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 Speicheraddresse des übergeordneten Speichers kann an genau einer Position des Caches abgelegt werden.

Zusätzlich speichert der Cache immer eine Cache-Line die aus mehreren Worten 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 4 B pro Wort ergeben sich somit 64 B 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 Speicheraddresse 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.

```
1 package RAM;
2
```

```
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);
```

```
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 512 bit 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;
```

```
33              interface Get request;
34                  method ActionValue#(RAMRequest#(wordBits, cacheLineBits)) get();
35                      let reqFromUser <- toGet(requestToRam).get();
36                      RAMRequest#(wordBits, cacheLineBits) reqToRam = tagged Read 0;
37
38                      case(reqFromUser) matches
39                          tagged Write .v: begin
40                              let reqAddress = tpl_1(v);
41                              let reqData    = tpl_2(v);
42                              let reqWriteEN = tpl_3(v);
43
44                              Bit#(cacheLineAddrBits)  firstByteOfWord  = truncate(reqAddress);
45                              Bit#(cacheLineBytes)              writeen = 0;
46                              for(Integer i = 0; i < 4; i = i + 1) begin
47                                  writeen[firstByteOfWord + fromInteger(i)] = reqWriteEN[i];
48                              end
49
50                              Bit#(cacheLineAddrBits)                     selWordT = truncate(reqAddress);
51                              Bit#(TSub#(cacheLineAddrBits, wordAddrBits)) selWord = truncateLSB(selWordT);
52                              Vector#(wordsPerCacheLine, Bit#(wordBits)) cacheLine = unpack(0);
53                              let cacheLineUpdated = update(cacheLine, selWord, reqData);
54                              reqToRam = tagged Write tuple3(reqAddress, pack(cacheLineUpdated), writeen);
55                          end
56                          tagged Read  .v: begin
57                              reqToRam = tagged Read v;
58                              Bit#(cacheLineAddrBits) selWordT  = truncate(v);
59                                                  selectedWord <= truncateLSB(selWordT);
60                          end
61                      endcase
62                      return reqToRam;
63                  endmethod
64              endinterface
65
66              interface Put response;
67                  method Action put(Bit#(cacheLineBits) cacheLine);
68                      Vector#(wordsPerCacheLine, Bit#(wordBits)) cacheLineVec = unpack(cacheLine);
69                      responseToUser.enq(cacheLineVec[selectedWord]);
70                  endmethod
71              endinterface
72          endinterface
73
74          interface Server toUser;
75              interface Put request;
76                  method Action put(RAMRequest#(wordBits, wordBits) req) if(!transferActive[1]);
77                      if(req matches tagged Read .v) transferActive[1] <= True;
78                      requestToRam.enq(req);
79                  endmethod
80              endinterface
81
82              interface Get response;
83                  method ActionValue#(Bit#(wordBits)) get();
84                      transferActive[0] <= False;
85                      let ret <- toGet(responseToUser).get();
86                      return ret;
87                  endmethod
```

```
            endinterface
        endinterface
    endmodule

endpackage
```

Word → Cache-Line

```
package CacheLine;

    import RAM ::*;
    import GetPut :: *;
    import ClientServer :: *;
    import Connectable :: *;
    import FIFO :: *;
    import SpecialFIFOs :: *;
    import Vector :: *;

    interface CacheLineRAM#(numeric type wordBits, numeric type cacheLineBits);
        interface RAMServer#(wordBits, cacheLineBits) toUser;
    endinterface

    module mkCacheLineRAM(CacheLineRAM#(wordBits, cacheLineBits))
    provisos(
            Mul#(cacheLineBytes, 8, cacheLineBits),
            Div#(cacheLineBits, 8, cacheLineBytes),
            Log#(cacheLineBytes, cacheLineAddrBits),
            Div#(cacheLineBits, cacheLineBytes, 8),
            Add#(cacheLineAddrBits, addrBitsWithoutCacheLine, 32),
            Add#(18, a__, addrBitsWithoutCacheLine),
            Div#(wordBits, 8 , wordBytes),
            Div#(wordBits, wordBytes, 8),
            Add#(wordAddrBits, b__, 32),
            Add#(18, c__, b__),
            Log#(wordBytes, wordAddrBits),
            Div#(cacheLineBits, wordBits, wordsPerCacheLine),
            Mul#(wordsPerCacheLine, wordBits, cacheLineBits)
            );
        FIFO#(RAMRequest#(wordBits, cacheLineBits)) requestIn <- mkFIFO();
        FIFO#(Bit#(cacheLineBits))                  dataOut  <- mkFIFO();

        RAMServer#(wordBits, wordBits) ram <- mkRAM();
        FetchCacheLine#(wordBits, cacheLineBits) fetcher <- mkFetchCacheLine();
        mkConnection(ram, fetcher.toRAM);

        Reg#(Bool)                        actionInProgress <- mkReg(False);
        Reg#(Bool)                                    read <- mkReg(False);
        Reg#(Bool)                             fetchActive <- mkReg(False);

        Reg#(Bit#(32))                  expectedCacheLineAddr <- mkReg(0);
        Reg#(Bit#(cacheLineBits))       currentWriteCacheLine <- mkReg(0);
        Reg#(Bit#(cacheLineBytes))             currentWriteEN <- mkReg(0);

        Reg#(UInt#(32))                           requestTimer <- mkReg(0);

        rule countRequest (fetchActive);
```

```
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
```

```
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             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
```

```
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 beschrieben 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)
```

```
        );
    FIFO#(RAMRequest#(wordBits, cacheLineBits)) requestIn <- mkFIFO();
    FIFO#(Bit#(cacheLineBits))                  dataOut   <- mkFIFO();

    FIFO#(RAMRequest#(wordBits, cacheLineBits)) requestOut <- mkFIFO();
    FIFO#(Bit#(cacheLineBits))                      dataIn     <- mkFIFO();

    FetchCacheLineWhole#(wordBits, cacheLineBits) fetcher <- mkFetchCacheLineWhole();
    mkConnection(fetcher.toRAM, toGPServer(requestOut, dataIn));

    Reg#(Bool)                        actionInProgress <- mkReg(False);

    Vector#(slots, CacheLine#(tagAddrBits, cacheLineBits)) cacheLines;
    for(Integer i = 0; i < valueOf(slots); i = i + 1) begin
        cacheLines[i].dirty <- mkReg(False);
        cacheLines[i].valid <- mkReg(False);
        cacheLines[i].cacheLine <- mkRegU;
        cacheLines[i].tag <- mkRegU;
    end

    Reg#(Bool) read        <- mkReg(False);
    Reg#(Bool) fetchActive <- mkReg(False);

    Reg#(Bit#(slotsAddrBits))  expectedSlot          <- mkReg(0);
    Reg#(Bit#(tagAddrBits))    expectedTag           <- mkReg(0);
    Reg#(Bit#(cacheLineBits))  currentWriteCacheLine <- mkReg(0);
    Reg#(Bit#(cacheLineBytes)) currentWriteEN        <- mkReg(0);
    Reg#(Bool)                 isMiss                <- mkReg(False);
    Reg#(UInt#(12))            cycleCounter          <- mkRegU();
    Reg#(UInt#(20))            requestsSinceLastMiss <- mkReg(0);

    Reg#(UInt#(32))            clockCounter          <- mkReg(0);
    Reg#(UInt#(32))            requestStart          <- mkReg(0);

    function Bit#(tagAddrBits) getTag(Bit#(32) addr);
        return addr[31:valueOf(addrBitsNoTag)];
    endfunction

    function Bit#(slotsAddrBits) getSlot(Bit#(32) addr);
        return addr[valueOf(addrBitsNoTag) - 1:valueOf(cacheLineAddrBits)];
    endfunction

    function Bit#(32) toAddr(Bit#(tagAddrBits) tag, Bit#(slotsAddrBits) slot);
        return {tag,slot,0};
    endfunction

    (* mutually_exclusive="fetchCorrectLine,updateCycleCounter" *)
    (* descending_urgency="writeResponse,updateCycleCounter" *)
    rule fetchCorrectLine(actionInProgress
                          && (expectedTag != cacheLines[expectedSlot].tag
                              || !cacheLines[expectedSlot].valid)
                          && !fetchActive);
        $display("(%0d) Read: %b Cache Miss for %x != %x @ %x IsDirty: %d. %d requests without miss.",
            $time, read, expectedTag, cacheLines[expectedSlot].tag, expectedSlot,
            cacheLines[expectedSlot].dirty, requestsSinceLastMiss);
```

```
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;
```

```
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))
```

```
        provisos(
            Div#(cacheLineBits, wordBits, wordsPerCacheLine),
            Div#(wordBits, 8, wordBytes),
            Div#(cacheLineBits, 8, cacheLineBytes)
            );
        FIFO#(RAMRequest#(wordBits, cacheLineBits)) requestOut <- mkBypassFIFO();
        FIFO#(Bit#(cacheLineBits))                  dataIn     <- mkBypassFIFO();

        Reg#(Bool)                                          dirty            <- mkReg(False);
        Reg#(Bool)                                          fetchInProgress  <- mkReg(False);
        Reg#(Bit#(cacheLineBits))                           currentCacheLine <- mkRegU();
        Reg#(Bit#(32))                                      readAddress      <- mkReg(0);
        Reg#(Bit#(32))                                      writeAddress     <- mkReg(0);
        Reg#(Bool)                                          requestDone      <- mkReg(False);

        rule operation (fetchInProgress && !requestDone);
            RAMRequest#(wordBits, cacheLineBits) req = tagged Read 0;
            if(dirty) begin
                Bit#(cacheLineBytes) writeEN = (1 << valueOf(cacheLineBytes)) - 1;
                req = tagged Write tuple3(writeAddress, currentCacheLine, writeEN);
            end else begin
                req = tagged Read readAddress;
            end
            requestDone <= True;
            requestOut.enq(req);
        endrule

        rule operationRead (fetchInProgress && !dirty && requestDone);
            let result <- toGet(dataIn).get();
            currentCacheLine <= result;
            fetchInProgress <= False;
        endrule

        rule switchOperation (fetchInProgress && dirty && requestDone);
            dirty           <= False;
            requestDone     <= False;
        endrule

        interface Server toUser;
            interface Put request;
                method Action put(FetchRequest#(cacheLineBits) req) if(!fetchInProgress);
                    case(req) matches
                        tagged FetchDirty .v: begin
                            dirty            <= True;
                            readAddress      <= tpl_1(v);
                            writeAddress     <= tpl_2(v);
                            currentCacheLine <= tpl_3(v);
                            fetchInProgress <= True;
                            requestDone     <= False;
                        end
                        tagged Fetch .v: begin
                            dirty <= False;
                            readAddress      <= v;
                            fetchInProgress <= True;
                            requestDone     <= False;
```

```
                        end
                        tagged JustWrite .v: begin
                            $display("Not handled!");
                            fetchInProgress <= False;
                            requestDone     <= False;
                        end
                    endcase
                endmethod
            endinterface

            interface Get response;
                method ActionValue#(Bit#(cacheLineBits)) get() if(!fetchInProgress);
                    return currentCacheLine;
                endmethod
            endinterface
        endinterface

        interface Client toRAM;
            interface Get request  = toGet(requestOut);
            interface Put response = toPut(dataIn);
        endinterface
    endmodule

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.

```
package Testbench;

    import GetPut :: *;
    import ClientServer :: *;
    import Connectable :: *;
    import Vector :: *;
    import StmtFSM :: *;
    import FIFO :: *;

    import RAM :: *;
    import Cache :: *;
    import CacheLine  :: *;
    import CacheLineToWord :: *;
```

```
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 = (
```

```
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                              l2cache.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                              l2cache.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                             l2cache.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
```

```
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                 endseq);
141
142         mkAutoFSM(s);
143     endmodule
144
145 endpackage
```