwei Kopien des Code-Fragments in einer Schleifeniteration), und ordnen Sie den Code so um, dass er auf dem gegebenen Prozessor möglichst schnell ausgeführt wird. Wie viele Takte werden nun pro Ergebniselement benötigt?

(d) Wodurch wird die Leistungssteigerung beim Abrollen von Schleifen im Allgemeinen erreicht?

Erzeugender Befehl (schreibt Register $x)	**Benutzender Befehl (liest Register $x)**	**Latenz / Zwischentakte (um Leerzyklen zu vermeiden)**
FP ALU operation	FP ALU operation	3
FP ALU operation	Store FP double	2
Load FP double	FP ALU operation	1
Load FP double	Store FP double	0
Load integer	Integer operation	1
Load integer	Branch	2
Integer operation	Integer operation	0
Integer operation	Branch	1