FernUniversität in Hagen

**FACHBEREICH**
**Informatik**
Lehrgebiet
Technische Informatik I
Rechnerarchitektur
Prof. Dr. Wolfram Schiffmann

FernUniversität in Hagen · 58084 Hagen

**An die Teilnehmer der Hauptklausur**

**TECHNISCHE INFORMATIK III 1709**

**im Sommersemster 2003**

| | |
|---|---|
| Ihr Zeichen | |
| Ihre Nachricht vom | |
| Mein Zeichen | PB |
| Auskunft erteilt | Herr Buschhorn |
| Telefon | 02331 987-4424 |
| Telefax | 02331 987-310 |
| E-Mail | Peter.Buschhorn @FernUni-Hagen.de |
| Hausanschrift | Universitätsstraße 1 58084 Hagen |
| Datum | 11.08.2003 |

## Hinweise zur Musterlösung der Hauptklausur

Sehr geehrte Teilnehmer der Hauptklausur des Kurses 1709 „Technische Informatik III“,
Sie können die Musterlösung der Hauptklausur vom 2.8.2003 auf der Seite des Lehrgebiets als PDF-Dokument herunterladen.

**http://www.informatik.ti1.fernuni-hagen.de/lehre/1709/klausur/ML1709HK.pdf**

Sollten Sie keine Möglichkeit haben, an dieses Dokument zu gelangen, melden Sie sich bitte bei dem Betreuer des Kurses

Herr Peter Buschhorn

Tel: 02331 987 4424
FAX: 02331 087 310

FernUniversität Hagen

Technische Informatik I

Informatikzentrum IZ

Universitätsstr. 1

58097 Hagen

Email: K1709@fernuni-hagen.de

um eine gedruckte Version zu erhalten.

## Aufgabe 1: Grundlagen I

a) Was unterscheidet die Architektur von der Mikroarchitektur eines Prozessors?

b) Welche Rechner fallen in die SIMD-Klasse der Flynnschen Klassifizierung?

c) Worin unterscheidet sich ein Feldrechner von einem Vektorrechner?

d) Ein Direct-Mapped-Cache ist prinzipiell immer langsamer als ein 4-Wege-Assoziativ-Cache. Richtig oder Falsch? Begründen Sie.

e) Je mehr Stufen eine Pipeline für einen gegebenen Befehlssatz besitzt, desto schneller kann das System getaktet werden. Richtig oder Falsch? Begründen Sie.

## *Musterlösung:*

a) Eine Prozessorarchitektur betrifft das äußere Erscheinungsbild, die Mikroarchitektur die interne Organisation.

b) Vektorrechner und Feldrechner, da bei beiden Rechnertypen mit einem Befehlswort eine Anzahl von Datenoperationen angestoßen und ausgeführt werden.

c) Beim Vektorrechner werden die Operationen in einer Vektor-Pipeline und beim Feldrechner taktsynchron auf meist sehr vielen Ausführungseinheiten ausgeführt.

d) Falsch, denn sowohl der Direct-Mapped-Cache wie auch der 4-Wege-Assoziativ-Cache müssen ein anliegendes Tag vergleichen. Der Aufwand ist beim Direct-Mapped-Cache geringer, dennoch wird bei beiden Organisationsformen der Tag durch ein 2-stufiges Schaltnetz verglichen.

e) Richtig. Wenn die Anzahl der Stufen einer Pipeline erhöht wird, werden die Aufgaben einer einzelnen Stufe vereinfacht. Dadurch wird bewirkt, dass jede Stufe für sich schneller fertig ist. Mithin kann der Takt entsprechend erhöht werden. Dies ist ein Grund für die hohe Stufenzahl in modernen Prozessoren.

## Aufgabe 2: Pipelining

Ausgangspunkt des Befehls-Pipelining ist die Aufspaltung eines Befehls in Teilschritte. Es setzt voraus, dass für die zugrundelegende Befehlsarchitektur eine Folge von Teilschritten gefunden wird, die für *alle* Befehle gleich ist. Alle Befehle (inklusive der zugehörigen Adressierungsarten) können beispielsweise in den folgenden fünf Teilschritten, welche Ihnen aus DLX-Architektur bekannt sind, bearbeitet werden (*Tabelle 1*).

**1. Instruction Fetch (IF):** Befehl holen (OpCode und Registeradressen können in einem Maschinenwort untergebracht werden).

**2. Instruction Decode (ID):** Befehl dekodieren und gleichzeitig die Quelloperanden aus dem Registerblock lesen.

**3. Execute (EX):** Führe eine arithmetische bzw. logische Operation mit den Operanden aus.

**4. Memory Access (MEM):** Daten holen (*LOAD*) oder Speichern (*STORE*)

**5. Write Back (WB):** Ergebnis in einem prozessorinternen Register speichern.

| | | | | | | |
|---|---|---|---|---|---|---|
| IF | ID | EX | MEM | WB | | |
| | IF | ID | EX | MEM | WB | |
| | | IF | ID | EX | MEM | WB |

**Tabelle 1:** Zeitlicher Ablauf der Teilschritte bei konfliktfreiem Befehls-Pipelining

Pipelinekonflikte können durch zusätzliche Hardware oder Software (Einfügen von `NOOP`, *no-operation-Befehle*) behoben werden. In den folgenden Teilaufgaben soll nur die Software-Lösung betrachtet werden. Wir untersuchen zunächst einen Pipelinekonflikt durch eine echte Datenabhängigkeit (*RAW Read After Write*), der in dem folgenden Programmfragment vorliegt:

| | |
|---|---|
| `ADD R3,R1,R2` | R3=R1+R2 |
| `SUB R4,R1,R3` | R4=R1-R3 |

a) Wie viele `NOOP`-Befehle sind zum Beheben dieses Konflikts nötig? Skizzieren Sie den Ablauf des Programms in der Pipeline und markieren Sie im Zeitdiagramm folgende Ereignisse:

**A)** Wert von R3 durch den `ADD`-Befehl errechnet,

**B)** Wert von R3 liegt *vor* dem Registerblock an,

**C)** Wert von R3 wurde im Registerblock gespeichert und

**D)** R3 liegt als Eingangsoperand für den `Sub`-Befehl an der ALU vor.

b) Nun betrachten wir einen Kontrollfluß-Konflikt, der im folgenden Programmfragment für einen DLX Prozessor vorliegt:

| | | |
|---|---|---|
| `1000` | `BEQZ 20` | Falls Zero-Bit 1 ist, springe um 20 Wörter relativ zum aktuellen Befehlszählerstand |
| ... | ... ... | ... |
| `1024` | `ADD R1,R2,R3` | R1=R2+R3 |

Wie viele NOOP-Befehle sind zum Beheben dieses Konflikts nötig? Skizzieren Sie den Ablauf des Programms in der Pipeline und markieren Sie im Zeitdiagramm (*Aufbau wie in Tabelle 1*) folgende Ereignisse:

i. Neuer Befehlszählerstand bestimmt und

ii. Wert von R1 wurde berechnet.

iii. Durch Umstellen von Befehlen kann ein optimierender Compiler die zur Konfliktbehebung erforderlichen NOOP-Befehle durch andere (unabhängige) Befehle ersetzen und so die Effizienz des Pipelining steigern. Gegeben seien weitere Befehle aus dem Umfeld des Programmfragments von Teilaufgabe a):

| | |
|---|---|
| JR 20 | |
| SUB R6,R2,R1 | R6=R2-R1 |
| AND R5,R2,R1 | R5=R2∧R1 |
| ... ... | ... |
| ADD R3,R1,R2 | R3=R1+R2 |
| SUB R4,R1,R3 | R4=R1+R3 |
| ... ... | ... |
| ADD R1,R4,R5 | R1=R4+R5 |
| XOR R6,R7,R8 | R6=R7⊕R8 |

Eliminieren Sie die NOOP-Befehle in Teilaufgabe a) durch geeignetes Umstellen von Befehlen.

## Musterlösung:

a) Die Datenabhängigkeit liegt in der Verwendung von *R3*. Erst nachdem der *ADD*-Befehl den neuen Wert von *R3* geschrieben hat, kann er im *SUB*-Befehl verwendet werden. Ein Operator lädt seine Operanden im zweiten Teilschritt (**ID**) und das Ergebnis des Operators wird im fünften Teilschritt (**WB**) zurückgeschrieben. Es müssen also so viele NOOPs eingefügt werden, dass das **ID** vom *SUB* nach dem **WB** vom *ADD* geschieht. Berücksichtigt man dies, ergibt sich das folgende Zeitdiagramm, in dem in der letzten Zeile auch die in der Aufgabenstellung angegebenen Ereignisse markiert sind:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| ADD R3,R1,R2 | IF | ID | EX | MEM | WB | | | |
| NOP | | NOP | NOP | X | X | X | | |
| NOP | | | NOP | NOP | X | X | X | |
| NOP | | | | NOP | NOP | X | X | X |
| SUB R4,R1,R3 | | | | | IF | ID | EX | MEM | WB |
| | | | | A) | B) | C) | D) | |

b) Die evtl. Modifikation des PC durch den *BRZ*-Befehl wird auch hier erst im letzten Teilschritt (**WB**) gültig. Mit dem Laden des nächsten Befehls muss man warten, bis der PC gültig ist und somit muss man also so viele NOOPs einfügen, bis die Pipeline leergelaufen ist. Der **IF** des nächsten Befehls kann natürlich erst geschehen, wenn feststeht, an welcher Stelle im Speicher der nächste Befehl steht.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| BEQZ 20 | IF | ID | EX | MEM | WB | | | | | |
| NOP | | NOP | NOP | X | X | X | | | | |
| NOP | | | NOP | NOP | X | X | X | | | |
| NOP | | | | NOP | NOP | X | X | X | | |
| NOP | | | | | NOP | NOP | X | X | X | |
| ADD R1,R2,R3 | | | | | | IF | ID | EX | MEM | WB |

A) B)

*c)* Betrachtet man das Programmfragment der Aufgabenstellung, dann muss ein nicht optimierender Compiler die folgende Codesequenz erzeugen, um alle RAW Datenabhängigkeiten zu beseitigen.

```
JR  20
SUB R6,R2,R1
AND R5,R2,R1
ADD R3,R1,R2  ; RAW Abhängigkeit mit dem nächsten Befehl
NOP
NOP
NOP
SUB R4,R1,R3  ; RAW Abhängigkeit mit dem nächsten Befehl
NOP
NOP
NOP
ADD R1,R4,R5
XOR R6,R7,R8
```

Es kommen also zwei Datenabhängigkeiten vor. Die erste ist schon in Teil a) analysiert worden, die zweite ist neu. Durch Umsortieren der Befehle kann man möglicherweise Datenabhängigkeiten beseitigen. Natürlich darf dabei der Sinn des Programms nicht verändert werden. Sinnvollerweise startet man mit einem Befehl, bei dem eine Datenabhängigkeit vorliegt und fügt dahinter Befehle ein, die unabhängig sind und keine neue Datenabhängigkeit erzeugen. Auf diese Art und Weise kann man eine Datenabhängigkeit beseitigen und erhält:

```
JR  20
ADD R3,R1,R2
SUB R6,R2,R1  ; Ersatz für NOP
AND R5,R2,R1  ; Ersatz für NOP
XOR R6,R7,R8  ; Ersatz für NOP
SUB R4,R1,R3  ; RAW Abhängigkeit mit dem nächsten Befehl
NOP
NOP
NOP
ADD R1,R4,R5
```

Durch diese Reihenfolge entsteht nun zwar eine WAW Datenabhängigkeit zwischen dem ersten *SUB*-Befehl und dem *XOR*-Befehl, was aber keine Konsequenzen hat.

## Aufgabe 3: Prediction

Das folgende Programm besteht aus zwei verschachtelten Schleifen. Das Register R4 sei mit $m > 0$ und das Register R3 mit $n > 0$ vorbelegt. Weiterhin sei das Register R1 mit 1 belegt.

```
Loop1:  SLE   R10, R3, R0    (1)  ; R3 <= R0 ? Status in R10
        BNEZ  R10, Ende1     (2)  ; wenn ja, dann gehe zu Ende1
Loop2:  SLE   R11, R4, R0    (3)  ; R4 <= R0 ? Status in R11
        BNEZ  R11, Ende2     (4)  ; wenn ja, dann gehe zu Ende2
...     ...   ...             ...  ...
        SUB   R4, R4, R1    (19)  ; R4 = R4 - R1
        J     Loop2         (20)  ; Goto Loop2
Ende2:  SUB   R3, R3, R1    (21)  ; R3 = R3 - R1
        J     Loop1         (22)  ; Goto Loop1
Ende1:  ...                 (23)  ;
```

Es sollen verschiedene Sprungvorhersagetechniken verglichen werden, aber dabei nur bedingte Sprünge betrachtet werden. Wie viele richtige und falsche Vorhersagen gibt es bei:

a) statischer Sprungvorhersage mit „Always taken"-Technik,

b) dynamischer Sprungvorhersage: Ein-Bit-Prädiktor (Initialzustand „predict taken"),

c) dynamischer Sprungvorhersage: Zwei-Bit-Prädiktor mit Sättigungszähler (Initialzustand: „predict weakly taken")?

Begründen Sie Ihre Ergebnisse durch Aufstellen einer Tabelle.

## *Musterlösung:*

Die folgende Tabelle zeigt die bedingten Sprünge und die Zustände der Sättigungszähler nach dem jeweiligen Sprung. Die Abkürzungen und Zeichen sind unter der Tabelle erläutert.

| Sprung | | Ein-Bit-Prädiktor | | Zwei-Bit_Prädiktor | | |
|---|---|---|---|---|---|---|
| äußere Schleife | innere Schleife | äußere Schleife | innere Schleife | äußere Schleife | innere Schleife | |
| Initialzustand | | T | T | | WT | |
| ng | | x NT | | x WNT | | |
| | ng | | x NT | | x WNT | |
| | ng<br>.<br>.<br>.<br>ng | | √ NT<br>.<br>.<br>.<br>√ NT | | √ SNT<br>.<br>.<br>.<br>√ SNT | $m$ |
| | g | | x T | | x WNT | |
| ng | | √ NT | | √ SNT | | |
| | g | | √ T | | x WT | $1$ |
| ng | | √ NT | | √ SNT | | |
| | g | | √ T | | √ ST | $1$ |
| ng | | √ NT | | √ SNT | | |
| .<br>.<br>. | .<br>.<br>. | .<br>.<br>. | .<br>.<br>. | .<br>.<br>. | .<br>.<br>. | |
| ng | | √ NT | | √ SNT | | |
| | g | | √ T | | √ ST | $1$ |
| ng | | √ NT | | √ SNT | | |
| | g | | √ T | | √ ST | $1$ |
| g | | x T | | x WNT | | |

$n$

Erläuterung der Abkürzungen:

| | |
|---|---|
| √ | Sprung richtig vorhergesagt |
| × | Sprung falsch vorhergesagt |
| g | Sprung wurde genommen |
| ng | Sprung wurde nicht genommen |
| T | predict taken |
| NT | predict not taken |
| ST | predict strongly taken |
| WT | predict weakly taken |
| WNT | predict weakly not taken |
| SNT | predict strongly not taken |

Für die äußere Schleife ergeben sich ein genommener und $n$ nicht genommene Sprünge und für die innere Schleife $n$ genommene und $m$ nicht genommene Sprünge.

| | äußere Schleife | | innere Schleife | |
|---|---|---|---|---|
| | richtig | falsch | richtig | falsch |
| a) | $1$ | $n$ | $n$ | $m$ |
| b) | $n - 1$ | $2$ | $n + m - 2$ | $2$ |
| c) | $n - 1$ | $2$ | $n + m - 3$ | $3$ |

## Aufgabe 4: Scheduling & Renaming

Gegeben sei ein Maschinenprogramm und ein superskalarer Prozessor mit zwei Integer-ALUs (IU) und zwei Load/Store-Einheiten (LS). Bis auf die Multiplikation benötigen alle Operationen nur einen Taktzyklus. Die Multiplikation erfordert zwei Taktzyklen.

| | | |
|---|---|---|
| 1. | LOAD | R1,[A] |
| 2. | LOAD | R2,[B] |
| 3. | MUL | R3,R1,R2 |
| 4. | LOAD | R1,[C] |
| 5. | LOAD | R2,[D] |
| 6. | MUL | R4,R1,R2 |
| 7. | ADD | R1,R3,R4 |
| 8. | STORE | [E],R1 |
| 9. | SUB | R2,R3,R4 |
| 10. | STORE | [F],R2 |

a) Wie viele Taktzyklen würden zur Ausführung des Maschinenprogramms auf einem skalaren Prozessor benötigt, der über nur je eine der o.g. Ausführungseinheiten verfügt und der diese nur nacheinander benutzen kann. Berücksichtigen nur die Zeiten, die von den Ausführungseinheiten benötigt werden!

b) Die Befehle sollen nun den o.g. Ausführungseinheiten des superskalaren Prozessors so zugeordnet werden, dass die Zeit zur Ausführung minimiert wird. Dabei darf zwar die vorgegebene Befehlsreihenfolge **nicht aber** die Registerzuordnung verändert werden (*Scheduling*).

c) Durch Umbenennung einiger Register (*register renaming*) kann die Ausführungszeit weiter verringert werden. Bestimmen Sie die maximal erreichbaren Beschleunigung und vergleichen Sie diesen Wert mit dem theoretischen Optimalwert.

## *Musterlösung:*

a) Bei sequenzieller Ausführung würden 12 Taktzyklen benötigt.

b) Es ergibt sich folgende Tabelle für die Zuordnung der Maschinenbefehle zu den Ausführungseinheiten:

| Takt | IU1 | IU2 | LS1 | LS2 |
|---|---|---|---|---|
| 1 | | | LOAD R1,[A] | LOAD R2,[B] |
| 2 | MUL R3,R1,R2 | | | |
| 3 | MUL R3,R1,R2 | | | |
| 4 | | | LOAD R1,[C] | LOAD R2,[D] |
| 5 | MUL R4,R1,R2 | | | |
| 6 | MUL R4,R1,R2 | | | |
| 7 | ADD R1,R3,R4 | SUB R2,R3,R4 | | |
| 8 | | | STORE [E],R1 | STORE [F],R2 |

Die Befehle 1 und 2 können parallel ausgeführt werden. Der Befehl 3 belegt die IU1 für zwei Taktzyklen. Die beiden LOAD-Befehle 4 und 5 können wegen der WAR-Abhängigkeit zu R1 bzw. R2 erst im 4. Taktzyklus ausgeführt werden. Wegen der RAW-Abhängigkeit bzgl. Register R4 zwischen dem Befehl 6 zu den Befehlen 7 und 9 können die beiden letztgenannten Be-

fehle erst im 7. Taktzyklus ausgeführt werden. Die Befehle 8 und 9 können vertauscht werden. Daher werden im 7. Taktzyklus beide IUs ausgelastet. Wegen der RAW-Abhängigkeiten bzgl. der Register R1 und R2 werden die beiden STORE-Befehle 8 und 10 erst im 8. Taktzyklus ausgeführt.

c) Um den WAR-Konflikt zwischen dem Befehl 3 und den LOAD-Befehlen 4 und 5 aufzulösen, kann man die betroffenen Register R1 und R2 in R11 und R22 umbenennen. Damit erhalten wir folgende Zuordnungstabelle:

| Takt | IU1 | IU2 | LS1 | LS2 |
|---|---|---|---|---|
| 1 | | | `LOAD R1,[A]` | `LOAD R2,[B]` |
| 2 | `MUL R3,R1,R2` | | `LOAD R11,[C]` | `LOAD R22,[D]` |
| 3 | `MUL R3,R1,R2` | `MUL R4,R11,R22` | | |
| 4 | | `MUL R4,R11,R22` | | |
| 5 | `ADD R1,R3,R4` | `SUB R2,R3,R4` | | |
| 6 | | | `STORE [E],R1` | `STORE [F],R2` |

Wir erkennen, dass durch die Registerumbenennung zwei weitere Taktzyklen eingespart werden können und dass damit die Auslastung der Ausführungseinheiten erhöht wird. Gegenüber der sequenziellen Lösung wird die Ausführungszeit halbiert, d.h. wir erhalten einen Speedup von 2. Dieser ist jedoch nur halb so groß wie der theoretische Maximalwert von 4 entsprechend der vier Ausführungseinheiten, die jedoch wegen der Datenabhängigkeiten nicht ständig parallel arbeiten können.

## Aufgabe 5: Cache

Gegeben seien drei Cache-Speicher **DM**, **A2** und **AV**, die jeweils acht Cacheblöcke besitzen, wobei jeder Cacheblock vier Bytes umfasst. Der Cache **DM** ist als direkt-abgebildeter Cache (*direct mapped*) organisiert, Cache **A2** als 2-fach assoziativer Cache (*2-way-set-associative*); Cache **AV** ist vollassoziativ (*fully associative*). Bei den Cachespeichern **A2** und **AV** soll die *„least recently used"* -Ersetzungsstrategie LRU angewendet werden. Nehmen Sie an, die Cachespeicher seien zu Beginn leer, und es soll eine Serie von einzelnen Bytes mit den folgenden 16-Bit-Adressen (**dezimal**!) gelesen werden:

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 70 | 09 | 39 | 83 | 66 | 68 | 35 | 80 | 93 | 67 | 79 | 37 | 84 | 09 |

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | | | | | | | | | 71 | 32 |
| 10 | 78 | 69 | | | | | | | | |
| 20 | | | | | | | | | | |
| 30 | | | 6 | 12 | 20 | 00 | 72 | 67 | 73 | 76 |
| 40 | | | | | | | | | | |
| 50 | | | | | | | | | | |
| 60 | | | | | 90 | 82 | 69 | 72 | 19 | 10 |
| 70 | 62 | 0 | | | | | 67 | 69 | 85 | 76 |
| 80 | 78 | 85 | 87 | 75 | 32 | 72 | 67 | 83 | | |
| 90 | | | 76 | 76 | 79 | 84 | | | | |

Speicherbelegung

Die linke Tabelle gibt den für diese Aufgabe relevanten Inhalt des Hauptspeichers im Adressbereich 0 bis 99 wieder.

a) Geben Sie für die drei Cachespeicher an, wie viele Bits zur Verwaltung eines Cacheblocks benötigt werden. Dabei sollen für den Zustand des Cacheblocks zwei Statusbits verwendet werden (Valid-Bit und Dirty-Bit).

b) Geben Sie für die drei Cachespeicher die Anzahl der erforderlichen Vergleicher und die jeweils zu vergleichende Bitanzahl an.

c) Welche Zeilen sind bei diesen Caches hinsichtlich einer Blockersetzung gleichzeitig auszuwerten, z.B. für die Ersetzungsstrategie nach dem LRU- Prinzip?

d) Geben Sie nun tabellarisch für jeden Cache an, ob es sich beim Lesezugriff auf die jeweilige Adresse um einen Treffer (Cache-Hit) oder um keinen Treffer (Cache-Miss) handelt.

e) Stellen Sie den Zustand der drei Caches nach dem letzten Speicherzugriff dar, d.h. für jeden Cacheblock den Cache-Tag und die vier Datenbytes. Entnehmen Sie die jeweils vier Datenbytes aus der obigen Speicherbelegung.

## *Musterlösung:*

a) Aus den folgenden Bildern kann man ablesen, wie viele Bits jeweils auf den Tag entfallen. Zusammen mit den beiden Zustandsbits (Valid und Dirty) werden daher 13 Bits beim DM-Cache, 14 Bits beim A2-Cache und 16 Bits beim AV-Cache für die Verwaltung eines Cacheblocks benötigt.

| | | | |
|---|---|---|---|
| DM | 15 … 5 | 4 … 2 | 1 … 0 |
| | Tag | Index | Offset |
| A2 | 15 … 4 | 3 … 2 | 1 … 0 |
| | Tag | Index | Offset |
| AV | 15 … 2 | | 1 … 0 |
| | Tag | | Offset |

b) Beim DM-Cache ist ein einziger 11-Bit-Vergleicher, beim A2-Cache sind zwei 12-Bit-Vergleicher, und beim AV-Cache sind acht 14-Bit-Vergleicher erforderlich.

c) Beim DM-Cache: nur die durch den Zeilenindex angewählte Zeile
Beim A2-Cache: jeweils die zwei durch den Satzindex angewählten Zeilen
Beim AV-Cache: Zeile 0 bis Zeile 7

d) Cache-Hit (X) oder Cache-Miss (--) beim Lesezugriff auf die jeweilige Adresse:

| Adresse | 70 | 9 | 39 | 83 | 66 | 68 | 35 | 80 | 93 | 67 | 79 | 37 | 84 | 9 | *Hits* |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Block-Nr.* | 11 | 02 | 09 | 14 | 10 | 11 | 08 | 14 | 17 | 10 | 13 | 09 | 15 | 2 | |
| DM-Cache | -- | -- | -- | -- | -- | -- | -- | X | -- | -- | -- | -- | -- | X | 2x |
| Index | 1 | 2 | 1 | 4 | 0 | 1 | 0 | 4 | 7 | 0 | 3 | 1 | 5 | 2 | |
| A2-Cache | -- | -- | -- | -- | -- | X | -- | -- | -- | -- | -- | X | -- | X | 3x |
| Index | 1 | 2 | 1 | 0 | 0 | 1 | 0 | 0 | 3 | 0 | 3 | 1 | 1 | 2 | |
| AV-Cache | -- | -- | -- | -- | -- | X | -- | X | -- | X | -- | X | -- | -- | 4x |

e) Zustand der drei Caches nach dem letzten Speicherzugriff:

**DM**

| Index | Wort 3 | Wort 2 | Wort 1 | Wort 0 | Tag |
|---|---|---|---|---|---|
| 0 | 72 | 69 | 82 | 90 | 2 |
| 1 | 76 | 73 | 67 | 72 | 1 |
| 2 | 69 | 78 | 32 | 71 | 0 |
| 3 | 76 | 85 | 69 | 67 | 2 |
| 4 | 75 | 87 | 85 | 78 | 2 |
| 5 | 83 | 67 | 72 | 32 | 2 |
| 6 | -- | -- | -- | -- | -- |
| 7 | 84 | 79 | 76 | 76 | 2 |

**A2**

| Index | Wort 3 | Wort 2 | Wort 1 | Wort 0 | Tag |
|---|---|---|---|---|---|
| 0 | 72 | 69 | 82 | 90 | 4 |
| | 75 | 87 | 85 | 78 | 5 |
| 1 | 83 | 67 | 72 | 32 | 5 |
| | 76 | 73 | 67 | 72 | 2 |
| 2 | 69 | 78 | 32 | 71 | 0 |
| | -- | -- | -- | -- | -- |
| 3 | 84 | 79 | 76 | 76 | 5 |
| | 76 | 85 | 69 | 67 | 4 |

**AV**

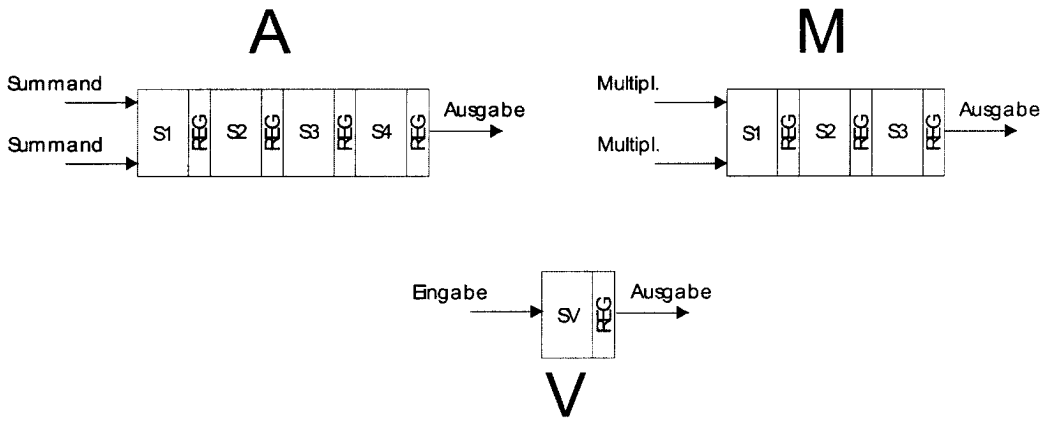
| Wort 3 | Wort 2 | Wort 1 | Wort 0 | Tag |
|---|---|---|---|---|
| 69 | 78 | 32 | 71 | 2 |
| 83 | 67 | 72 | 32 | 21 |
| 76 | 73 | 67 | 72 | 9 |
| 75 | 87 | 85 | 78 | 20 |
| 72 | 69 | 82 | 90 | 16 |
| 00 | 20 | 12 | 6 | 8 |
| 84 | 79 | 76 | 76 | 23 |
| 76 | 85 | 69 | 67 | 19 |

* Die Blocknummer ergibt sich aus der Adresse ohne die Wortadresse (*offset*) in hexadezimaler Schreibweise

## Aufgabe 6: Pipelineprozessor

Als Bauelemente für einen Pipelineprozessor seien folgende Funktionseinheiten gegeben:

- dreistufige Pipeline-Gleitkomma-Multiplizierer M,
- vierstufige Pipeline-Gleitkomma-Addierer A
- und Verzögerungselemente V.




a) Konstruieren Sie mit den oben gegebenen Elementen[1] eine synchrone Pipeline (alle Register werden mit demselben Taktsignal angesteuert), die aus einem Eingabestrom *e(1),e(2),e(3),...* von Gleitkommazahlen die Ausgabewerte:

$$a(j)=e(i)+e(i+1)+e(i+2)\cdot e(i+3)$$

berechnet. Hierbei soll *e(i)* während des *i*-ten Taktzyklus anliegen. Zeichnen Sie ein Blockdiagramm!

b) Die Stufen eines Multiplizierers bzw. eines Addierers besitzen folgende Verzögerungszeiten $\tau_{mi}$ bzw. $\tau_{ai}$:

| $\tau$ | S1 | S2 | S3 | S4 |
|---|---|---|---|---|
| Addierer | 12 ns | 14 ns | 17 ns | 11 ns |
| Multiplizierer | 13 ns | 16 ns | 15 ns | |

Die Verzögerungszeiten der Register bzw. der Verzögerungselemente beträgt $\tau_{reg}$=3 ns und $\tau_v$=2ns. Wie groß ist die maximale Taktfrequenz *f* dieser Pipeline?

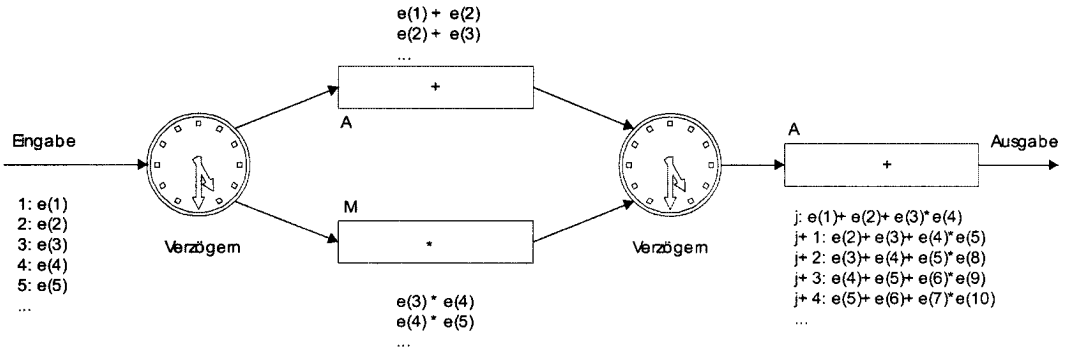
c) Wie viele Verarbeitungsaufträge *n* können während einer Zeitspanne von 660 ns ausgeführt werden? Berechnen Sie den zugehörigen Durchsatz $D_k$ (Anzahl Verarbeitungsaufträge / 660 ns). Ein Taktzyklus $T_k$ dauert $1/f$ Sekunden. Setzen Sie voraus, dass die Pipeline ständig gefüllt ist.

## *Musterlösung:*

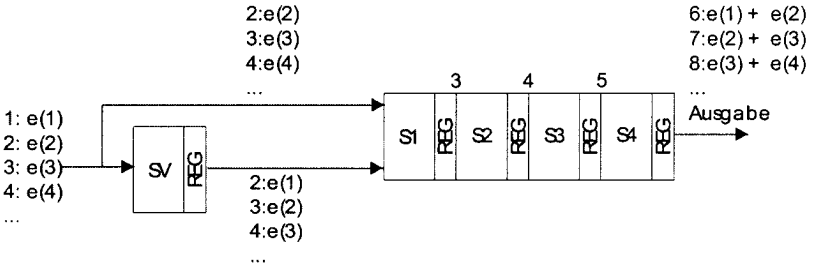
---

[1] es dürfen durchaus mehrere Komponenten des gleichen Typs verwendet werden.

a) Die Berechnung von $a(j)=e(i)+e(i+1)+e(i+2)\cdot e(i+3)$ soll so aufgeteilt werden, dass mit den gegebenen Komponenten M, A und V ein möglichst hoher Parallelisierungsgrad erreicht wird. Es liegt daher nahe, die Berechnung von $a(j)$ in drei Teilaufgaben zu zerlegen, deren jeweilige Ergebnisse durch eine geeignete Anzahl und Anordnung von Verzögerungsgliedern synchronisiert wird.

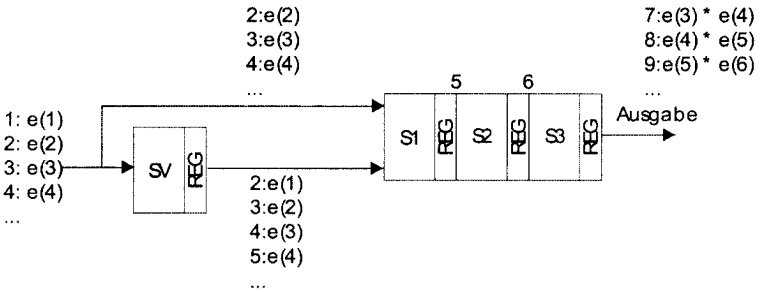



Betrachten wir zuerst den Addierer, welcher $e(i)+e(i+1)$ berechnet. Dieser Addierer benötigt an seinen beiden Eingängen $e(i)$ und $e(i+1)$; man muss also einen Eingang des Addierers mit einem Verzögerungsglied V versehen und es ergibt sich:

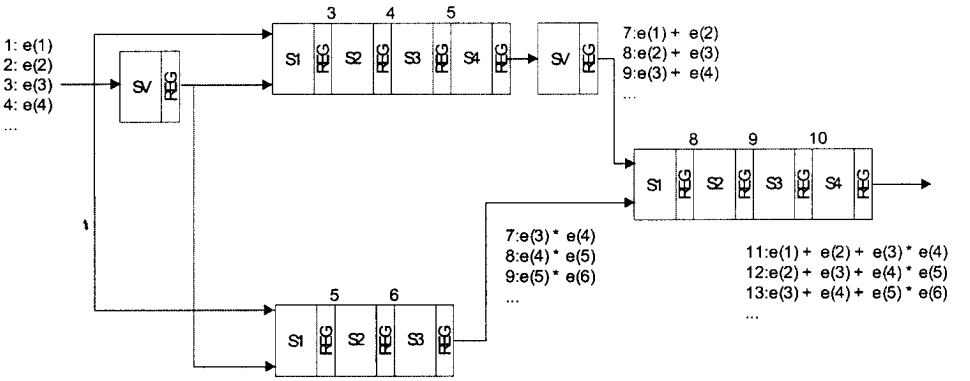



Die Addiererpipeline liefert also ab dem 6. Takt "sinnvolle" Ergebnisse.

Analog dazu wird die Multiplizierstufe behandelt:




Ab dem 4. Takt liegen die gewünschten Eingabewerte $e(3)$ und $e(4)$ an den Eingängen der Multiplizierstufe an. Nach dem 7.Takt ist das Ergebnis $e(3)\cdot e(4)$ berechnet. Bisher ergibt sich also, dass $e(i)+e(i+1)$ nach dem $i$+5 ten Takt und $e(i+2)\cdot e(i+3)$ nach dem $i$+6 ten Takt vorliegt. $e(i)+e(i+1)$ muss demzufolge noch einmal vor der Addition mit $e(i+2)\cdot e(i+3)$ verzögert werden. Insgesamt sieht das Blockschaltbild wie folgt aus:




Für dieses synchrone Pipeline gilt $j=i+10$, es liegt eine 10-stufige Pipeline vor.

b) Die maximal erreichbare Taktfrequenz $f$ ist abhängig von der langsamsten Komponente. Hier ist es die Stufe *S3* der Addiererpipeline mit 17 ns. Hinzukommt für diese Stufe noch das Register mit 3 ns. Somit darf die Taktperiode nicht kürzer als $\tau_{max} = 17\ ns + 3\ ns = 20\ ns$ sein. Die maximale Taktfrequenz ergibt sich aus dem Kehrwert von $\tau_{max}$ zu **1/20 ns = 50 MHz.**

*c)* Unter der Voraussetzung, dass die Pipeline immer gefüllt ist, kann ein Pipelineprozessor mit $k$ Stufen $n$ Verarbeitungsaufträge in $Z_k=k+(n-1)$ Taktzyklen vollständig ausführen. Umstellen nach $n$ ergibt $n=Z_k-k+1$. Für die hier gegebene Pipeline ergibt sich mit $k$=10 und $Z_k$=660 ns / 20 ns = 33 für $n$: $n$ = 33-10+1 = 24 Verarbeitungsaufträge. Der Durchsatz $D_{10}$= 24/660 ns ≅ 36363636 ≅ 36 MIPs.