timon.kelter [☺] tu-dortmund.de

Übung zur Vorlesung
Rechnerarchitektur
Sommersemester 2014

Übungsblatt 5

(10 Punkte)

Besprechung am Montag, 19. Mai 2014

5.1 Scoreboarding (3 Punkte)

Im Online-Material zur Übung finden Sie ein Beispiel zu den gespeicherten Informationen innerhalb der Scoreboarding-Datenstrukturen, für einen aus der Vorlesung bekannten Beispielcode. Konkret ist in den Tabellen folgende Situation festgehalten:

- erste Lade-Instruktion ist vollständig verarbeitet und hat ihr Ergebnis in der *write result*-Phase geschrieben.
- zweite Lade-Instruktion hat die Ausführungsphase abgeschlossen und wartet auf das Schreiben ihres Ergebnisses.
- restliche Instruktionen befinden sich vor bzw. in der *issue*-Phase.

a) Führen Sie die Programmausführung für die nächsten 10 Takte fort und stellen Sie für jeden Takt den aktuellen Stand der Scoreboarding-Datenstrukturen am Taktende dar. Benutzen Sie hierfür die Vorlage *scoreboarding.ppt* von der Webseite der Übung. Verwenden Sie für diese Aufgabe folgende Latenzzeiten der *execute*-Phase:

   - Addition: 2 Taktzyklen
   - Multiplikation: 5 Taktzyklen
   - Division: 8 Taktzyklen

   Die Latenzzeiten sind so zu verstehen, daß die Execute-Phase exakt die angegebene Anzahl Takte benötigt um zu terminieren, d.h. wenn beispielsweise eine Addition in Takt x gestartet wird endet sie am Ende von Takt $x+1$. *Forwarding* von Ergebnissen wird nicht unterstützt, d.h. Ergebnisse die in der Write-Results-Phase geschrieben werden, können erst im nächsten Takt in der Read-Operands-Phase gelesen werden. Außerdem werden keine Operanden gepuffert, d.h. wenn ein Operand verfügbar ist, ein anderer aber noch nicht, werden beide erst gelesen wenn auch beide verfügbar sind.

b) Erklären Sie kurz den Unterschied zwischen den im Skript erwähnten Verfahren des *Scoreboardings* und dem Verfahren von *Tomasulo*. Welche Unzulänglichkeiten, die beim Scoreboarding auftreten, werden von dem Verfahren von Tomasulo beseitigt? Warum?

5.2 Tomasulo-Algorithmus (2 Punkte)

Im Online-Material zur Übung finden Sie eine Vorlage, die die gespeicherten Informationen innerhalb der *Tomasulo reservation stations* für ein gegebenes MIPS-Codefragment zeigt. Konkret ist in den Tabellen folgende Situation festgehalten:

- erste Lade-Instruktion ist vollständig verarbeitet und hat ihr Ergebnis in der *write result*-Phase geschrieben.
- zweite Lade-Instruktion hat die Ausführungs-Phase abgeschlossen und wartet auf das Schreiben ihres Ergebnisses.
- restliche Instruktionen befinden sich in der *issue*-Phase.

a) Führen Sie die Programmausführung für die 8 folgenden Takte fort und zeigen Sie, welche Informationen am Ende jedes Taktes in den Tomasulo-Datenstrukturen gespeichert sind. Eine Vorlage hierfür wird auf der Übungshomepage bereit gestellt. Es ist ausreichend, wenn Sie in jeder Tabelle nur die Änderungen zum vorhergehenden Takt eintragen.

Verwenden Sie für diese Aufgabe folgende Latenzzeiten (der *EX*-Phase):

Instruktionstyp	Latenzzeit
Laden	1 Taktzyklus
Addition	1 Taktzyklus
Multiplikation	3 Taktzyklen
Division	6 Taktzyklen

Die Latenzzeiten sind so zu verstehen, daß die Execute-Phase auf der entsprechenden Verarbeitugnseinheit exakt die angegebene Anzahl Takte benötigt um zu terminieren, d.h. wenn beispielsweise eine Multiplikation in Takt x gestartet wird endet sie am Ende von Takt $x+2$. Wenn ein Ergebnis in der *Write*-Phase geschrieben wird, kommt es noch im selben Takt bei allen darauf wartenden Reservation Stations an, und diese fangen im darauf folgenden Takt mit der Abarbeitung der Instruktion an. Pro Takt kann über den *Common Data Bus* nur ein einzelnes Ergebnis übertragen werden. Wenn mehrere Ergebnisse im selben Takt übertragen werden sollen setzt sich immer das Ergebnis durch, dessen Instruktion zuerst die Issue-Phase durchlaufen hat (*in-order* Issue).

b) Welche Änderungen am Tomasulo-Algorithmus sind nötig, um in jedem Takt die Ausführung mehrerer Instruktionen beginnen (*multiple issue*) und Befehle spekulativ ausführen zu können?

5.3 Sprungvorhersage (2 Punkte)

Die Vergrößerung eines Sprungvorhersage-Puffers reduziert die Wahrscheinlichkeit, dass zwei Sprung-Befehle (mit gleichen niederwertigen Adressbits) dasselbe Prädiktionsbit nutzen. Im Allgemeinen ist ein Prädiktor, der ausschließlich Sprungvorhersagen bezüglich einer einzigen Instruktion macht, genauer als derselbe Prädiktor, der von mehreren Sprungbefehlen geteilt wird.

a) Nehmen Sie an, dass sich zwei Sprungbefehle in einer Schleife befinden und abwechselnd ausgeführt werden. Erstellen Sie ein Sequenz von *branch taken*- und *branch not taken*-Aktionen (für die beiden Sprünge), die verdeutlicht, dass das gemeinsame Nutzen eines 1-bit-Prädiktors die Vorhersage-Fehlerrate erhöht.

b) Listen Sie diesmal eine Abfolge von *branch taken*- und *branch not taken*-Aktionen auf, die das Gegenteil zum Aufgabenteil a) darstellt, nämlich dass die gemeinsame Nutzung des Prädiktors die Fehlerrate verringert.

5.4 Mehrstufige Prädiktoren (3 Punkte)

Betrachten Sie das Verhalten eines mehrstufigen (1,2) Sprungprädiktors anhand des folgenden Codefragmentes, das an Adresse `0xA0000000` im Speicher liegt und zwei voneinander abhängige Sprünge enthält. Nehmen Sie an, daß jede der unten aufgeführten Instruktionen als 32-Bit MIPS-Instruktion codiert ist und dass der Adressraum 2^{32} Bytes umfasst.

```
        li   $s0, 0
loop:   div  $s0, 2
        mfhi $s3
        bne  $s3, $zero, else_1
then_1: addi $s1, $s1, 4
else_1: mul  $s1, $s1, 2
        bne  $s3, $zero, else_2
then_2: addi $s2, $s2, 1
```

```
else_2: addi $s0, $s0, 1
        bne  $s0, 5, loop
```

a) Listen Sie alle Zustandsänderungen und Vorhersagen des Sprungprädiktors während der Ausführung des Codefragmentes in chronologisch korrekter Reihenfolge auf. Gehen Sie dazu wie folgt vor:

- Annotieren Sie an jeder Instruktion die Speicheradresse an der sie steht.
- Stellen Sie dann den Ausgangszustand des Prädiktors mit der Vorlage von der Übungswebseite dar. Nehmen Sie hierfür an, dass die einzelnen 2-Bit Prädiktoren mit dem Wert $(1)_{10}$ (nicht $(11)_2$) initialisiert sind und jeweils 4 Einträge haben und dass `$s1` und `$s2` mit 0 initialisiert sind. Der in der Branch History abgelegte Wert beim Betreten des Codefragments sei 0.
- Gehen Sie das Fragment Schritt für Schritt durch und geben Sie bei jedem ausgeführtem Sprung die getroffene Vorhersage und die Änderungen am Prädiktorzustand an.

b) Welche Vorteile bietet die Kopplung des Prädiktors aus a) mit einem Branch-Target-Buffer bzw. einem Branch-Folding?

c) Um wieviel Prozent steigt der Speicherbedarf des angegebenen Prädiktors wenn ein Branch-Target-Buffer (*BTB*) bzw. ein Branch-Folding (*BF*) (für eine einzelne Instruktion) integriert wird? Nehmen Sie hierzu an, dass der Prädiktor auch schon in der Konfiguration ohne BTB und BF für jede Zelle einen Tag speichert um gemeinsame Nutzung einer Prädiktor-Zelle durch mehrere Sprungbefehle zu verhindern.