Aufgabe 2.08: Cache

Gegeben sei ein Cache mit 4 Cache-Zeilen und 4 Bytes pro Cache-Zeile. Zum Vergleich ist der Cache als Direct-Mapped, 2-Wege Assoziativ und Voll-Assoziativ vorhanden. Die Speicheradresse beträgt 8-Bits.

a) Wie ist die Aufteilung der Speicheradresse bei den Verschiedenen Cache-Typen. Pro Bit der Speicheradresse ist ein Kästchen vorhanden. Jedes Bit kann entweder für den Tag (t), Index (i) oder Byteoffset (o) verwendet werden:

Direct-Mapped

t	t	t	t	i	i	o	o

2-Wege Assoziativ

t	t	t	t	t	i	o	o

Voll-Assoziativ

t	t	t	t	t	t	o	o

b) Es werden jetzt nacheinander 6 Lesezugriffe auf den jeweiligen Cache-Typen ausgeführt. Die Speicheradressen der Lesezugriffe sind in Dualform in der ersten Spalte vorhanden. In der 2ten Spalte (Hit?) werden Cache-Hits eingetragen. Die restlichen Spalten sind für den Inhalt der 4 Cache-Zeilen vorgesehen. Als Inhalt genügt es den Tag der Cache-Zeile einzutragen. Als Verdrängungsstrategie ist LRU (Least Recently Used) vorgesehen.

	Hit?	**Line 0**	**Line 1**	**Line 2**	**Line 3**
1000 0011	-	1000			
1000 0100	-	1000	1000		
1000 1001	-	1000	1000	1000	
1000 0010	x	1000	1000	1000	
1001 0010	-	1001	1000	1000	
1000 1000	x	1001	1000	1000	

Tabelle 2.6: Direct-mapped cache

	Hit?	**Line 0**	**Line 1**	**Line 2**	**Line 3**
1000 0011	-	10000			
1000 0100	-	10000		10000	
1000 1001	-	10000	10001	10000	
1000 0010	x	10000	10001	10000	
1001 0010	-	10000	10010	10000	
1000 1000	-	10001	10010	10000	

Tabelle 2.7: 2-Wege assoziativer Cache

	Hit?	**Line 0**	**Line 1**	**Line 2**	**Line 3**
1000 0011	-	100000			
1000 0100	-	100000	100001		
1000 1001	-	100000	100001	100010	

1000 0010	x	100000	100001	100010	
1001 0010	-	100000	100001	100010	100100
1000 1000	x	100000	100001	100010	100100

Tabelle 2.8: Voll-assoziativer Cache

c) Wieviel Speicherplatz verbraucht ein Cache insgesamt? Geben sie ihre Antwort in Abhängigkeit der ihnen bekannten Cache-Parameter an.

Es wird Speicherplatz für die Adresse und die eigentlichen Daten benötigt, die der Cache zwischenspeichern soll. Beides muss jeweils für alle vorhandenen Cache-lines vorgehalten werden. Für die Adresse reicht es dabei aus den Tag zu speichern, da Index und offset direkt für die Ansteuerung der Cache-line bzw des Datums in der Cache-line genutzt werden und daher nicht vorgemerkt werden müssen. Der Speicherplatz für die Daten wiederum hängt von der kleinsten Adressierbaren Größe (üblicherweise ein Byte) und der Anzahl von Daten in einer Cache-line ab (also wieviele dieser Adressen in einer Zeile gemeinsam geladen werden).

d) Eine Matrixmultiplikation wird üblicherweise mit Hilfe von zwei verschachtelten Schleifen implementiert die über alle Elemente der Matrix laufen (x,y). Wieso gibt es einen riesigen Laufzeitunterschied auf einem GPP mit einem Datencache, je nachdem über welche Dimension zuerst (d.h. die äußere Schleife) iteriert wird?

Eine Matrix wird meist über ein zweidimensionales Array implementiert, welches kontinuierlich im Speicher liegt. Wenn die Matrixmultiplikation in der richtigen Reihenfolge durchgeführt wird, so wird bereits mit der ersten Operation eine Cachezeile geladen, in welcher die Operanden für die nächste Operation mit enthalten sind. Wird sie falsch herum durchgeführt, so findet bei jeder Operation ein Cache-miss statt und die Operanden müssen einzeln über den langsamen Hauptspeicher gefetched werden.

Aufgabe 2.09: DSP

Im Folgenden ist eine Implementierung eines FIR-Filters als C-Code gegeben:

```
sum = 0.0;
for (i=0; i<N; i++)
sum = sum + a[i]*b[i];
```

Ein solcher Code kann auf einem DSP unter Ausnutzung der für solche Domänen angepassten Spezialhardware sehr effizient ausgeführt werden. Im Folgenden ist der Code zu sehen, nachdem er optimiert in Assembler übersetzt wurde:

```
RPTS   N -1                ;Repeat next instruction.
MPYF3  *AR0++%,*AR1++%,R0  ;Multiply…
|| ADDF3  R0,R2,R2         ;… and accumulate.
ADDF   R0,R2               ;Last product accumulated.
```

Der Code nutzt für die Ausführung viele Optimierungen eines DSPs: Es werden zero-overhead loops (RPTS), Circular addressing (*AR0++%) und Multiply-accumulate (||)

eingesetzt. Die gesamte MAC Instruktion wird dabei pipelined ausgeführt und kann daher pro Takt eine Iteration berechnen und aufsummieren.

a) Wie viele Takte benötigt der optimierte Assembler Code für N = 10?

   12 Takte

b) Schreiben sie nun das Programm in reinem Assembler ohne alle Spezialbefehle und DSP-Optimierungen. Nutzen sie nur die im Folgenden angegebenen Instruktionen. Wie viele zusätzliche Takte braucht das Programm hierdurch für N = 10?

Für die Aufgabe stehen ihnen die folgenden Assemblerbefehle zur Verfügung. Als Quelle „src“ können sowohl Register als auch konstante Werte direkt genutzt werden. Weiterhin können sie davon ausgehen, dass alle Register bereits mit dem Wert 0 initialisiert sind und die Startadressen der beiden Eingabearrays a[] und b[] bereits in den Registern R1 bzw R2 liegen.

LDF	src, dst	Läd den floating-point Wert an der Adresse src in das Register dst
ADDF3	src2, src1, dst	Addiert die floating-point Werte aus den Registern src1 und src2 und speichert das Ergebnis in dst
ADDI3	src2, src1, dst	Addiert die Integer Werte aus den Registern src1 und src2 und speichert das Ergebnis in dst
MPYF3	src2, src1, dst	Multipliziert die Floating-point Werte aus den Registern src1 und src2 und speichert das Ergebnis in dst
CMPI3	src2, src1	Berechnet src1-src2 und setzt das Status bit „Z“ auf 1, falls das Ergebnis der Subtraktion 0 ist
BZ	src	Bedingter Sprung zu einem Label oder um einem relativen Wert, sofern im Statusbit „Z“ eine 1 steht
BR	src	Unbedingter Sprung zu einem Label oder um einen relativen Wert

```
Loop:  CMPI3  R0,  10
       BZ     9
       LDF    R1,  R3
       LDF    R2,  R4
       MPYF3  R3,  R4,  R5
       ADDF3  R5,  R6,  R6
       ADDI3  R0,  1,   R0
       ADDI3  R1,  4,   R1
       ADDI3  R2,  4,   R2
       BR     Loop
```

Ohne alle Optimierungen: 102 Takte

Aufgabe 2.10: FPGA

Gegeben ist eine einfache Version eines Xilinx FPGAs bestehend aus CLBs und Switch-Matrixs. Ein CLB enthalt eine 8 elementige LUT mit 3 Eingängen.

a) Realisieren Sie einen Volladdierer mittels zwei CLBs.

Ein Volladdierer besitzt drei Eingänge (x, y, cin) und zwei Ausgänge (cout, s). Es werden sämtliche Eingänge addiert und das Ergebnis als 2-Bit Zahl auf die Ausgänge gelegt. So ist s=1 wenn die Summe aller Eingänge ungerade ist und cout =1 wenn die Summe aller Eingänge >= 2 ist.

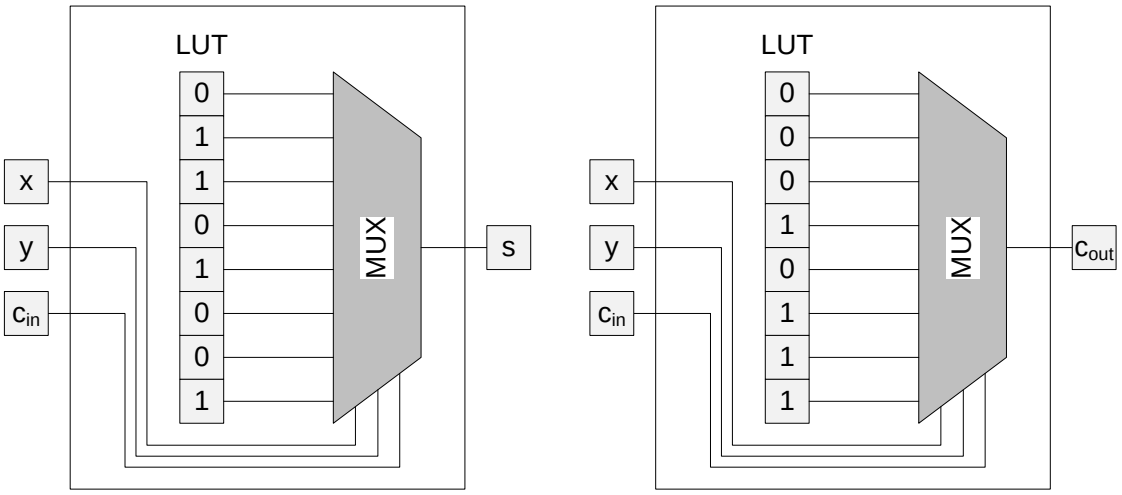

Figure 2.1: LUT Realisierung für einen Volladdierer