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#(512), Bit#(64)) Write;
13                            }
14                            RAMRequest deriving (Bits, Eq, FShow);
15
16      typedef union tagged {Bit#(32) Read;
17                            Tuple3#(Bit#(32), Bit#(32), Bit#(4)) Write;
18                            }
19                            RAMRequestOnWords deriving (Bits, Eq, FShow);
20
21      typedef Server#(RAMRequest, Bit#(512)) RAMServer;
22      typedef Client#(RAMRequest, Bit#(512)) RAMClient;
23
24      typedef Server#(RAMRequestOnWords, Bit#(32)) RAMServerOnWords;
25      typedef Client#(RAMRequestOnWords, Bit#(32)) RAMClientOnWords;
26
27      module mkRAM(RAMServerOnWords);
28          FIFO#(RAMRequestOnWords) requestIn <- mkBypassFIFO();
29          FIFO#(Bit#(32)) dataOut <- mkBypassFIFO();
30
31          BRAM_Configure conf = defaultValue;
32          conf.memorySize = 262144;
33
34          BRAM1PortBE#(Bit#(18), Bit#(32), 4) bram <- mkBRAM1ServerBE(conf);
35
36          rule handleRead (requestIn.first() matches tagged Read .v);
37              requestIn.deq();
38              Bit#(30) addr_l = truncateLSB(v);
39              Bit#(18) addr   = truncate(addr_l);
40              BRAMRequestBE#(Bit#(18), Bit#(32), 4) req =
41                  BRAMRequestBE{writeen: 0, responseOnWrite: False, address: addr, datain: 0};
42              bram.portA.request.put(req);
43          endrule
44
45          rule handleReadResponse;
46              let response <- bram.portA.response.get();
47              dataOut.enq(extend(response));
48          endrule
49
50          rule handleWrite (requestIn.first() matches tagged Write .v);
51              requestIn.deq();
52              Bit#(30) addr_l = truncateLSB(tpl_1(v));
53              Bit#(18) addr   = truncate(addr_l);
54              if(tpl_3(v) != 0) begin
55                  BRAMRequestBE#(Bit#(18), Bit#(32), 4) req =
56                      BRAMRequestBE{writeen: tpl_3(v), responseOnWrite: False, address: addr, datain: tpl_2(v)};
57                  bram.portA.request.put(req);
```

```
            end
        endrule

        interface Put request  = toPut(requestIn);
        interface Get response = toGet(dataOut);
    endmodule
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

```
package CacheLineToWord;

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

    interface CacheLineToWord;
        interface RAMClient toRAM;
        interface RAMServerOnWords toUser;
    endinterface

    module mkCacheLineToWord(CacheLineToWord);

        FIFO#(RAMRequestOnWords) requestToRam   <- mkBypassFIFO();
        FIFO#(Bit#(32)) responseToUser <- mkBypassFIFO();

        Reg#(Bit#(4)) selectedWord <- mkReg(0);
        Reg#(Bool) transferActive[2] <- mkCReg(2, False);

        interface Client toRAM;
            interface Get request;
                method ActionValue#(RAMRequest) get();
                    let reqFromUser <- toGet(requestToRam).get();
                    RAMRequest reqToRam = tagged Read 0;

                    case(reqFromUser) matches
                        tagged Write .v: begin
                            let reqAddress = tpl_1(v);
                            let reqData    = tpl_2(v);
                            let reqWriteEN = tpl_3(v);

                            Bit#(6) firstByteOfWord  = truncate(reqAddress);
                            Bit#(64) writeen = 0;
                            for(Integer i = 0; i < 4; i = i + 1) begin
                                writeen[firstByteOfWord + fromInteger(i)] = reqWriteEN[i];
                            end
```

```
41
42                              Bit#(6) selWordT = truncate(reqAddress);
43                              Bit#(4) selWord = truncateLSB(selWordT);
44                              Vector#(16, Bit#(32)) cacheLine = unpack(0);
45                              let cacheLineUpdated = update(cacheLine, selWord, reqData);
46                              reqToRam = tagged Write tuple3(reqAddress, pack(cacheLineUpdated), writeen);
47                          end
48                          tagged Read  .v: begin
49                              reqToRam = tagged Read v;
50                              Bit#(6) selWordT  = truncate(v);
51                                  selectedWord <= truncateLSB(selWordT);
52                          end
53                      endcase
54                      return reqToRam;
55                  endmethod
56              endinterface
57
58              interface Put response;
59                  method Action put(Bit#(512) cacheLine);
60                      Vector#(16, Bit#(32)) cacheLineVec = unpack(cacheLine);
61                      responseToUser.enq(cacheLineVec[selectedWord]);
62                  endmethod
63              endinterface
64          endinterface
65
66          interface Server toUser;
67              interface Put request;
68                  method Action put(RAMRequestOnWords req) if(!transferActive[1]);
69                      if(req matches tagged Read .v) transferActive[1] <= True;
70                      requestToRam.enq(req);
71                  endmethod
72              endinterface
73
74              interface Get response;
75                  method ActionValue#(Bit#(32)) get();
76                      transferActive[0] <= False;
77                      let ret <- toGet(responseToUser).get();
78                      return ret;
79                  endmethod
80              endinterface
81          endinterface
82      endmodule
83
84  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;
12          interface RAMServer toUser;
13      endinterface
14
15      module mkCacheLineRAM(CacheLineRAM);
16          FIFO#(RAMRequest) requestIn <- mkFIFO();
17          FIFO#(Bit#(512))   dataOut  <- mkFIFO();
18
19          RAMServerOnWords ram <- mkRAM();
20          FetchCacheLine fetcher <- mkFetchCacheLine();
21          mkConnection(ram, fetcher.toRAM);
22
23          Reg#(Bool)                       actionInProgress <- mkReg(False);
24          Reg#(Bool)                                   read <- mkReg(False);
25          Reg#(Bool)                            fetchActive <- mkReg(False);
26
27          Reg#(Bit#(32))                   expectedCacheLineAddr <- mkReg(0);
28          Reg#(Bit#(512))         currentWriteCacheLine <- mkReg(0);
29          Reg#(Bit#(64))               currentWriteEN <- mkReg(0);
30
31          Reg#(UInt#(32))                              requestTimer <- mkReg(0);
32
33          rule countRequest (fetchActive);
34              requestTimer <= requestTimer + 1;
35          endrule
36
37          rule fetchCorrectLine(actionInProgress && !fetchActive && read);
38              fetchActive          <= True;
39              let req = tagged Fetch expectedCacheLineAddr;
40              fetcher.toUser.request.put(req);
41
42              requestTimer         <= 0;
43          endrule
44
45          rule forwardWriteRequest(actionInProgress && !fetchActive && !read);
46              let req = tagged JustWrite tuple3(expectedCacheLineAddr, currentWriteCacheLine, currentWriteEN);
47              fetcher.toUser.request.put(req);
48              actionInProgress <= False;
49          endrule
50
51          rule handleFetcherResponse (actionInProgress && fetchActive && read);
52              let fetchedLine  <- fetcher.toUser.response.get();
53              fetchActive      <= False;
54              dataOut.enq(fetchedLine);
55              actionInProgress <= False;
56          endrule
57
58
59          rule handleRead (requestIn.first() matches tagged Read .v &&& !actionInProgress);
60              requestIn.deq();
61              Bit#(6) mask = 6'h3f;
62              Bit#(32) lineAddress = v & ~zeroExtend(mask);
63
64              expectedCacheLineAddr     <= lineAddress;
```

```
65              read                        <= True;
66              actionInProgress            <= True;
67          endrule
68
69          rule handleWriteRequest (requestIn.first() matches tagged Write .v &&& !actionInProgress);
70              requestIn.deq();
71
72              let address = tpl_1(v);
73              Bit#(6) mask = 6'h3f;
74              Bit#(32) lineAddress = address & ~zeroExtend(mask);
75              let data    = tpl_2(v);
76              let writeen = tpl_3(v);
77
78              expectedCacheLineAddr     <= lineAddress;
79              currentWriteCacheLine     <= unpack(data);
80              currentWriteEN            <= unpack(writeen);
81              read                      <= False;
82              actionInProgress          <= True;
83          endrule
84
85          interface RAMServer toUser;
86              interface Put request  = toPut(requestIn);
87              interface Get response = toGet(dataOut);
88          endinterface
89      endmodule
90
91      typedef union tagged {
92                                              Bit#(32) Fetch;
93          Tuple3#(Bit#(32), Bit#(32), Bit#(512)) FetchDirty;
94          Tuple3#(Bit#(32), Bit#(512), Bit#(64)) JustWrite;
95      } FetchRequest deriving (Bits, Eq, FShow);
96
97      interface FetchCacheLine;
98          interface RAMClientOnWords toRAM;
99          interface Server#(FetchRequest, Bit#(512)) toUser;
100     endinterface
101
102     module mkFetchCacheLine(FetchCacheLine);
103         FIFO#(RAMRequestOnWords) requestOut <- mkBypassFIFO();
104         FIFO#(Bit#(32)) dataIn <- mkBypassFIFO();
105
106         Reg#(Bool) dirty <- mkReg(False);
107         Reg#(Bool) justWrite <- mkReg(False);
108         Reg#(Bool) fetchInProgress <- mkReg(False);
109         Reg#(UInt#(5)) currentWord <- mkReg(0);
110         Reg#(UInt#(5)) requestedWords <- mkReg(0);
111         Vector#(16, Reg#(Bit#(32))) currentCacheLine <- replicateM(mkRegU);
112         Vector#(16, Reg#(Bit#(4))) currentWriteEN <- replicateM(mkRegU);
113         Reg#(Bit#(32)) readAddress <- mkReg(0);
114         Reg#(Bit#(32)) writeAddress <- mkReg(0);
115
116         (* conflict_free="operation, operationRead" *)
117         rule operation (fetchInProgress && currentWord    < 16
118                                         && requestedWords < 16);
119             RAMRequestOnWords req = tagged Read 0;
```

```
120             if(dirty) begin
121                 writeAddress   <= writeAddress + 4;
122                 currentWord    <= currentWord  + 1;
123                 Bit#(4) writeEN = 0;
124                 for(Integer i = 0; i < valueOf(4); i = i + 1) begin
125                     writeEN[i] = currentWriteEN[currentWord][i];
126                 end
127                 req = tagged Write tuple3(writeAddress, currentCacheLine[currentWord], writeEN);
128             end else begin
129                 readAddress    <=  readAddress + 4;
130                 requestedWords <=  requestedWords + 1;
131                 req = tagged Read readAddress;
132             end
133             requestOut.enq(req);
134         endrule
135
136         rule operationRead (fetchInProgress && !dirty && currentWord < 16);
137             currentWord <= currentWord + 1;
138             let result <- toGet(dataIn).get();
139             currentCacheLine[currentWord] <= result;
140         endrule
141
142         rule switchOperation (fetchInProgress && dirty && currentWord >= 16);
143             dirty           <= False;
144             currentWord     <= 0;
145             if(justWrite) begin
146                 fetchInProgress <= False;
147             end
148         endrule
149
150         rule operationDone (fetchInProgress && !dirty && currentWord >= 16);
151             fetchInProgress <= False;
152         endrule
153
154         interface Server toUser;
155             interface Put request;
156                 method Action put(FetchRequest req) if(!fetchInProgress);
157                     fetchInProgress <= True;
158                     currentWord <= 0;
159                     requestedWords <= 0;
160                     case(req) matches
161                         tagged FetchDirty .v: begin
162                             dirty        <= True;
163                             justWrite    <= False;
164                             readAddress  <= tpl_1(v);
165                             writeAddress <= tpl_2(v);
166                             writeVReg(currentCacheLine, unpack(tpl_3(v)));
167                         end
168                         tagged JustWrite .v: begin
169                             dirty        <= True;
170                             justWrite    <= True;
171                             writeAddress <= tpl_1(v);
172                             writeVReg(currentCacheLine, unpack(tpl_2(v)));
173                             writeVReg(currentWriteEN,   unpack(tpl_3(v)));
174                         end
```

```
                        tagged Fetch .v: begin
                            dirty     <= False;
                            justWrite <= False;
                        end
                    endcase
                endmethod
            endinterface

            interface Get response;
                method ActionValue#(Bit#(512)) get() if(!fetchInProgress);
                    return pack(readVReg(currentCacheLine));
                endmethod
            endinterface
        endinterface

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

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.

```
package Cache;

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

    typedef  32 AddressBits;
    typedef  13 AddressBitsWithoutTag;
    typedef  19 TagBits;
    typedef   7 SlotBits;
    typedef 128 Slots;
    typedef 512 CacheLineBits;
    typedef 6   CacheLineAddrBits;

    typedef struct {
        Reg#(Bool) dirty;
        Reg#(Bool) valid;
        Reg#(Bit#(19)) tag;
        Reg#(Bit#(512)) cacheLine;
    } CacheLine;

    interface Cache;
        interface RAMClient toRAM;
```

```
29          interface RAMServer toUser;
30          method    Action clearCache();
31      endinterface
32
33      module mkCache(Cache);
34          FIFO#(RAMRequest) requestIn <- mkFIFO();
35          FIFO#(Bit#(512)) dataOut <- mkFIFO();
36
37          FIFO#(RAMRequest) requestOut <- mkFIFO();
38          FIFO#(Bit#(512)) dataIn <- mkFIFO();
39
40          FetchCacheLineWhole fetcher <- mkFetchCacheLineWhole();
41          mkConnection(fetcher.toRAM, toGPServer(requestOut, dataIn));
42
43          Reg#(Bool) actionInProgress <- mkReg(False);
44
45          Vector#(128, CacheLine) cacheLines;
46          for(Integer i = 0; i < 128; i = i + 1) begin
47              cacheLines[i].dirty <- mkReg(False);
48              cacheLines[i].valid <- mkReg(False);
49              cacheLines[i].cacheLine <- mkRegU;
50              cacheLines[i].tag <- mkRegU;
51          end
52
53          Reg#(Bool) read        <- mkReg(False);
54          Reg#(Bool) fetchActive <- mkReg(False);
55
56          Reg#(Bit#(7))     expectedSlot          <- mkReg(0);
57          Reg#(Bit#(19))    expectedTag           <- mkReg(0);
58          Reg#(Bit#(512))   currentWriteCacheLine <- mkReg(0);
59          Reg#(Bit#(64))    currentWriteEN        <- mkReg(0);
60          Reg#(Bool)                isMiss                <- mkReg(False);
61          Reg#(UInt#(12))           cycleCounter          <- mkRegU();
62          Reg#(UInt#(20))           requestsSinceLastMiss <- mkReg(0);
63
64          Reg#(UInt#(32))           clockCounter          <- mkReg(0);
65          Reg#(UInt#(32))           requestStart          <- mkReg(0);
66
67          function Bit#(19) getTag(Bit#(32) addr);
68              return addr[31:13];
69          endfunction
70
71          function Bit#(7) getSlot(Bit#(32) addr);
72              return addr[13 - 1:6];
73          endfunction
74
75          function Bit#(32) toAddr(Bit#(19) tag, Bit#(7) slot);
76              return {tag,slot,0};
77          endfunction
78
79          (* mutually_exclusive="fetchCorrectLine,updateCycleCounter" *)
80          (* descending_urgency="writeResponse,updateCycleCounter" *)
81          rule fetchCorrectLine(actionInProgress
82                                && (expectedTag != cacheLines[expectedSlot].tag
83                                  || !cacheLines[expectedSlot].valid)
```

```
                            && !fetchActive);
        isMiss <= True;
        requestsSinceLastMiss <= 0;
        cycleCounter <= 0;
        FetchRequest req = tagged Fetch toAddr(cacheLines[expectedSlot].tag,
                                                          expectedSlot);
        if(cacheLines[expectedSlot].dirty) begin
            req = tagged FetchDirty tuple3(toAddr(expectedTag, expectedSlot),
                                           toAddr(cacheLines[expectedSlot].tag, expectedSlot),
                                           cacheLines[expectedSlot].cacheLine);
        end
        cacheLines[expectedSlot].valid               <= False;
        cacheLines[expectedSlot].dirty               <= False;
        cacheLines[expectedSlot].tag                 <= expectedTag;

        fetchActive          <= True;

        fetcher.toUser.request.put(req);
    endrule

    rule updateCycleCounter(isMiss);
        cycleCounter <= cycleCounter + 1;
    endrule

    rule updateClockCounter;
        clockCounter <= clockCounter + 1;
    endrule

    rule handleFetcherResponse (actionInProgress && fetchActive);
        let fetchedLine  <- fetcher.toUser.response.get();
        cacheLines[expectedSlot].valid <= True;
        cacheLines[expectedSlot].cacheLine <= fetchedLine;

        fetchActive      <= False;
    endrule

    rule writeResponse(!fetchActive && cacheLines[expectedSlot].tag == expectedTag
                                    && cacheLines[expectedSlot].valid
                                    && actionInProgress);
        if(read) begin
            dataOut.enq(cacheLines[expectedSlot].cacheLine);
        end else begin
            Bit#(512) writeENBits = 0;
            for(Integer i = 0; i < 64; i = i + 1) begin
                for(Integer j = 0; j < 8; j = j + 1) begin
                    writeENBits[(i * 8) + j] = currentWriteEN[i];
                end
            end
            cacheLines[expectedSlot].cacheLine <= (cacheLines[expectedSlot].cacheLine & ~writeENBits)
                                                | (currentWriteCacheLine & writeENBits);
            cacheLines[expectedSlot].dirty <= True;
        end
        requestsSinceLastMiss <= requestsSinceLastMiss + 1;
        actionInProgress <= False;
        isMiss <= False;
```

```
139         endrule
140
141
142         rule handleRead (requestIn.first() matches tagged Read .v &&& !actionInProgress);
143             requestIn.deq();
144
145             expectedTag               <= getTag(v);
146             expectedSlot              <= getSlot(v);
147             read                      <= True;
148             actionInProgress          <= True;
149
150             requestStart <= clockCounter;
151         endrule
152
153         rule handleWriteRequest (requestIn.first() matches tagged Write .v &&& !actionInProgress);
154             requestIn.deq();
155
156             let address = tpl_1(v);
157             let data    = tpl_2(v);
158             let writeen = tpl_3(v);
159
160             expectedTag               <= getTag(address);
161             expectedSlot              <= getSlot(address);
162             currentWriteCacheLine     <= unpack(data);
163             currentWriteEN            <= unpack(writeen);
164             read                      <= False;
165             actionInProgress          <= True;
166
167             requestStart <= clockCounter;
168         endrule
169
170         method Action clearCache() if(!actionInProgress);
171             for(Integer i = 0; i < 128; i = i + 1) begin
172                 cacheLines[i].dirty <= False;
173                 cacheLines[i].valid <= False;
174             end
175         endmethod
176
177         interface Client toRAM;
178             interface Get request  = toGet(requestOut);
179             interface Put response = toPut(dataIn);
180         endinterface
181         interface Server toUser;
182             interface Put request  = toPut(requestIn);
183             interface Get response = toGet(dataOut);
184         endinterface
185     endmodule
186
187     interface FetchCacheLineWhole;
188         interface RAMClient toRAM;
189         interface Server#(FetchRequest, Bit#(512)) toUser;
190     endinterface
191
192     module mkFetchCacheLineWhole(FetchCacheLineWhole);
193         FIFO#(RAMRequest) requestOut <- mkBypassFIFO();
```

```
194         FIFO#(Bit#(512)) dataIn     <- mkBypassFIFO();
195
196         Reg#(Bool)                                      dirty           <- mkReg(False);
197         Reg#(Bool)                                      fetchInProgress <- mkReg(False);
198         Reg#(Bit#(512))                                 currentCacheLine <- mkRegU();
199         Reg#(Bit#(32))                                  readAddress     <- mkReg(0);
200         Reg#(Bit#(32))                                  writeAddress    <- mkReg(0);
201         Reg#(Bool)                                      requestDone     <- mkReg(False);
202
203         rule operation (fetchInProgress && !requestDone);
204             RAMRequest req = tagged Read 0;
205             if(dirty) begin
206                 Bit#(64) writeEN = 64'hffffffffffffffff;
207                 req = tagged Write tuple3(writeAddress, currentCacheLine, writeEN);
208             end else begin
209                 req = tagged Read readAddress;
210             end
211             requestDone <= True;
212             requestOut.enq(req);
213         endrule
214
215         rule operationRead (fetchInProgress && !dirty && requestDone);
216             let result <- toGet(dataIn).get();
217             currentCacheLine <= result;
218             fetchInProgress <= False;
219         endrule
220
221         rule switchOperation (fetchInProgress && dirty && requestDone);
222             dirty           <= False;
223             requestDone     <= False;
224         endrule
225
226         interface Server toUser;
227             interface Put request;
228                 method Action put(FetchRequest req) if(!fetchInProgress);
229                     case(req) matches
230                         tagged FetchDirty .v: begin
231                             dirty           <= True;
232                             readAddress     <= tpl_1(v);
233                             writeAddress    <= tpl_2(v);
234                             currentCacheLine <= tpl_3(v);
235                             fetchInProgress <= True;
236                             requestDone     <= False;
237                         end
238                         tagged Fetch .v: begin
239                             dirty <= False;
240                             readAddress     <= v;
241                             fetchInProgress <= True;
242                             requestDone     <= False;
243                         end
244                         tagged JustWrite .v: begin
245                             $display("Not handled!");
246                             fetchInProgress <= False;
247                             requestDone     <= False;
248                         end
```

```
249                     endcase
250                 endmethod
251             endinterface
252
253             interface Get response;
254                 method ActionValue#(Bit#(512)) get() if(!fetchInProgress);
255                     return currentCacheLine;
256                 endmethod
257             endinterface
258         endinterface
259
260         interface Client toRAM;
261             interface Get request  = toGet(requestOut);
262             interface Put response = toPut(dataIn);
263         endinterface
264     endmodule
265
266 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 :: *;
14
15     (* synthesize *)
16     module mkTestbench(Empty);
17
18         CacheLineRAM   ram           <- mkCacheLineRAM();
19         CacheLineToWord wordConverter <- mkCacheLineToWord();
```

```
20
21          Cache cache <- mkCache();
22
23          // With Cache
24          mkConnection(cache.toRAM, ram.toUser);
25          mkConnection(wordConverter.toRAM, cache.toUser);
26
27          Reg#(UInt#(32)) clockCounter <- mkReg(0);
28          Reg#(UInt#(32)) startTime    <- mkRegU;
29
30          rule count;
31              clockCounter <= clockCounter + 1;
32          endrule
33
34          Vector#(5, UInt#(30)) testPoints;
35          testPoints[0] = 128;
36          testPoints[1] = 256;
37          testPoints[2] = 512;
38          testPoints[3] = 1024;
39          testPoints[4] = 16384;
40
41          Vector#(5, UInt#(30)) skipPoints;
42          skipPoints[0] = 1;
43          skipPoints[1] = 8;
44          skipPoints[2] = 16;
45          skipPoints[3] = 32;
46          skipPoints[4] = 128;
47
48          Reg#(UInt#(32)) testCounter <- mkRegU;
49          Reg#(UInt#(32)) skipCounter <- mkRegU;
50
51          Reg#(UInt#(30)) loopCounter <- mkReg(0);
52          Stmt s = (
53   seq
54      for(skipCounter <= 0; skipCounter < 5; skipCounter <= skipCounter + 1) seq
55          $display("Write test %d", skipCounter);
56          for(testCounter <= 0; testCounter < 5; testCounter <= testCounter + 1) seq
57              cache.clearCache();
58              startTime <= clockCounter;
59              for(loopCounter <= 0; loopCounter < testPoints[testCounter];
60                  loopCounter <= loopCounter + skipPoints[skipCounter]) seq
61                  action
62                      Bit#(4) byteEnable = 4'hF;
63                      RAMRequestOnWords req = tagged Write tuple3({pack(loopCounter),2'b0},
64                                                  extend(pack(loopCounter)), byteEnable);
65                      wordConverter.toUser.request.put(req);
66                  endaction
67              endseq
68              $display("%d: %d", testCounter, clockCounter - startTime);
69              delay(128);
70          endseq
71          $display("Read test %d", skipCounter);
72          for(testCounter <= 0; testCounter < 5; testCounter <= testCounter + 1) seq
73              cache.clearCache();
74              startTime <= clockCounter;
```

```
75              for(loopCounter <= 0; loopCounter < testPoints[testCounter];
76                  loopCounter <= loopCounter + skipPoints[skipCounter]) seq
77                  action
78                      Bit#(32) addr = {pack(loopCounter),2'b0};
79                      RAMRequestOnWords req = tagged Read addr;
80                      wordConverter.toUser.request.put(req);
81                  endaction
82                  action
83                      let result <- wordConverter.toUser.response.get();
84                  endaction
85              endseq
86              $display("%d: %d", testCounter, clockCounter - startTime);
87              delay(128);
88          endseq
89      endseq
90  endseq);
91
92          mkAutoFSM(s);
93      endmodule
94
95  endpackage
```