Übungen zur Vorlesung "Architektur und Programmierung von Grafik- und Koprozessoren"
Übungsblatt 2

Sommersemester 2018

2 Parallele Architekturen

Aufgabe 2.1

a.)
Auf einer hypothetischen CPU stehen Ihnen zwei arithmetisch- logische Einheiten (ALU0 und ALU1) sowie zwei Speicherverwaltungseinheiten (MMU0 und MMU1) zur Verfügung. Der Instruktionssatz sieht die Instruktionen LD, ST, ADD und MUL vor. Diese benötigen zur Ausführung jeweils 4, 4, 1 und 2 Taktzyklen.

Die LD und ST Instruktionen benötigen im ersten Taktzyklus eine ALU und in den übrigen Taktzyklen bis zum Ende der Ausführungszeit genau eine MMU. Die ST Instruktion kann außerdem nur auf MMU1 geplant werden. Die ADD und MUL Instruktionen benötigen lediglich ALU Ressourcen.

Geben Sie alle möglichen alternativen Reservierungstabellen für die vier Instruktionen an. Gehen Sie grundsätzlich davon aus, dass in zwei aufeinanderfolgenden Taktzyklen immer nur die gleiche Ressource belegt werden darf, wenn in diesen Taktzyklen der gleiche Ressourcentyp benötigt wird. (5 Punkte)

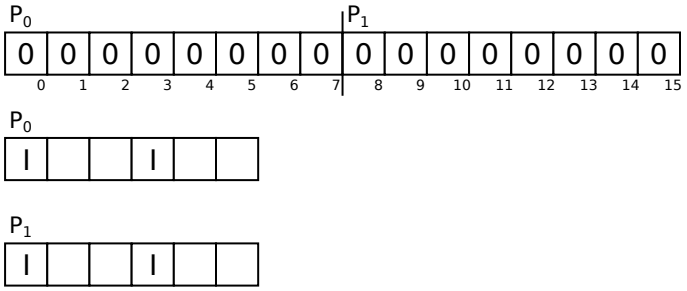
b.)
Auf einer hypothetischen CPU stehen Ihnen eine MMU und eine ALU zur Verfügung. Speicherzugriffsinstruktionen belegen die MMU für jeweils zwei Taktzyklen. Arithmetische Instruktionen belegen die ALU für jeweils einen Taktzyklus. Nehmen Sie vereinfachend an, dass zu Anfang jeder Schleifenausführung alle benötigten Daten mit *einer* LD Instruktion aus dem Speicher in Register geladen werden.
Geben Sie die Reservierungstabelle für die erste Iteration der nachfolgenden Schleife an.

```
for (i = 0; i < N; ++i)
    y[i] = x[i] * a[i] / b[i];
```

Welches sind die ungültigen Latenzen und nach wievielen Taktzyklen kann die zweite Iteration frühestens geplant werden? Erstellen Sie darauf aufbauend auch die Reservierungstabelle für die ersten beiden Iterationen. Welches sind die ungültigen Latenzen und nach wievielen Taktzyklen kann eine weitere Iteration geplant werden? (5 Punkte)

Aufgabe 2.2

Wir betrachten eine NUMA Maschine mit 8-bit Wortbreite, die das MESI Protokoll für Cache Kohärenz implementiert. Die Prozessoren P_0 und P_1 verfügen jeweils über acht Byte Hauptspeicher mit geteiltem Adressraum. Auf die Speicherworte kann über die Adressen [0..15] zugegriffen werden. Jeder Prozessor verfügt über vier Byte zwei Wege assoziativen L1 Cache. Cache Lines haben eine Länge von zwei Speicherworten. Speicherworte an den Adressen [0..3] sowie [8..11] sind mit Cache Line 0 assoziiert, Speicherworte an den Adressen [4..7] sowie [12..15] sind mit Cache Line 1 assoziiert. Es wird immer von 2-Byte alignierten Adressen vom Speicher in den Cache gelesen. Soll beispielsweise das Speicherwort an Adresse 11 in den Cache transferiert werden, lädt der Cache Controller zwei Byte ab Adresse 10 in Cache Line 0. Transfers zwischen Registern und Caches erfolgen mit Granularität von einem Speicherwort. Die Caches implementieren eine einfache Replacement Strategie: wird ein Item aus dem Speicher gelesen, wird dieses immer auch in den lokalen Cache geschrieben.
Beim Programmstart ist der Speicher 0-initialisiert und alle Cache Lines invalidert.



Dokumentieren Sie alle Zustandsveränderungen sowie die jeweiligen Inhalte der L1 Caches graphisch wie in der Abbildung für die folgenden Ereignisse. Geben Sie außerdem an, welche Nachrichten durch die Ereignisse ausgelöst werden und welche Ereignisse zu Schreiboperationen in den Hauptspeicher führen.

1. P_0 schreibt den Wert 1 an Adresse 0.
2. P_0 schreibt den Wert 3 an Adresse 1.
3. P_0 schreibt den Wert 5 an Adresse 2.
4. P_0 schreibt den Wert 7 an Adresse 3.
5. P_1 liest von Adresse 4.
6. P_1 schreibt den Wert 3 an Adresse 5.
7. P_0 liest von Adresse 13.
8. P_0 liest von Adresse 12.
9. P_0 schreibt den Wert 2 an Adresse 12.
10. P_1 liest von Adresse 12.

(5 Punkte)

Aufgabe 2.3

Die Template Klasse `sync_queue` in der beigefügten Datei `queue.cpp` wird von mehreren Threads gleichzeitig als Schlange zum Austauschen von Nachrichten genutzt. Leider hat der Programmierer die Klasse nicht vollständig implementiert, sodass es zu *data races* kommt, wenn die Threads sie benutzen. Erweitern Sie die Klasse, sodass sie threadsicher ist:

- `pop_front()` wird nur ausgeführt, falls sich Nachrichten in der Schlange befinden. Andernfalls blockiert die Funktion. Die Synchronisation können Sie mit einer Semaphore implementieren. Verwenden Sie die Klasse aus der beigefügten Datei `semaphore.h`. Solange die Schlange leer ist, wird gewartet (`semaphore.wait()`). Befinden sich Nachrichten in der Schlange, wird den wartenden Threads signalisiert, dass nicht mehr gewartet werden muss (`semaphore.notify()`).
- Zugriffe auf das der Datenstruktur zugrunde liegende `std::deque` Objekt erfolgen innerhalb einer Critical Section. Dies können Sie beispielsweise mit Hilfe der C++ STL Klassen `std::mutex` und `std::unique_lock` implementieren.

Wenn Sie die Klasse `sync_queue` erfolgreich erweitert haben, sollten die Threads Nachrichten ohne data races austauschen können. (5 Punkte)

Abgabe bitte bis zum 02.05.2018, 22:00h in Ilias.