

# Übung zur Vorlesung Einführung in Computer Microsystems

Prof. Dr.-Ing. A. Koch  
Jaco Hofmann, MSc.



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

## Sommersemester 15 Übungsblatt 4 - Lösungsvorschlag

Diese Übung hat die Implementierung eines „direct-mapped-cache“ in Bluespec zum Inhalt. Zuerst wird eine Testumgebung mit RAM und entsprechenden Hilfsmodulen erstellt. In dieser wird der Cache implementiert und schließlich getestet.

Diese Übung erfordert ein erhöhtes Maß an eigenständiger Leistung. Wenn Sie bei der Bearbeitung der Aufgaben Probleme haben können Sie gerne das Forum oder die Sprechstunde nutzen. Zusätzlich werden eine Woche vor der Besprechung der Aufgabe Teile des Lösungsvorschlags veröffentlicht um Ihnen die Bearbeitung der Kernaufgabe (Cache implementieren) zu vereinfachen.

### Aufgabe 4.1 Cache

Ein „direct-mapped-cache“ ist die einfachste Form eines Caches. Jede Speicheradresse des übergeordneten Speichers kann an genau einer Position des Caches abgelegt werden.

Zusätzlich speichert der Cache immer eine Cache-Line die aus mehreren Wörtern besteht anstatt nur ein Wort zu speichern. Dies wird gemacht um räumliche Lokalität zwischen nahe im Speicher liegenden Daten auszunutzen. Eine typische Cache-Line besteht aus 16 Wörtern. Bei einem Byte-adressierten Speicher mit 4B pro Wort ergeben sich somit 64B pro Cache-Line. Dementsprechend benötigt man mindestens 6 bit um die Position eines Bytes in der Cache-Line eindeutig zu bestimmen.

Der Cache hat eine festgelegte Anzahl an Plätzen die „Slot“ genannt werden. Jede Speicheradresse kann, wie oben erwähnt, in genau einen dieser Slots geladen werden. Für dieses Beispiel nehmen wir an der Cache habe 128 Slots. Dementsprechend benötigt man 7 bit um alle Slots eindeutig kodieren zu können.

Angenommen die Speicheradresse besteht aus 32 bit dann werden die unteren 6 bit genutzt um das Wort in der Cache-Line zu spezifizieren, die darüber liegenden 7 bit spezifizieren den Slot und die restlichen 19 bit der Adresse sind das sogenannte Tag.

Wird eine Speicheranfrage an den Cache gestellt wird das Tag der in dem zur Speicheranfrage gehörenden Slot mit dem Tag der Speicheranfrage verglichen. Sind diese gleich gibt es einen so genannten Cache-Hit und das Wort mit dem zur Speicheranfrage gehörenden Offset wird zurückgegeben. Passen die Tags nicht zusammen gibt es einen Cache-Miss und die entsprechende Cache-Line muss aus dem höherliegenden Speicher geladen werden.

Der zu implementierende Cache soll dabei die Schreibstrategien „write-back“ und „write-allocate“ implementieren. Das heißt ein Schreibzugriff wird erst lokal im Cache ausgeführt und erst zurück in den höherliegenden Speicher geschrieben wenn die Cache-Line verdrängt wird. Dementsprechend muss vor dem Schreiben die korrekte Cache-Line im Cache vorliegen.

Weitere Informationen finden Sie in den GDI 3 (<https://www.ra.informatik.tu-darmstadt.de/lehre/gdi-iii/>) Folien unter „Speicherhierarchie 1-3“.

#### Aufgabe 4.1.1 RAM

Implementieren Sie einen Byte-adressierbaren RAM mit 1 048 576 B Größe. Verwenden Sie dafür das BRAM Modul aus der letzten Übung. Die Adressbreite soll 32 bit betragen. Geben Sie immer an Wortgrenzen angeordnete Wörter zurück (Anfrage für Adresse 0x03 würde das Wort, das an Speicherstelle 0x00 startet, zurückgeben). Die Wortbreite soll 32 bit betragen. Verwenden Sie einen RAM mit Byte-Enable, damit Sie wählen können, dass nur Teile von einem Wort geschrieben werden.

Überlegen Sie sich ein sinnvolles Interface (z.B. ClientServer mit entsprechenden Anfragen und Rückgaben).

Zusatzaufgabe: Implementieren Sie das Modul generisch für beliebige Wortbreiten.

<sup>1</sup> package RAM;

<sup>2</sup>

## Übung zur Vorlesung Einführung in Computer Microsystems

```
3     import BRAM ::*;
4     import GetPut :: *;
5     import ClientServer :: *;
6     import Connectable :: *;
7     import FIFO :: *;
8     import SpecialFIFOs :: *;
9     import Vector :: *;
10
11    typedef union tagged {Bit#(32) Read;
12                           Tuple3#(Bit#(32), Bit#(cacheLineBits), Bit#(TDiv#(cacheLineBits, 8))) Write;
13                         }
14                           RAMRequest#(numeric type wordBits, numeric type cacheLineBits)
15                         deriving (Bits, Eq, FShow);
16
17    typedef Server#(RAMRequest#(wordBits, cacheLineBits), Bit#(cacheLineBits))
18      RAMServer#(numeric type wordBits, numeric type cacheLineBits);
19    typedef Client#(RAMRequest#(wordBits, cacheLineBits), Bit#(cacheLineBits))
20      RAMClient#(numeric type wordBits, numeric type cacheLineBits);
21
22    module mkRAM(RAMServer#(wordBits, cacheLineBits))
23      provisos(
24        Mul#(cacheLineBytes, 8, cacheLineBits),
25        Div#(cacheLineBits, 8, cacheLineBytes),
26        Log#(cacheLineBytes, cacheLineAddrBits),
27        Div#(cacheLineBits, cacheLineBytes, 8),
28        Add#(cacheLineAddrBits, addrBitsWithoutCacheLine, 32),
29        Add#(18, a__, addrBitsWithoutCacheLine),
30        Div#(wordBits, 8 , wordBytes),
31        Log#(wordBytes, wordAddrBits),
32        Add#(wordAddrBits, cacheLineToWord, cacheLineAddrBits),
33        Add#(cacheLineToWord, b__, 32)
34      );
35      FIFO#(RAMRequest#(wordBits, cacheLineBits)) requestIn <- mkBypassFIFO();
36      FIFO#(Bit#(cacheLineBits)) dataOut <- mkBypassFIFO();
37
38      BRAM_Configure conf = defaultValue;
39      conf.memorySize = 262144;
40
41      BRAM1PortBE#(Bit#(18), Bit#(cacheLineBits), cacheLineBytes) bram <- mkBRAM1ServerBE(conf);
42
43      rule handleRead (requestIn.first() matches tagged Read .v);
44        requestIn.deq();
45        Bit#(addrBitsWithoutCacheLine) addr_l = truncateLSB(v);
46        Bit#(18) addr = truncate(addr_l);
47        BRAMRequestBE#(Bit#(18), Bit#(cacheLineBits), cacheLineBytes) req =
48          BRAMRequestBE{writeen: 0, responseOnWrite: False, address: addr, datain: 0};
49        bram.portA.request.put(req);
50      endrule
51
52      rule handleReadResponse;
53        let response <- bram.portA.response.get();
54        dataOut.enq(extend(response));
55      endrule
56
57      rule handleWrite (requestIn.first() matches tagged Write .v);
```

## Übung zur Vorlesung Einführung in Computer Microsystems

```
58     requestIn.deq();
59     Bit#(addrBitsWithoutCacheLine) addr_l = truncateLSB(tpl_1(v));
60     Bit#(18) addr = truncate(addr_l);
61     if(tpl_3(v) != 0) begin
62         BRAMRequestBE#(Bit#(18), Bit#(cacheLineBits), cacheLineBytes) req =
63             BRAMRequestBE{writeen: tpl_3(v), responseOnWrite: False,
64                         address: addr, datain: tpl_2(v)};
65         bram.portA.request.put(req);
66     end
67   endrule
68
69   interface Put request = toPut(requestIn);
70   interface Get response = toGet(dataOut);
71 endmodule
72 endpackage
```

### Aufgabe 4.1.2 Hilfsmodule

Schreiben Sie zwei Hilfsmodule zum Umgang mit Cache-Lines. Das erste Modul soll den in der vorherigen Aufgabe implementierten RAM nutzen und Cache-Lines mit 512bit zur Verfügung stellen. Das zweite Modul soll auf der einen Seite Anfragen mit Wortbreite annehmen und auf der anderen Seite Anfragen mit Cache-Line Breite ausgeben.

Zusatzaufgabe: Implementieren Sie die Module generisch für beliebige Wort- und Cache-Line-Breiten.

Cache-Line → Word

```
1 package CacheLineToWord;
2
3   import RAM ::::*;
4   import GetPut :: *;
5   import ClientServer :: *;
6   import Connectable :: *;
7   import FIFO :: *;
8   import SpecialFIFOs :: *;
9   import Vector :: *;
10
11   interface CacheLineToWord#(numeric type wordBits, numeric type cacheLineBits);
12     interface RAMClient#(wordBits, cacheLineBits) toRAM;
13     interface RAMServer#(wordBits, wordBits) toUser;
14   endinterface
15
16   module mkCacheLineToWord(CacheLineToWord#(wordBits, cacheLineBits))
17     provisos(Div#(cacheLineBits, 8, cacheLineBytes),
18               Log#(cacheLineBytes, cacheLineAddrBits),
19               Add#(remainderOfAddress, cacheLineAddrBits, 32),
20               Mul#(wordsPerCacheLine, wordBits, cacheLineBits),
21               Div#(wordBits, 8, wordBytes),
22               Log#(wordBytes, wordAddrBits),
23               Div#(cacheLineBits, wordBits, wordsPerCacheLine)
24             );
25
26     FIFO#(RAMRequest#(wordBits, wordBits)) requestToRam <- mkBypassFIFO();
27     FIFO#(Bit#(wordBits)) responseToUser <- mkBypassFIFO();
28
29     Reg#(Bit#(TSub#(cacheLineAddrBits, wordAddrBits))) selectedWord <- mkReg(0);
30     Reg#(Bool) transferActive[2] <- mkCReg(2, False);
31
32     interface Client toRAM;
```

Übung zur Vorlesung Einführung in Computer Microsystems

## Übung zur Vorlesung Einführung in Computer Microsystems

```
88         endinterface
89     endinterface
90 endmodule
91
92 endpackage
```

Word → Cache-Line

```
1 package CacheLine;
2
3     import RAM ::::*;
4     import GetPut :: *;
5     import ClientServer :: *;
6     import Connectable :: *;
7     import FIFO :: *;
8     import SpecialFIFOs :: *;
9     import Vector :: *;
10
11    interface CacheLineRAM#(numeric type wordBits, numeric type cacheLineBits);
12        interface RAMServer#(wordBits, cacheLineBits) toUser;
13    endinterface
14
15    module mkCacheLineRAM(CacheLineRAM#(wordBits, cacheLineBits))
16        provisos(
17            Mul#(cacheLineBytes, 8, cacheLineBits),
18            Div#(cacheLineBits, 8, cacheLineBytes),
19            Log#(cacheLineBytes, cacheLineAddrBits),
20            Div#(cacheLineBits, cacheLineBytes, 8),
21            Add#(cacheLineAddrBits, addrBitsWithoutCacheLine, 32),
22            Add#(18, a__, addrBitsWithoutCacheLine),
23            Div#(wordBits, 8 , wordBytes),
24            Div#(wordBits, wordBytes, 8),
25            Add#(wordAddrBits, b__, 32),
26            Add#(18, c__, b__),
27            Log#(wordBytes, wordAddrBits),
28            Div#(cacheLineBits, wordBits, wordsPerCacheLine),
29            Mul#(wordsPerCacheLine, wordBits, cacheLineBits)
30        );
31        FIFO#(RAMRequest#(wordBits, cacheLineBits)) requestIn <- mkFIFO();
32        FIFO#(Bit#(cacheLineBits))           dataOut   <- mkFIFO();
33
34        RAMServer#(wordBits, wordBits) ram <- mkRAM();
35        FetchCacheLine#(wordBits, cacheLineBits) fetcher <- mkFetchCacheLine();
36        mkConnection(ram, fetcher.toRAM());
37
38        Reg#(Bool)                      actionInProgress <- mkReg(False);
39        Reg#(Bool)                      read <- mkReg(False);
40        Reg#(Bool)                      fetchActive <- mkReg(False);
41
42        Reg#(Bit#(32))                  expectedCacheLineAddr <- mkReg(0);
43        Reg#(Bit#(cacheLineBits))       currentWriteCacheLine <- mkReg(0);
44        Reg#(Bit#(cacheLineBytes))      currentWriteEN <- mkReg(0);
45
46        Reg#(UInt#(32))                requestTimer <- mkReg(0);
47
48        rule countRequest (fetchActive);
```

## Übung zur Vorlesung Einführung in Computer Microsystems

```
49      requestTimer <= requestTimer + 1;
50  endrule
51
52  rule fetchCorrectLine(actionInProgress && !fetchActive && read);
53      fetchActive          <= True;
54      let req = tagged Fetch expectedCacheLineAddr;
55      fetcher.toUser.request.put(req);
56
57      requestTimer         <= 0;
58  endrule
59
60  rule forwardWriteRequest(actionInProgress && !fetchActive && !read);
61      let req = tagged JustWrite tuple3(expectedCacheLineAddr, currentWriteCacheLine, currentWriteEN);
62      fetcher.toUser.request.put(req);
63      actionInProgress <= False;
64  endrule
65
66  rule handleFetcherResponse (actionInProgress && fetchActive && read);
67      let fetchedLine  <- fetcher.toUser.response.get();
68      fetchActive     <= False;
69      dataOut.enq(fetchedLine);
70      actionInProgress <= False;
71  endrule
72
73
74  rule handleRead (requestIn.first() matches tagged Read .v &&& !actionInProgress);
75      requestIn.deq();
76      Bit#(cacheLineAddrBits) mask = (1 << valueOf(cacheLineBits)) - 1;
77      Bit#(32) lineAddress = v & ~zeroExtend(mask);
78
79      expectedCacheLineAddr    <= lineAddress;
80      read                     <= True;
81      actionInProgress        <= True;
82  endrule
83
84  rule handleWriteRequest (requestIn.first() matches tagged Write .v &&& !actionInProgress);
85      requestIn.deq();
86
87      let address = tpl_1(v);
88      Bit#(cacheLineAddrBits) mask = (1 << valueOf(cacheLineBits)) - 1;
89      Bit#(32) lineAddress = address & ~zeroExtend(mask);
90      let data   = tpl_2(v);
91      let writeen = tpl_3(v);
92      expectedCacheLineAddr    <= lineAddress;
93      currentWriteCacheLine   <= unpack(data);
94      currentWriteEN          <= unpack(writeen);
95      read                   <= False;
96      actionInProgress        <= True;
97  endrule
98
99  interface RAMServer toUser;
100     interface Put request  = toPut(requestIn);
101     interface Get response = toGet(dataOut);
102  endinterface
103 endmodule
```

## Übung zur Vorlesung Einführung in Computer Microsystems

```
104
105     typedef union tagged {
106         Bit#(32) Fetch;
107         Tuple3#(Bit#(32), Bit#(32), Bit#(cacheLineBits)) FetchDirty;
108         Tuple3#(Bit#(32), Bit#(cacheLineBits), Bit#(TDiv#(cacheLineBits,8))) JustWrite;
109     } FetchRequest#(numeric type cacheLineBits) deriving (Bits, Eq, FShow);
110
111     interface FetchCacheLine#(numeric type wordBits, numeric type cacheLineBits);
112         interface RAMClient#(wordBits, wordBits)                                     toRAM;
113         interface Server#(FetchRequest#(cacheLineBits), Bit#(cacheLineBits)) toUser;
114     endinterface
115
116     module mkFetchCacheLine(FetchCacheLine#(wordBits, cacheLineBits))
117         provisos(
118             Div#(cacheLineBits, wordBits, wordsPerCacheLine),
119             Div#(wordBits, 8, wordBytes),
120             Div#(cacheLineBits, 8, cacheLineBytes),
121             Bits#(Vector::Vector#(wordsPerCacheLine, Bit#(wordBits)), cacheLineBits),
122             Bits#(Vector::Vector#(wordsPerCacheLine, Bit#(wordBytes)), TDiv#(cacheLineBits, 8))
123             );
124         FIFO#(RAMRequest#(wordBits, wordBits)) requestOut <- mkBypassFIFO();
125         FIFO#(Bit#(wordBits))                           dataIn <- mkBypassFIFO();
126
127         Reg#(Bool)                               dirty           <- mkReg(False);
128         Reg#(Bool)                               justWrite      <- mkReg(False);
129         Reg#(Bool)                               fetchInProgress <- mkReg(False);
130         Reg#(UInt#(TAdd#(TLog#(wordsPerCacheLine),1))) currentWord    <- mkReg(0);
131         Reg#(UInt#(TAdd#(TLog#(wordsPerCacheLine),1))) requestedWords <- mkReg(0);
132         Vector#(wordsPerCacheLine, Reg#(Bit#(wordBits)))   currentCacheLine <- replicateM(mkRegU);
133         Vector#(wordsPerCacheLine, Reg#(Bit#(wordBytes)))   currentWriteEN <- replicateM(mkRegU);
134         Reg#(Bit#(32))                            readAddress    <- mkReg(0);
135         Reg#(Bit#(32))                            writeAddress   <- mkReg(0);
136
137         (* conflict_free="operation, operationRead" *)
138         rule operation (fetchInProgress && currentWord < fromInteger(valueOf(wordsPerCacheLine))
139                         && requestedWords < fromInteger(valueOf(wordsPerCacheLine)));
140             RAMRequest#(wordBits, wordBits) req = tagged Read 0;
141             if(dirty) begin
142                 writeAddress <- writeAddress + fromInteger(valueOf(wordBytes));
143                 currentWord <- currentWord + 1;
144                 Bit#(wordBytes) writeEN = 0;
145                 for(Integer i = 0; i < valueOf(wordBytes); i = i + 1) begin
146                     writeEN[i] = currentWriteEN[currentWord][i];
147                 end
148                 req = tagged Write tuple3(writeAddress, currentCacheLine[currentWord], writeEN);
149             end else begin
150                 readAddress <- readAddress + fromInteger(valueOf(wordBytes));
151                 requestedWords <- requestedWords + 1;
152                 req = tagged Read readAddress;
153             end
154             requestOut.enq(req);
155         endrule
156
157         rule operationRead (fetchInProgress && !dirty
158                         && currentWord < fromInteger(valueOf(wordsPerCacheLine)));
159     endrule
160
161     method Action write(Bit#(wordBytes) value);
162         currentCacheLine[currentWord] = value;
163         currentWriteEN[currentWord] = writeEN;
164     endaction
165
166     method Action read(Bit#(wordBytes) value);
167         value = currentCacheLine[currentWord];
168     endaction
169
170     method Action update(Bit#(wordBytes) value);
171         currentCacheLine[currentWord] = value;
172         currentWriteEN[currentWord] = writeEN;
173     endaction
174
175     method Action flush();
176         currentCacheLine = Vector();
177         currentWriteEN = Vector();
178     endaction
179
180     method Action invalidate();
181         currentCacheLine = Vector();
182         currentWriteEN = Vector();
183         currentWord = 0;
184         requestedWords = 0;
185         dirty = False;
186         justWrite = False;
187         fetchInProgress = False;
188     endaction
189
190     method Action bypass();
191         currentCacheLine = Vector();
192         currentWriteEN = Vector();
193         currentWord = 0;
194         requestedWords = 0;
195         dirty = False;
196         justWrite = False;
197         fetchInProgress = False;
198     endaction
199
200     method Action bypassFIFO();
201         requestOut = FIFO();
202         dataIn = FIFO();
203     endaction
204
205     method Action enable();
206         fetchInProgress = True;
207     endaction
208
209     method Action disable();
210         fetchInProgress = False;
211     endaction
212
213     method Action setWordBits(Bit#(wordBits) value);
214         wordBits = value;
215     endaction
216
217     method Action setCacheLineBits(Bit#(cacheLineBits) value);
218         cacheLineBits = value;
219     endaction
220
221     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
222         TDiv#(cacheLineBits,8) = value;
223     endaction
224
225     method Action setBypassFIFO();
226         requestOut = FIFO();
227         dataIn = FIFO();
228     endaction
229
230     method Action setBypass();
231         currentCacheLine = Vector();
232         currentWriteEN = Vector();
233         currentWord = 0;
234         requestedWords = 0;
235         dirty = False;
236         justWrite = False;
237         fetchInProgress = False;
238     endaction
239
240     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
241         TDiv#(cacheLineBits,8) = value;
242     endaction
243
244     method Action setWordBits(Bit#(wordBits) value);
245         wordBits = value;
246     endaction
247
248     method Action setCacheLineBits(Bit#(cacheLineBits) value);
249         cacheLineBits = value;
250     endaction
251
252     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
253         TDiv#(cacheLineBits,8) = value;
254     endaction
255
256     method Action setBypassFIFO();
257         requestOut = FIFO();
258         dataIn = FIFO();
259     endaction
260
261     method Action setBypass();
262         currentCacheLine = Vector();
263         currentWriteEN = Vector();
264         currentWord = 0;
265         requestedWords = 0;
266         dirty = False;
267         justWrite = False;
268         fetchInProgress = False;
269     endaction
270
271     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
272         TDiv#(cacheLineBits,8) = value;
273     endaction
274
275     method Action setWordBits(Bit#(wordBits) value);
276         wordBits = value;
277     endaction
278
279     method Action setCacheLineBits(Bit#(cacheLineBits) value);
280         cacheLineBits = value;
281     endaction
282
283     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
284         TDiv#(cacheLineBits,8) = value;
285     endaction
286
287     method Action setBypassFIFO();
288         requestOut = FIFO();
289         dataIn = FIFO();
290     endaction
291
292     method Action setBypass();
293         currentCacheLine = Vector();
294         currentWriteEN = Vector();
295         currentWord = 0;
296         requestedWords = 0;
297         dirty = False;
298         justWrite = False;
299         fetchInProgress = False;
300     endaction
301
302     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
303         TDiv#(cacheLineBits,8) = value;
304     endaction
305
306     method Action setWordBits(Bit#(wordBits) value);
307         wordBits = value;
308     endaction
309
310     method Action setCacheLineBits(Bit#(cacheLineBits) value);
311         cacheLineBits = value;
312     endaction
313
314     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
315         TDiv#(cacheLineBits,8) = value;
316     endaction
317
318     method Action setBypassFIFO();
319         requestOut = FIFO();
320         dataIn = FIFO();
321     endaction
322
323     method Action setBypass();
324         currentCacheLine = Vector();
325         currentWriteEN = Vector();
326         currentWord = 0;
327         requestedWords = 0;
328         dirty = False;
329         justWrite = False;
330         fetchInProgress = False;
331     endaction
332
333     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
334         TDiv#(cacheLineBits,8) = value;
335     endaction
336
337     method Action setWordBits(Bit#(wordBits) value);
338         wordBits = value;
339     endaction
340
341     method Action setCacheLineBits(Bit#(cacheLineBits) value);
342         cacheLineBits = value;
343     endaction
344
345     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
346         TDiv#(cacheLineBits,8) = value;
347     endaction
348
349     method Action setBypassFIFO();
350         requestOut = FIFO();
351         dataIn = FIFO();
352     endaction
353
354     method Action setBypass();
355         currentCacheLine = Vector();
356         currentWriteEN = Vector();
357         currentWord = 0;
358         requestedWords = 0;
359         dirty = False;
360         justWrite = False;
361         fetchInProgress = False;
362     endaction
363
364     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
365         TDiv#(cacheLineBits,8) = value;
366     endaction
367
368     method Action setWordBits(Bit#(wordBits) value);
369         wordBits = value;
370     endaction
371
372     method Action setCacheLineBits(Bit#(cacheLineBits) value);
373         cacheLineBits = value;
374     endaction
375
376     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
377         TDiv#(cacheLineBits,8) = value;
378     endaction
379
380     method Action setBypassFIFO();
381         requestOut = FIFO();
382         dataIn = FIFO();
383     endaction
384
385     method Action setBypass();
386         currentCacheLine = Vector();
387         currentWriteEN = Vector();
388         currentWord = 0;
389         requestedWords = 0;
390         dirty = False;
391         justWrite = False;
392         fetchInProgress = False;
393     endaction
394
395     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
396         TDiv#(cacheLineBits,8) = value;
397     endaction
398
399     method Action setWordBits(Bit#(wordBits) value);
400         wordBits = value;
401     endaction
402
403     method Action setCacheLineBits(Bit#(cacheLineBits) value);
404         cacheLineBits = value;
405     endaction
406
407     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
408         TDiv#(cacheLineBits,8) = value;
409     endaction
410
411     method Action setBypassFIFO();
412         requestOut = FIFO();
413         dataIn = FIFO();
414     endaction
415
416     method Action setBypass();
417         currentCacheLine = Vector();
418         currentWriteEN = Vector();
419         currentWord = 0;
420         requestedWords = 0;
421         dirty = False;
422         justWrite = False;
423         fetchInProgress = False;
424     endaction
425
426     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
427         TDiv#(cacheLineBits,8) = value;
428     endaction
429
430     method Action setWordBits(Bit#(wordBits) value);
431         wordBits = value;
432     endaction
433
434     method Action setCacheLineBits(Bit#(cacheLineBits) value);
435         cacheLineBits = value;
436     endaction
437
438     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
439         TDiv#(cacheLineBits,8) = value;
440     endaction
441
442     method Action setBypassFIFO();
443         requestOut = FIFO();
444         dataIn = FIFO();
445     endaction
446
447     method Action setBypass();
448         currentCacheLine = Vector();
449         currentWriteEN = Vector();
450         currentWord = 0;
451         requestedWords = 0;
452         dirty = False;
453         justWrite = False;
454         fetchInProgress = False;
455     endaction
456
457     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
458         TDiv#(cacheLineBits,8) = value;
459     endaction
460
461     method Action setWordBits(Bit#(wordBits) value);
462         wordBits = value;
463     endaction
464
465     method Action setCacheLineBits(Bit#(cacheLineBits) value);
466         cacheLineBits = value;
467     endaction
468
469     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
470         TDiv#(cacheLineBits,8) = value;
471     endaction
472
473     method Action setBypassFIFO();
474         requestOut = FIFO();
475         dataIn = FIFO();
476     endaction
477
478     method Action setBypass();
479         currentCacheLine = Vector();
480         currentWriteEN = Vector();
481         currentWord = 0;
482         requestedWords = 0;
483         dirty = False;
484         justWrite = False;
485         fetchInProgress = False;
486     endaction
487
488     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
489         TDiv#(cacheLineBits,8) = value;
490     endaction
491
492     method Action setWordBits(Bit#(wordBits) value);
493         wordBits = value;
494     endaction
495
496     method Action setCacheLineBits(Bit#(cacheLineBits) value);
497         cacheLineBits = value;
498     endaction
499
500     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
501         TDiv#(cacheLineBits,8) = value;
502     endaction
503
504     method Action setBypassFIFO();
505         requestOut = FIFO();
506         dataIn = FIFO();
507     endaction
508
509     method Action setBypass();
510         currentCacheLine = Vector();
511         currentWriteEN = Vector();
512         currentWord = 0;
513         requestedWords = 0;
514         dirty = False;
515         justWrite = False;
516         fetchInProgress = False;
517     endaction
518
519     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
520         TDiv#(cacheLineBits,8) = value;
521     endaction
522
523     method Action setWordBits(Bit#(wordBits) value);
524         wordBits = value;
525     endaction
526
527     method Action setCacheLineBits(Bit#(cacheLineBits) value);
528         cacheLineBits = value;
529     endaction
530
531     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
532         TDiv#(cacheLineBits,8) = value;
533     endaction
534
535     method Action setBypassFIFO();
536         requestOut = FIFO();
537         dataIn = FIFO();
538     endaction
539
540     method Action setBypass();
541         currentCacheLine = Vector();
542         currentWriteEN = Vector();
543         currentWord = 0;
544         requestedWords = 0;
545         dirty = False;
546         justWrite = False;
547         fetchInProgress = False;
548     endaction
549
550     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
551         TDiv#(cacheLineBits,8) = value;
552     endaction
553
554     method Action setWordBits(Bit#(wordBits) value);
555         wordBits = value;
556     endaction
557
558     method Action setCacheLineBits(Bit#(cacheLineBits) value);
559         cacheLineBits = value;
560     endaction
561
562     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
563         TDiv#(cacheLineBits,8) = value;
564     endaction
565
566     method Action setBypassFIFO();
567         requestOut = FIFO();
568         dataIn = FIFO();
569     endaction
570
571     method Action setBypass();
572         currentCacheLine = Vector();
573         currentWriteEN = Vector();
574         currentWord = 0;
575         requestedWords = 0;
576         dirty = False;
577         justWrite = False;
578         fetchInProgress = False;
579     endaction
580
581     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
582         TDiv#(cacheLineBits,8) = value;
583     endaction
584
585     method Action setWordBits(Bit#(wordBits) value);
586         wordBits = value;
587     endaction
588
589     method Action setCacheLineBits(Bit#(cacheLineBits) value);
590         cacheLineBits = value;
591     endaction
592
593     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
594         TDiv#(cacheLineBits,8) = value;
595     endaction
596
597     method Action setBypassFIFO();
598         requestOut = FIFO();
599         dataIn = FIFO();
600     endaction
601
602     method Action setBypass();
603         currentCacheLine = Vector();
604         currentWriteEN = Vector();
605         currentWord = 0;
606         requestedWords = 0;
607         dirty = False;
608         justWrite = False;
609         fetchInProgress = False;
610     endaction
611
612     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
613         TDiv#(cacheLineBits,8) = value;
614     endaction
615
616     method Action setWordBits(Bit#(wordBits) value);
617         wordBits = value;
618     endaction
619
620     method Action setCacheLineBits(Bit#(cacheLineBits) value);
621         cacheLineBits = value;
622     endaction
623
624     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
625         TDiv#(cacheLineBits,8) = value;
626     endaction
627
628     method Action setBypassFIFO();
629         requestOut = FIFO();
630         dataIn = FIFO();
631     endaction
632
633     method Action setBypass();
634         currentCacheLine = Vector();
635         currentWriteEN = Vector();
636         currentWord = 0;
637         requestedWords = 0;
638         dirty = False;
639         justWrite = False;
640         fetchInProgress = False;
641     endaction
642
643     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
644         TDiv#(cacheLineBits,8) = value;
645     endaction
646
647     method Action setWordBits(Bit#(wordBits) value);
648         wordBits = value;
649     endaction
650
651     method Action setCacheLineBits(Bit#(cacheLineBits) value);
652         cacheLineBits = value;
653     endaction
654
655     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
656         TDiv#(cacheLineBits,8) = value;
657     endaction
658
659     method Action setBypassFIFO();
660         requestOut = FIFO();
661         dataIn = FIFO();
662     endaction
663
664     method Action setBypass();
665         currentCacheLine = Vector();
666         currentWriteEN = Vector();
667         currentWord = 0;
668         requestedWords = 0;
669         dirty = False;
670         justWrite = False;
671         fetchInProgress = False;
672     endaction
673
674     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
675         TDiv#(cacheLineBits,8) = value;
676     endaction
677
678     method Action setWordBits(Bit#(wordBits) value);
679         wordBits = value;
680     endaction
681
682     method Action setCacheLineBits(Bit#(cacheLineBits) value);
683         cacheLineBits = value;
684     endaction
685
686     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
687         TDiv#(cacheLineBits,8) = value;
688     endaction
689
690     method Action setBypassFIFO();
691         requestOut = FIFO();
692         dataIn = FIFO();
693     endaction
694
695     method Action setBypass();
696         currentCacheLine = Vector();
697         currentWriteEN = Vector();
698         currentWord = 0;
699         requestedWords = 0;
700         dirty = False;
701         justWrite = False;
702         fetchInProgress = False;
703     endaction
704
705     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
706         TDiv#(cacheLineBits,8) = value;
707     endaction
708
709     method Action setWordBits(Bit#(wordBits) value);
710         wordBits = value;
711     endaction
712
713     method Action setCacheLineBits(Bit#(cacheLineBits) value);
714         cacheLineBits = value;
715     endaction
716
717     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
718         TDiv#(cacheLineBits,8) = value;
719     endaction
720
721     method Action setBypassFIFO();
722         requestOut = FIFO();
723         dataIn = FIFO();
724     endaction
725
726     method Action setBypass();
727         currentCacheLine = Vector();
728         currentWriteEN = Vector();
729         currentWord = 0;
730         requestedWords = 0;
731         dirty = False;
732         justWrite = False;
733         fetchInProgress = False;
734     endaction
735
736     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
737         TDiv#(cacheLineBits,8) = value;
738     endaction
739
740     method Action setWordBits(Bit#(wordBits) value);
741         wordBits = value;
742     endaction
743
744     method Action setCacheLineBits(Bit#(cacheLineBits) value);
745         cacheLineBits = value;
746     endaction
747
748     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
749         TDiv#(cacheLineBits,8) = value;
750     endaction
751
752     method Action setBypassFIFO();
753         requestOut = FIFO();
754         dataIn = FIFO();
755     endaction
756
757     method Action setBypass();
758         currentCacheLine = Vector();
759         currentWriteEN = Vector();
760         currentWord = 0;
761         requestedWords = 0;
762         dirty = False;
763         justWrite = False;
764         fetchInProgress = False;
765     endaction
766
767     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
768         TDiv#(cacheLineBits,8) = value;
769     endaction
770
771     method Action setWordBits(Bit#(wordBits) value);
772         wordBits = value;
773     endaction
774
775     method Action setCacheLineBits(Bit#(cacheLineBits) value);
776         cacheLineBits = value;
777     endaction
778
779     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
780         TDiv#(cacheLineBits,8) = value;
781     endaction
782
783     method Action setBypassFIFO();
784         requestOut = FIFO();
785         dataIn = FIFO();
786     endaction
787
788     method Action setBypass();
789         currentCacheLine = Vector();
790         currentWriteEN = Vector();
791         currentWord = 0;
792         requestedWords = 0;
793         dirty = False;
794         justWrite = False;
795         fetchInProgress = False;
796     endaction
797
798     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
799         TDiv#(cacheLineBits,8) = value;
800     endaction
801
802     method Action setWordBits(Bit#(wordBits) value);
803         wordBits = value;
804     endaction
805
806     method Action setCacheLineBits(Bit#(cacheLineBits) value);
807         cacheLineBits = value;
808     endaction
809
810     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
811         TDiv#(cacheLineBits,8) = value;
812     endaction
813
814     method Action setBypassFIFO();
815         requestOut = FIFO();
816         dataIn = FIFO();
817     endaction
818
819     method Action setBypass();
820         currentCacheLine = Vector();
821         currentWriteEN = Vector();
822         currentWord = 0;
823         requestedWords = 0;
824         dirty = False;
825         justWrite = False;
826         fetchInProgress = False;
827     endaction
828
829     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
830         TDiv#(cacheLineBits,8) = value;
831     endaction
832
833     method Action setWordBits(Bit#(wordBits) value);
834         wordBits = value;
835     endaction
836
837     method Action setCacheLineBits(Bit#(cacheLineBits) value);
838         cacheLineBits = value;
839     endaction
840
841     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
842         TDiv#(cacheLineBits,8) = value;
843     endaction
844
845     method Action setBypassFIFO();
846         requestOut = FIFO();
847         dataIn = FIFO();
848     endaction
849
850     method Action setBypass();
851         currentCacheLine = Vector();
852         currentWriteEN = Vector();
853         currentWord = 0;
854         requestedWords = 0;
855         dirty = False;
856         justWrite = False;
857         fetchInProgress = False;
858     endaction
859
860     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
861         TDiv#(cacheLineBits,8) = value;
862     endaction
863
864     method Action setWordBits(Bit#(wordBits) value);
865         wordBits = value;
866     endaction
867
868     method Action setCacheLineBits(Bit#(cacheLineBits) value);
869         cacheLineBits = value;
870     endaction
871
872     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
873         TDiv#(cacheLineBits,8) = value;
874     endaction
875
876     method Action setBypassFIFO();
877         requestOut = FIFO();
878         dataIn = FIFO();
879     endaction
880
881     method Action setBypass();
882         currentCacheLine = Vector();
883         currentWriteEN = Vector();
884         currentWord = 0;
885         requestedWords = 0;
886         dirty = False;
887         justWrite = False;
888         fetchInProgress = False;
889     endaction
890
891     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
892         TDiv#(cacheLineBits,8) = value;
893     endaction
894
895     method Action setWordBits(Bit#(wordBits) value);
896         wordBits = value;
897     endaction
898
899     method Action setCacheLineBits(Bit#(cacheLineBits) value);
900         cacheLineBits = value;
901     endaction
902
903     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
904         TDiv#(cacheLineBits,8) = value;
905     endaction
906
907     method Action setBypassFIFO();
908         requestOut = FIFO();
909         dataIn = FIFO();
910     endaction
911
912     method Action setBypass();
913         currentCacheLine = Vector();
914         currentWriteEN = Vector();
915         currentWord = 0;
916         requestedWords = 0;
917         dirty = False;
918         justWrite = False;
919         fetchInProgress = False;
920     endaction
921
922     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
923         TDiv#(cacheLineBits,8) = value;
924     endaction
925
926     method Action setWordBits(Bit#(wordBits) value);
927         wordBits = value;
928     endaction
929
930     method Action setCacheLineBits(Bit#(cacheLineBits) value);
931         cacheLineBits = value;
932     endaction
933
934     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
935         TDiv#(cacheLineBits,8) = value;
936     endaction
937
938     method Action setBypassFIFO();
939         requestOut = FIFO();
940         dataIn = FIFO();
941     endaction
942
943     method Action setBypass();
944         currentCacheLine = Vector();
945         currentWriteEN = Vector();
946         currentWord = 0;
947         requestedWords = 0;
948         dirty = False;
949         justWrite = False;
950         fetchInProgress = False;
951     endaction
952
953     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
954         TDiv#(cacheLineBits,8) = value;
955     endaction
956
957     method Action setWordBits(Bit#(wordBits) value);
958         wordBits = value;
959     endaction
960
961     method Action setCacheLineBits(Bit#(cacheLineBits) value);
962         cacheLineBits = value;
963     endaction
964
965     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
966         TDiv#(cacheLineBits,8) = value;
967     endaction
968
969     method Action setBypassFIFO();
970         requestOut = FIFO();
971         dataIn = FIFO();
972     endaction
973
974     method Action setBypass();
975         currentCacheLine = Vector();
976         currentWriteEN = Vector();
977         currentWord = 0;
978         requestedWords = 0;
979         dirty = False;
980         justWrite = False;
981         fetchInProgress = False;
982     endaction
983
984     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
985         TDiv#(cacheLineBits,8) = value;
986     endaction
987
988     method Action setWordBits(Bit#(wordBits) value);
989         wordBits = value;
990     endaction
991
992     method Action setCacheLineBits(Bit#(cacheLineBits) value);
993         cacheLineBits = value;
994     endaction
995
996     method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
997         TDiv#(cacheLineBits,8) = value;
998     endaction
999
1000    method Action setBypassFIFO();
1001        requestOut = FIFO();
1002        dataIn = FIFO();
1003    endaction
1004
1005    method Action setBypass();
1006        currentCacheLine = Vector();
1007        currentWriteEN = Vector();
1008        currentWord = 0;
1009        requestedWords = 0;
1010        dirty = False;
1011        justWrite = False;
1012        fetchInProgress = False;
1013    endaction
1014
1015    method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
1016        TDiv#(cacheLineBits,8) = value;
1017    endaction
1018
1019    method Action setWordBits(Bit#(wordBits) value);
1020        wordBits = value;
1021    endaction
1022
1023    method Action setCacheLineBits(Bit#(cacheLineBits) value);
1024        cacheLineBits = value;
1025    endaction
1026
1027    method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
1028        TDiv#(cacheLineBits,8) = value;
1029    endaction
1030
1031    method Action setBypassFIFO();
1032        requestOut = FIFO();
1033        dataIn = FIFO();
1034    endaction
1035
1036    method Action setBypass();
1037        currentCacheLine = Vector();
1038        currentWriteEN = Vector();
1039        currentWord = 0;
1040        requestedWords = 0;
1041        dirty = False;
1042        justWrite = False;
1043        fetchInProgress = False;
1044    endaction
1045
1046    method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
1047        TDiv#(cacheLineBits,8) = value;
1048    endaction
1049
1050    method Action setWordBits(Bit#(wordBits) value);
1051        wordBits = value;
1052    endaction
1053
1054    method Action setCacheLineBits(Bit#(cacheLineBits) value);
1055        cacheLineBits = value;
1056    endaction
1057
1058    method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
1059        TDiv#(cacheLineBits,8) = value;
1060    endaction
1061
1062    method Action setBypassFIFO();
1063        requestOut = FIFO();
1064        dataIn = FIFO();
1065    endaction
1066
1067    method Action setBypass();
1068        currentCacheLine = Vector();
1069        currentWriteEN = Vector();
1070        currentWord = 0;
1071        requestedWords = 0;
1072        dirty = False;
1073        justWrite = False;
1074        fetchInProgress = False;
1075    endaction
1076
1077    method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
1078        TDiv#(cacheLineBits,8) = value;
1079    endaction
1080
1081    method Action setWordBits(Bit#(wordBits) value);
1082        wordBits = value;
1083    endaction
1084
1085    method Action setCacheLineBits(Bit#(cacheLineBits) value);
1086        cacheLineBits = value;
1087    endaction
1088
1089    method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
1090        TDiv#(cacheLineBits,8) = value;
1091    endaction
1092
1093    method Action setBypassFIFO();
1094        requestOut = FIFO();
1095        dataIn = FIFO();
1096    endaction
1097
1098    method Action setBypass();
1099        currentCacheLine = Vector();
1100        currentWriteEN = Vector();
1101        currentWord = 0;
1102        requestedWords = 0;
1103        dirty = False;
1104        justWrite = False;
1105        fetchInProgress = False;
1106    endaction
1107
1108    method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
1109        TDiv#(cacheLineBits,8) = value;
1110    endaction
1111
1112    method Action setWordBits(Bit#(wordBits) value);
1113        wordBits = value;
1114    endaction
1115
1116    method Action setCacheLineBits(Bit#(cacheLineBits) value);
1117        cacheLineBits = value;
1118    endaction
1119
1120    method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
1121        TDiv#(cacheLineBits,8) = value;
1122    endaction
1123
1124    method Action setBypassFIFO();
1125        requestOut = FIFO();
1126        dataIn = FIFO();
1127    endaction
1128
1129    method Action setBypass();
1130        currentCacheLine = Vector();
1131        currentWriteEN = Vector();
1132        currentWord = 0;
1133        requestedWords = 0;
1134        dirty = False;
1135        justWrite = False;
1136        fetchInProgress = False;
1137    endaction
1138
1139    method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
1140        TDiv#(cacheLineBits,8) = value;
1141    endaction
1142
1143    method Action setWordBits(Bit#(wordBits) value);
1144        wordBits = value;
1145    endaction
1146
1147    method Action setCacheLineBits(Bit#(cacheLineBits) value);
1148        cacheLineBits = value;
1149    endaction
1150
1151    method Action setDiv(Bit#(TDiv#(cacheLineBits,8)) value);
1152        TDiv#(cacheLineBits,8) = value;
1153    endaction
1154
1155    method Action setBypassFIFO();
1156        requestOut = FIFO();
1157        dataIn = FIFO();
1158    endaction
1159
1160    method Action setBypass();
1161        currentCacheLine = Vector();
1162        currentWriteEN = Vector();
1163        currentWord = 0;
1164        requestedWords = 0;
1165        dirty = False;
1166        justWrite = False;
1167        fetchInProgress = False;
1168    endaction
1169
1170    method Action setDiv(Bit#(TDiv#(cacheLine
```

## Übung zur Vorlesung Einführung in Computer Microsystems

```
159     currentWord <= currentWord + 1;
160     let result <- toGet(dataIn).get();
161     currentCacheLine[currentWord] <= result;
162 endrule
163
164 rule switchOperation (fetchInProgress && dirty
165                         && currentWord >= fromInteger(valueOf(wordsPerCacheLine)));
166     dirty          <= False;
167     currentWord    <= 0;
168     if(justWrite) begin
169         fetchInProgress <= False;
170     end
171 endrule
172
173 rule operationDone (fetchInProgress && !dirty
174                         && currentWord >= fromInteger(valueOf(wordsPerCacheLine)));
175     fetchInProgress <= False;
176 endrule
177
178 interface Server toUser;
179     interface Put request;
180         method Action put(FetchRequest#(cacheLineBits) req) if(!fetchInProgress);
181             fetchInProgress <= True;
182             currentWord <= 0;
183             requestedWords <= 0;
184             case(req) matches
185                 tagged FetchDirty .v: begin
186                     dirty      <= True;
187                     justWrite   <= False;
188                     readAddress <= tpl_1(v);
189                     writeAddress <= tpl_2(v);
190                     writeVReg(currentCacheLine, unpack(tpl_3(v)));
191                 end
192                 tagged JustWrite .v: begin
193                     dirty      <= True;
194                     justWrite   <= True;
195                     writeAddress <= tpl_1(v);
196                     writeVReg(currentCacheLine, unpack(tpl_2(v)));
197                     writeVReg(currentWriteEN,   unpack(tpl_3(v)));
198                 end
199                 tagged Fetch .v: begin
200                     dirty      <= False;
201                     justWrite   <= False;
202                 end
203             endcase
204         endmethod
205     endinterface
206
207     interface Get response;
208         method ActionValue#(Bit#(cacheLineBits)) get() if(!fetchInProgress);
209             return pack(readVReg(currentCacheLine));
210         endmethod
211     endinterface
212 endinterface
213
```

## Übung zur Vorlesung Einführung in Computer Microsystems

```
214     interface Client toRAM;
215         interface Get request = toGet(requestOut);
216         interface Put response = toPut(dataIn);
217     endinterface
218 endmodule
219
220 endpackage
```

### Aufgabe 4.1.3 Direct-Mapped-Cache

Implementieren Sie den oben beschriebenen Cache mit 128 Slots.

Zusatzaufgabe: Implementieren Sie den Cache generisch für eine beliebige Anzahl Slots.

```
1 package Cache;
2
3     import RAM      :: *;
4     import CacheLine :: *;
5     import GetPut   :: *;
6     import ClientServer :: *;
7     import Connectable :: *;
8     import FIFO     :: *;
9     import SpecialFIFOs :: *;
10    import Vector   :: *;
11
12    typedef struct {
13        Reg#(Bool) dirty;
14        Reg#(Bool) valid;
15        Reg#(Bit#(tagAddrBits)) tag;
16        Reg#(Bit#(cacheLineBits)) cacheLine;
17    } CacheLine#(numeric type tagAddrBits, numeric type cacheLineBits);
18
19    interface Cache#(numeric type wordBits, numeric type cacheLineBits, numeric type slots);
20        interface RAMClient#(wordBits, cacheLineBits) toRAM;
21        interface RAMServer#(wordBits, cacheLineBits) toUser;
22        method Action clearCache();
23    endinterface
24
25    module mkCache(Cache#(wordBits, cacheLineBits, slots))
26        provisos(
27            Mul#(cacheLineBytes, 8, cacheLineBits),
28            Div#(cacheLineBits, 8, cacheLineBytes),
29            Log#(cacheLineBytes, cacheLineAddrBits),
30            Div#(cacheLineBits, cacheLineBytes, 8),
31            Add#(cacheLineAddrBits, addrBitsWithoutCacheLine, 32),
32            Add#(16, a__, addrBitsWithoutCacheLine),
33            Div#(wordBits, 8, wordBytes),
34            Div#(wordBits, wordBytes, 8),
35            Add#(wordAddrBits, b__, 32),
36            Add#(16, c__, b__),
37            Log#(wordBytes, wordAddrBits),
38            Div#(cacheLineBits, wordBits, wordsPerCacheLine),
39            Mul#(wordsPerCacheLine, wordBits, cacheLineBits),
40            Log#(slots, slotsAddrBits),
41            Add#(slotsAddrBits, cacheLineAddrBits, addrBitsNoTag),
42            Add#(addrBitsNoTag, tagAddrBits, 32),
43            Div#(cacheLineBits, cacheLineBits, 1)
```

## Übung zur Vorlesung Einführung in Computer Microsystems

```
44
45     );
46 FIFO#(RAMRequest#(wordBits, cacheLineBits)) requestIn <- mkFIFO();
47 FIFO#(Bit#(cacheLineBits))           dataOut   <- mkFIFO();
48
49 FIFO#(RAMRequest#(wordBits, cacheLineBits)) requestOut <- mkFIFO();
50 FIFO#(Bit#(cacheLineBits))           dataIn    <- mkFIFO();
51
52 FetchCacheLineWhole#(wordBits, cacheLineBits) fetcher <- mkFetchCacheLineWhole();
53 mkConnection(fetcher.toRAM, toGPServer(requestOut, dataIn));
54
55 Reg#(Bool)                      actionInProgress <- mkReg(False);
56
57 Vector#(slots, CacheLine#(tagAddrBits, cacheLineBits)) cacheLines;
58 for(Integer i = 0; i < valueOf(slots); i = i + 1) begin
59     cacheLines[i].dirty <- mkReg(False);
60     cacheLines[i].valid <- mkReg(False);
61     cacheLines[i].cacheLine <- mkRegU;
62     cacheLines[i].tag <- mkRegU;
63 end
64
65 Reg#(Bool) read      <- mkReg(False);
66 Reg#(Bool) fetchActive <- mkReg(False);
67
68 Reg#(Bit#(slotsAddrBits)) expectedSlot      <- mkReg(0);
69 Reg#(Bit#(tagAddrBits))   expectedTag        <- mkReg(0);
70 Reg#(Bit#(cacheLineBits)) currentWriteCacheLine <- mkReg(0);
71 Reg#(Bit#(cacheLineBytes)) currentWriteEN      <- mkReg(0);
72 Reg#(Bool)           isMiss          <- mkReg(False);
73 Reg#(UInt#(12))        cycleCounter      <- mkRegU();
74 Reg#(UInt#(20))        requestsSinceLastMiss <- mkReg(0);
75
76 Reg#(UInt#(32))        clockCounter      <- mkReg(0);
77 Reg#(UInt#(32))        requestStart      <- mkReg(0);
78
79 function Bit#(tagAddrBits) getTag(Bit#(32) addr);
80     return addr[31:valueOf(addrBitsNoTag)];
81 endfunction
82
83 function Bit#(slotsAddrBits) getSlot(Bit#(32) addr);
84     return addr[valueOf(addrBitsNoTag) - 1:valueOf(cacheLineAddrBits)];
85 endfunction
86
87 function Bit#(32) toAddr(Bit#(tagAddrBits) tag, Bit#(slotsAddrBits) slot);
88     return {tag,slot,0};
89 endfunction
90
91 (* mutually_exclusive="fetchCorrectLine,updateCycleCounter" *)
92 (* descending_urgency="writeResponse,updateCycleCounter" *)
93 rule fetchCorrectLine(actionInProgress
94             && (expectedTag != cacheLines[expectedSlot].tag
95                  || !cacheLines[expectedSlot].valid)
96                  && !fetchActive);
97     $display("(%0d) Read: %b Cache Miss for %x != %x @ %x IsDirty: %d. %d requests without miss.", 
98             $time, read, expectedTag, cacheLines[expectedSlot].tag, expectedSlot,
99             cacheLines[expectedSlot].dirty, requestsSinceLastMiss);
```

## Übung zur Vorlesung Einführung in Computer Microsystems

```
99         isMiss <= True;
100        requestsSinceLastMiss <= 0;
101        cycleCounter <= 0;
102        FetchRequest#(cacheLineBits) req = tagged Fetch
103                      toAddr(cacheLines[expectedSlot].tag, expectedSlot);
104        if(cacheLines[expectedSlot].dirty) begin
105            req = tagged FetchDirty tuple3(toAddr(expectedTag, expectedSlot),
106                                         toAddr(cacheLines[expectedSlot].tag, expectedSlot),
107                                         cacheLines[expectedSlot].cacheLine);
108        end
109        cacheLines[expectedSlot].valid           <= False;
110        cacheLines[expectedSlot].dirty          <= False;
111        cacheLines[expectedSlot].tag            <= expectedTag;
112
113        fetchActive      <= True;
114
115        fetcher.toUser.request.put(req);
116    endrule
117
118    rule updateCycleCounter(isMiss);
119        cycleCounter <= cycleCounter + 1;
120    endrule
121
122    rule updateClockCounter;
123        clockCounter <= clockCounter + 1;
124    endrule
125
126    rule handleFetcherResponse (actionInProgress && fetchActive);
127        let fetchedLine <- fetcher.toUser.response.get();
128        cacheLines[expectedSlot].valid <= True;
129        cacheLines[expectedSlot].cacheLine <= fetchedLine;
130
131        fetchActive      <= False;
132    endrule
133
134    rule writeResponse(!fetchActive && cacheLines[expectedSlot].tag == expectedTag
135                           && cacheLines[expectedSlot].valid
136                           && actionInProgress);
137        if(isMiss) $display("(%0d) Miss took %d cycles.", $time, cycleCounter);
138        $display("(%0d) Request handling took %d cycles.", $time, clockCounter - requestStart);
139        if(read) begin
140            dataOut.enq(cacheLines[expectedSlot].cacheLine);
141        end else begin
142            Bit#(cacheLineBits) writeENBits = 0;
143            for(Integer i = 0; i < valueOf(cacheLineBytes); i = i + 1) begin
144                for(Integer j = 0; j < 8; j = j + 1) begin
145                    writeENBits[(i * 8) + j] = currentWriteEN[i];
146                end
147            end
148            cacheLines[expectedSlot].cacheLine <= (cacheLines[expectedSlot].cacheLine & ~writeENBits)
149                                         | (currentWriteCacheLine & writeENBits);
150            cacheLines[expectedSlot].dirty <= True;
151        end
152        requestsSinceLastMiss <= requestsSinceLastMiss + 1;
153        actionInProgress <= False;
```

## Übung zur Vorlesung Einführung in Computer Microsystems

```
154     isMiss <= False;
155 endrule
156
157
158     rule handleRead (requestIn.first() matches tagged Read .v &&& !actionInProgress);
159         requestIn.deq();
160
161         expectedTag          <= getTag(v);
162         expectedSlot         <= getSlot(v);
163         read                 <= True;
164         actionInProgress     <= True;
165
166         requestStart <= clockCounter;
167 endrule
168
169     rule handleWriteRequest (requestIn.first() matches tagged Write .v &&& !actionInProgress);
170         requestIn.deq();
171
172         let address = tpl_1(v);
173         let data   = tpl_2(v);
174         let writeen = tpl_3(v);
175
176         expectedTag          <= getTag(address);
177         expectedSlot         <= getSlot(address);
178         currentWriteCacheLine <= unpack(data);
179         currentWriteEN        <= unpack(writeen);
180         read                 <= False;
181         actionInProgress     <= True;
182
183         requestStart <= clockCounter;
184 endrule
185
186     method Action clearCache() if(!actionInProgress);
187         for(Integer i = 0; i < valueOf(slots); i = i + 1) begin
188             cacheLines[i].dirty <= False;
189             cacheLines[i].valid <= False;
190         end
191     endmethod
192
193     interface Client toRAM;
194         interface Get request  = toGet(requestOut);
195         interface Put response = toPut(dataIn);
196     endinterface
197     interface Server toUser;
198         interface Put request  = toPut(requestIn);
199         interface Get response = toGet(dataOut);
200     endinterface
201 endmodule
202
203 interface FetchCacheLineWhole#(numeric type wordBits, numeric type cacheLineBits);
204     interface RAMClient#(wordBits, cacheLineBits) toRAM;
205     interface Server#(FetchRequest#(cacheLineBits), Bit#(cacheLineBits)) toUser;
206 endinterface
207
208 module mkFetchCacheLineWhole(FetchCacheLineWhole#(wordBits, cacheLineBits))
```

## Übung zur Vorlesung Einführung in Computer Microsystems

```
209     provisos(
210         Div#(cacheLineBits, wordBits, wordsPerCacheLine),
211         Div#(wordBits, 8, wordBytes),
212         Div#(cacheLineBits, 8, cacheLineBytes)
213     );
214     FIFO#(RAMRequest#(wordBits, cacheLineBits)) requestOut <- mkBypassFIFO();
215     FIFO#(Bit#(cacheLineBits)) dataIn <- mkBypassFIFO();
216
217     Reg#(Bool) dirty <- mkReg(False);
218     Reg#(Bool) fetchInProgress <- mkReg(False);
219     Reg#(Bit#(cacheLineBits)) currentCacheLine <- mkRegU();
220     Reg#(Bit#(32)) readAddress <- mkReg(0);
221     Reg#(Bit#(32)) writeAddress <- mkReg(0);
222     Reg#(Bool) requestDone <- mkReg(False);
223
224     rule operation (fetchInProgress && !requestDone);
225         RAMRequest#(wordBits, cacheLineBits) req = tagged Read 0;
226         if(dirty) begin
227             Bit#(cacheLineBytes) writeEN = (1 << valueOf(cacheLineBytes)) - 1;
228             req = tagged Write tuple3(writeAddress, currentCacheLine, writeEN);
229         end else begin
230             req = tagged Read readAddress;
231         end
232         requestDone <= True;
233         requestOut.enq(req);
234     endrule
235
236     rule operationRead (fetchInProgress && !dirty && requestDone);
237         let result <- toGet(dataIn).get();
238         currentCacheLine <= result;
239         fetchInProgress <= False;
240     endrule
241
242     rule switchOperation (fetchInProgress && dirty && requestDone);
243         dirty <= False;
244         requestDone <= False;
245     endrule
246
247     interface Server toUser;
248         interface Put request;
249             method Action put(FetchRequest#(cacheLineBits) req) if(!fetchInProgress);
250                 case(req) matches
251                     tagged FetchDirty .v: begin
252                         dirty <= True;
253                         readAddress <= tpl_1(v);
254                         writeAddress <= tpl_2(v);
255                         currentCacheLine <= tpl_3(v);
256                         fetchInProgress <= True;
257                         requestDone <= False;
258                     end
259                     tagged Fetch .v: begin
260                         dirty <= False;
261                         readAddress <= v;
262                         fetchInProgress <= True;
263                         requestDone <= False;
```

## Übung zur Vorlesung Einführung in Computer Microsystems

```
264           end
265           tagged JustWrite .v: begin
266             $display("Not handled!");
267             fetchInProgress <= False;
268             requestDone     <= False;
269           end
270         endcase
271       endmethod
272     endinterface
273
274   interface Get response;
275     method ActionValue#(Bit#(cacheLineBits)) get() if(!fetchInProgress);
276       return currentCacheLine;
277     endmethod
278   endinterface
279 endinterface
280
281 interface Client toRAM;
282   interface Get request = toGet(requestOut);
283   interface Put response = toPut(dataIn);
284 endinterface
285 endmodule
286
287 endpackage
```

### Aufgabe 4.1.4 Testen

Erstellen Sie eine Testbench, die die obigen Module nutzt und zusammenschaltet. Testen Sie verschiedene Speicherzugriffsmuster mit und ohne Cache. Mögliche Muster sind zum Beispiel:

- a) Aufsteigendes Lesen und Schreiben für verschiedene Anzahl von Wörtern
- b) Aufsteigendes Lesen und Schreiben mehrmals auf die selben Wörter
- c) Lesen und Schreiben aller  $n$  Wörter für  $n = 4, 8, 16, 32$
- d) Zufälliger Zugriff auf Speicherelemente
- e) Spalten- oder zeilenweise Matrixmultiplikation

Vergleichen Sie die Zugriffszeit pro Element für die verschiedenen Zugriffsmuster. Was stellen Sie fest, wenn Sie die selben Zugriffsmuster auf den Poolrechnern implementieren und die Laufzeiten vergleichen?

Zusatzaufgabe: Testen Sie die obigen Speichermuster mit verschiedener Cache-Line Größe und Anzahl von Slots.

```
1 package Testbench;
2
3   import GetPut :: *;
4   import ClientServer :: *;
5   import Connectable :: *;
6   import Vector :: *;
7   import StmtFSM :: *;
8   import FIFO :: *;
9
10  import RAM :: *;
11  import Cache :: *;
12  import CacheLine :: *;
13  import CacheLineToWord :: *;
```

## Übung zur Vorlesung Einführung in Computer Microsystems

```
14
15     typedef 512 CacheLineBits;
16     typedef 32 WordBits;
17     typedef 128 Slots;
18     typedef 512 SlotsL2;
19
20     typedef TDiv#(WordBits, 8) WordBytes;
21
22     (* synthesize *)
23     module mkTestbench(Empty);
24
25         CacheLineRAM#(WordBits, CacheLineBits)      ram           <- mkCacheLineRAM();
26         CacheLineToWord#(WordBits, CacheLineBits) wordConverter <- mkCacheLineToWord();
27
28         //Cache#(WordBits, CacheLineBits, Slots)      cache <- mkCache();
29         //Cache#(WordBits, CacheLineBits, SlotsL2) l2cache <- mkCache();
30
31         // Without Cache
32         mkConnection(wordConverter.toRAM, ram.toUser);
33
34         // With Cache
35         //mkConnection(cache.toRAM, ram.toUser);
36         //mkConnection(wordConverter.toRAM, cache.toUser);
37
38         // With L2 Cache
39         //mkConnection(l2cache.toRAM, ram.toUser);
40         //mkConnection(cache.toRAM, l2cache.toUser);
41         //mkConnection(wordConverter.toRAM, cache.toUser);
42
43         Reg#(UInt#(32)) clockCounter <- mkReg(0);
44         Reg#(UInt#(32)) startTime      <- mkRegU;
45
46         rule count;
47             clockCounter <= clockCounter + 1;
48         endrule
49
50         Vector#(5, UInt#(30)) testPoints;
51         testPoints[0] = 128;
52         testPoints[1] = 256;
53         testPoints[2] = 512;
54         testPoints[3] = 1024;
55         testPoints[4] = 16384;
56
57         Vector#(5, UInt#(30)) skipPoints;
58         skipPoints[0] = 1;
59         skipPoints[1] = 8;
60         skipPoints[2] = 16;
61         skipPoints[3] = 32;
62         skipPoints[4] = 128;
63
64         Reg#(UInt#(32)) testCounter <- mkRegU;
65         Reg#(UInt#(32)) skipCounter <- mkRegU;
66
67         Reg#(UInt#(30)) loopCounter <- mkReg(0);
68         Stmt s = (
```

## Übung zur Vorlesung Einführung in Computer Microsystems

```
69      seq
70          for(skipCounter <= 0; skipCounter < 5; skipCounter <= skipCounter + 1) seq
71              $display("Write test %d", skipCounter);
72              for(testCounter <= 0; testCounter < 5; testCounter <= testCounter + 1) seq
73                  12cache.clearCache();
74                  cache.clearCache();
75                  startTime <= clockCounter;
76                  for(loopCounter <= 0; loopCounter < testPoints[testCounter];
77                      loopCounter <= loopCounter + skipPoints[skipCounter]) seq
78                      action
79                          Bit#(TDIV#(WordBits, 8)) byteEnable = (1 << valueOf(WordBytes)) - 1;
80                          RAMRequest#(WordBits, WordBits) req = tagged Write
81                                          tuple3({pack(loopCounter), 2'b0},
82                                              extend(pack(loopCounter)),
83                                              byteEnable);
84                          wordConverter.toUser.request.put(req);
85                      endaction
86                  endseq
87                  $display("%d: %d", testCounter, clockCounter - startTime);
88                  delay(128);
89              endseq
90              $display("Read test %d", skipCounter);
91              for(testCounter <= 0; testCounter < 5; testCounter <= testCounter + 1) seq
92                  12cache.clearCache();
93                  cache.clearCache();
94                  startTime <= clockCounter;
95                  for(loopCounter <= 0; loopCounter < testPoints[testCounter];
96                      loopCounter <= loopCounter + skipPoints[skipCounter]) seq
97                      action
98                          Bit#(32) addr = {pack(loopCounter), 2'b0};
99                          RAMRequest#(WordBits, WordBits) req = tagged Read addr;
100                         wordConverter.toUser.request.put(req);
101                     endaction
102                     action
103                         let result <- wordConverter.toUser.response.get();
104                     endaction
105                 endseq
106                 $display("%d: %d", testCounter, clockCounter - startTime);
107                 delay(128);
108             endseq
109             $display("Read + Write test %d", skipCounter);
110             for(testCounter <= 0; testCounter < 5; testCounter <= testCounter + 1) seq
111                 12cache.clearCache();
112                 cache.clearCache();
113                 startTime <= clockCounter;
114                 for(loopCounter <= 0; loopCounter < testPoints[testCounter];
115                     loopCounter <= loopCounter + skipPoints[skipCounter]) seq
116                     action
117                         Bit#(TDIV#(WordBits, 8)) byteEnable = (1 << valueOf(WordBytes)) - 1;
118                         RAMRequest#(WordBits, WordBits) req = tagged Write
119                                         tuple3({pack(loopCounter), 2'b0},
120                                             extend(pack(loopCounter)),
121                                             byteEnable);
122                         wordConverter.toUser.request.put(req);
123                     endaction
```

## Übung zur Vorlesung Einführung in Computer Microsystems

```
124          endseq
125      for(loopCounter <= 0; loopCounter < testPoints[testCounter];
126          loopCounter <= loopCounter + skipPoints[skipCounter]) seq
127          action
128              Bit#(32) addr = {pack(loopCounter), 2'b0};
129              RAMRequest #(WordBits, WordBits) req = tagged Read addr;
130              wordConverter.toUser.request.put(req);
131          endaction
132          action
133              let result <- wordConverter.toUser.response.get();
134          endaction
135          endseq
136          $display("%d: %d", testCounter, clockCounter - startTime);
137          delay(128);
138      endseq
139      endseq
140  );
141
142      mkAutoFSM(s);
143  endmodule
144
145 endpackage
```