Tutoriumsblatt 13
Rechnerarchitektur im SoSe 2020

Zu den Modulen P,Q

Tutorium: Die Aufgaben werden in Tutorien-Videos vorgestellt, die am 16. Juli 2020 (17 Uhr) veröffentlicht werden.

Aufgabe T35: Arbeitsweise Caches

(– Pkt.)

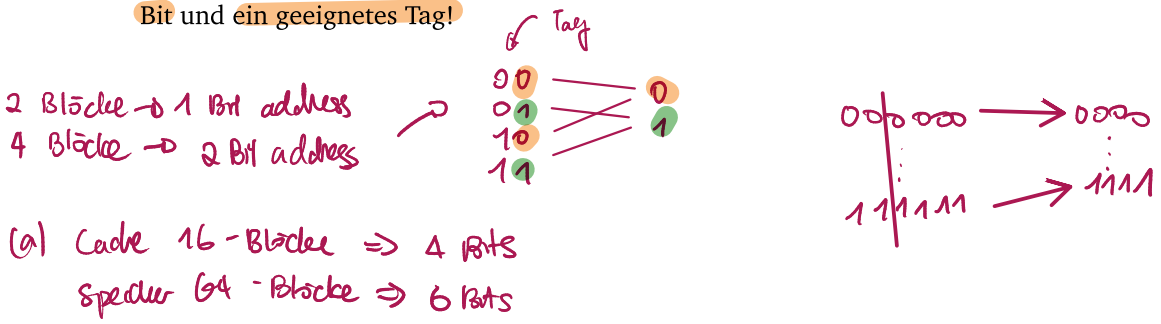
Nehmen Sie einen Speicher mit 64 und einen Cache mit 16 Blöcken an. Wir benötigen nacheinander folgende Adresszugriffe bei anfangs leerem Cache:

`1, 4, 8, 5, 20, 17, 19, 56, 9, 11, 4, 43, 5, 6, 9, 17`

a. Geben Sie für jede Referenz an, ob ein Cache-Hit oder ein Cache-Miss eintritt. Gehen Sie dabei von dem in der Vorlesung eingeführten Direktabbildungs-Verfahren aus.

b. Stellen Sie den Inhalt des Caches dar, nachdem alle Zugriffe erfolgt sind.

c. Wie viel Speicherplatz ist erforderlich, um einen Direct-mapped Cache zu realisieren, der 256 KByte Daten zwischenspeichern kann, wenn die Größe jedes Cache-Blocks und jedes Datenwortes im Speicher 32 Bit = 4 Byte beträgt. Gehen Sie von 32-Bit Adressen aus (es werden ganze Datenworte adressiert). Hinweis: Jeder Cache-Block benötigt ein Validierungs-Bit und ein geeignetes Tag!



(a) Cache 16-Blöcke ⇒ 4 Bits
Speicher 64-Blöcke ⇒ 6 Bits

Add(DEC)	Add(BIN)	Hit/Miss	Cacheblock
1	000001	M	0001
4	000100	M	0100
8	001000	M	1000
5	000101	M	0101
20	010100	M	0100
17	010001	M	0001
19	010011	M	0011
56	111000	M	1000
9	001001	M	1001
11	001011	M	1011
4	000100	M	0100
43	101011	M	1011
5	000101	H	0101
6	000110	M	0110
9	001001	H	1001
17	010001	H	0001

Cache:

Addresse	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
Inhalt	~~1~~ 17			19	~~4~~ 20 4	5	6		~~8~~ 56	9		~~11~~ 43				

(b) Cache:

Addresse	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
Inhalt	17			19	4	5	6		56	9		43				

(c) 256 kByte in 32 Bit Speicher $\Rightarrow$ $\frac{256\ \text{kByte}}{32\ \text{Bit}} = \frac{2^{18}\ \text{Byte}}{2^2\ \text{Byte}} = 2^{16}$ Blöcke

(Total Speicher; 32 Bit = 4 Byte = 1 Word)

$\Rightarrow$ 16 Bit address for Cache

Since we use 32 Bit addresses $\Rightarrow$ 32 − 16 = 16 Bit Tag

[16 Bit] [16 Bit]
TAG Cache Block

↳ Pro Block: 32 Bit Speicher + 16 Bit Tag + 1 Bit Valid Bit = 49 Bit

$\Rightarrow$ 2^{16} Blocks × 49 bits each = 3 211 264 Bit = 401 408 Byte
= 392 KB

Aufgabe T36: Pipelining

(– Pkt.)

Beantworten Sie die folgenden Fragen zum Thema Pipelining.

a. Erklären Sie die prinzipielle Idee von Pipelining.

b. Wie bestimmt man die Länge einer Pipelinestufe, d.h. wie lang muss das Zeitintervall (Taktzyklus) für eine Stufe mindestens sein?

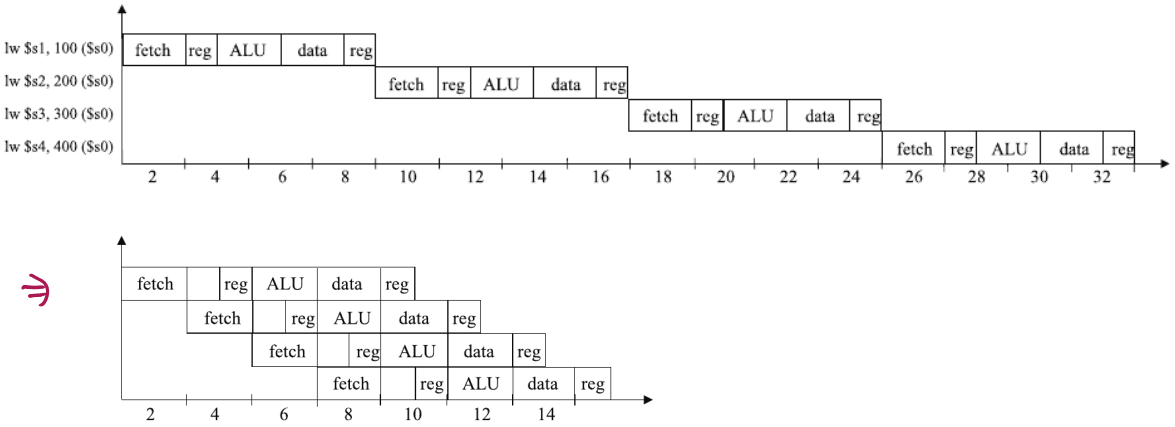
c. Warum müssen die Pipelinestufen eine gleich lange Ausführungszeit besitzen?

d. Von welchen zwei Eigenschaften hängt der Leistungsgewinn einer idealen Pipeline ab (also ohne Berücksichtigung von Konflikten)?

e. Nennen Sie zwei Gründe, warum es keinen Sinn macht bzw. nicht möglich ist, die Anzahl der Stufen beim Design einer Pipeline beliebig hoch zu wählen?

f. Benennen und erläutern Sie die drei verschiedenen Arten von Konflikten (Hazards), die durch die Einführung von Pipelining entstehen können? Geben Sie je ein Beispiel für diejenigen Hazards an, die bei der MIPS-Architektur auftreten können. Begründen Sie kurz, warum ein Hazard im Falle der MIPS-Architektur nicht entstehen kann,

(a)



⇒

Wir beobachten, dass häufig die gleichen Befehlskomponenten in der gleichen Reihenfolge auftreten. Deshlab unterteilen wir die Befehle in Teilinstruktionen (sog. Pipelinestufen), die dann parallel ausgeführt werden können. Danach erhöht sich der Durchsatz der CPU.

(b) Die Länge der Pipelinestufen muss der Länge der langsamsten Teilinstruktion entsprechen.

(c) Da der Stufen sich sonst überschneiden und nicht mehr zum Taktzyklus passen würden.

(d)
- Anzahl der Stufen: je mehr Stufen, desto effizienter.
- Ausführungszeit pro Stufe: je kürzer, desto schneller.

(e) Mehr Stufen führen zu mehr potenziellen Hazards (sog. Konflikten) und somit zu einer Minderung der Effizienz.
Außerdem ist es nicht möglich, Befehle in beliebig viele Stufen zu unterteilen. Auch aus Hardwaresicht steigen die Kosten mit steigender Stufenanzahl.

18) < HAZARDS >

1) Structural Hazards

Die Hardware ist nicht in der Lage, bestimmte Teilinstruktionen gleichzeitig auszuführen. Bei MIPS kommt das nicht vor, da die Architektur für das Pipelining konzipiert würde. Es gibt z.B. keinen gleichzeitigen Lese- und Schreibzugriff auf ein Register.

2) Control Hazards

Die Pipeline muss warten, ob ein bedingter Sprung ausgeführt werden muss.

Bsp:

```
beq     $t0, $t1, label
add     $t4, $t0, $t2
```

zu Beginn des add-Befehls ist der PC noch nicht bekannt.
=> Holen der nächsten Instruktion im folgenden Taktzyklus ist nicht möglich

Mögliche Lösungen:
- Einfügen von „Stalls“ (Warten) bis die Information bekannt sind.
- Branchprediction (Sprungvorhersage). Hierbei wird spekulativ weitergerechnet, bis feststeht, ob sich die Vorhersage als richtig erwiesen hat. Im Falle einer falschen vorhersage müssen die ausgeführten Befehle verworfen werden (Pipeline Flush), was sehr viel Zeit kostet.

3) Data Hazards

Ein Befehl kann nicht ausgeführt werden, der die benötigten Daten noch nicht bereitstehen.

Bsp:

```
add     $t0, $t1, $t2
add     $t4, $t0, $t2
```

Das Register $t0 wird gelesen, obwohl es eine veralteten Inhalt besitzt.

Mögliche Lösung: Einfügen von „Stalls“ (wie oben)