Übungen zur Vorlesung "Architektur und Programmierung von Grafik- und Koprozessoren"
Übungsblatt 1

Sommersemester 2019

1 Parallele Architekturen

Aufgabe 1.1

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.

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?

Aufgabe 1.2

Wir betrachten den nachfolgenden Ausschnitt des Befehlssatzes einer hypothetischen CPU:

Instruktion	**Beschreibung**	**Latenz**
`ADD $R1` C	Addiere den konstanten Wert C auf den Inhalt von Register `$R1`	4 ns
`ADD $R1 $R2`	Addiere Registerinhalte `$R1` und `$R2`, speichere das Ergebnis in `$R1`	4 ns
`MUL $R1 $R2`	Multipliziere Registerinhalte `$R1` und `$R2`, speichere das Ergebnis in `$R1`	4 ns
`LD $R1` $[S]$	Lade den Inhalt an Speicherstelle $[S]$ in Register `$R1`	8 ns
`ST` $[S]$ `$R1`	Speichere den Inhalt von Register `$R1` an Speicherstelle $[S]$	8 ns
`JNZ $R1 label1`	Falls Inhalt von Register `$R1` $\neq 0$, springe zu Label `label1`	1 ns

sowie das nachfolgende Programm:

```
LD $R1 [0xFF-0]
LD $R2 [0xFF-1]
LOOP:
ADD $R1 -1
LD $R3 [0xC0-$R1]
ADD $R2 $R3
LD $R4 [0x80-$R1]
MUL $R2 $R4
MUL $R2 $R2
JNZ $R1 LOOP
ST 0xFF $R2
```

Nehmen Sie an, dass der Inhalt der Register `$R1`, `$R2`, `$R3` und `$R4` zunächst 0 beträgt und dass initial im Speicher an der Adresse `0xFF` der konstante Wert 2 steht.

a.)
Zeichnen Sie den Datenabhängigkeitsgraph für die Schleife im Programm auf.

b.)
Unrollen Sie die Schleife. Planen Sie die so resultierenden Schleifenausführungen auf einer hypothetischen CPU mit einer ALU, auf der alle arithmetischen Operationen ausgeführt werden und einer MMU, auf der die Speichertransferoperationen ausgeführt werden.

Aufgabe 1.3

a.)
Übersetzen Sie die den Übungsunterlagen beigefügte Datei `queue.cpp` mit einem C++11 kompatiblen Compiler. Eine entsprechende Kommandozeile für `gcc` kann beispielsweise wie folgt aussehen.

```
g++ queue.cpp -std=c++11 -o queue
```

Das Programm `queue` wurde mit `gcc` getestet, sollte aber auch mit anderen Compilern übersetzen. Das so erzeugte *Binary* `queue` können Sie z. B. auf der Kommandozeile ausführen. Es wird allerdings mit einer Fehlermeldung abstürzen, da im Programm noch Programmierfehler vorhanden sind, die Sie im Folgenden beheben sollen.

b.)
Die Template Klasse `sync_queue` in der beigefügten Datei `queue.cpp` wird von mehreren Threads gleichzeitig als Schlange zum Austauschen von Nachrichten genutzt. Intern verwaltet die Schlange ihre Daten mittels der Standard Template Library (STL) Klasse `std::deque<T>`, die nicht threadsicher ist. Die Schlange exponiert nur zwei Funktionen: `push_back()`, mit der Elemente hinten in die Schlange eingefügt werden können und `pop_front()`, die das vordere Element zurückgibt und aus der Schlange entfernt. Die `main()` Funktion erzeugt eine Reihe von Threads, mit denen das Verhalten der Klasse getestet wird.

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()` soll nur ausgeführt werden, falls sich bereits Nachrichten in der Schlange befinden. Andernfalls soll die Funktion blockieren. Die Synchronisation können Sie mit einer Semaphore implementieren. Verwenden Sie die Klasse aus der beigefügten Datei `semaphore.h`. Solange die Schlange leer ist, soll gewartet werden (`semaphore.wait()`). Befinden sich Nachrichten in der Schlange, soll den wartenden Threads signalisiert werden, dass nicht mehr gewartet werden muss (`semaphore.notify()`).
- Zugriffe auf das der Datenstruktur zugrunde liegende `std::deque` Objekt sollen innerhalb einer Critical Section erfolgen. 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.

Das Übungsblatt wird am 18.04.2019 besprochen.