```
1  ----------------------------------------------------------------------------------------------
2  -- Rechnerarchitektur und Eingebettete Systeme
3  -- Uebungszettel 2 - Aufgabe 2: Direct Mapped Cache
4  --
5  -- Der Grossteil der Quelltextkommentierung wurde hier durchgefuehrt, um die Uebersicht im Code zu
6  -- erhalten. Kommentare in den Codeabschnitten dienen groesstenteils zur einfacheren Verfolgung
7  -- des Programmablaufs.
8  --
9  --
10 -- Der Cache hat, entsprechend der Aufgabenstellung, eine Nettogroesse von 16 Byte in 16
11 -- Cachelines.
12 --
13 -- Als Schreibstrategien wurde "Write Back" (WB) verwendet um Schreibzugriffe auf den unter-
14 -- liegenden Speicher zu verringern. Konkret heisst das, dass beim schreibenden Zugriff auf den
15 -- Speicher das Datum nur in den Cache geschrieben wird und die Cacheline als "dirty" (d) markiert
16 -- wird. Der eigentliche Schreibzugriff auf den Speicher wird erst ausgeloest, wenn die Cacheline
17 -- durch einen anderen Wert ersetzt werden soll (Lesen oder Schreiben einer anderen Adresse mit
18 -- gleichem "Tag").
19 --
20 -- Um 16 Byte im Cache Speichern zu koennen werden 16 Cachelines angelegt, die ueber die 4 LSB der
21 -- Speicheradresse (2^4 = 16) indexiert werden. Daraus folgt das die restlichen 4 Bit (MSB: 7 - 4)
22 -- als "Tag" in der Cacheline gespeichert werden muessen. Fuer die Daten werden 8 Bit in der Cache-
23 -- line benoetigt. Zur Verwaltung der Cachelines werden noch ein Valid-Bit und ein Dirty-Bit je
24 -- Cacheline gespeichert. Der eigentliche Cache hat somit eine Brutogroesse von 16* 14Bit = 28Byte.
25 --
26 -- Das Dirty-Bit bestimmt ob die Cacheline ggf. bei Aenderung in den Speicher zurueckgeschrieben
27 -- werden muss. Das Valid-Bit bestimmt ob die Cacheline Daten aus dem Speicher enthaelt.
28 --
29 -- Die Realisierung des Caches wurde ueber vier Prozesse und ein Ausgabenetz ausserhalb der
30 -- umgesetzt.
31 -- Der Prozess "mem_clk_process" generiert den Clock fuer den Speicher, welche mit 1/8 des Cache-
32 -- clocks laeuft (Aufgabenstellung).
33 -- Der Prozess "execute" verarbeitet die Eingaenge des Caches und wird im weiteren Verlauf auch als
34 -- "main" bezeichnet. Er prueft welche Eingaenge gesetzt sind und entscheidet dann je nach Zustand
35 -- des Caches/ der Cacheline und der Eingaben was getan werden soll. Zugriffe auf den langsamen
36 -- Speicher werden an den Prozess "memDriver" weitergeleitet, welcher dann
```

```
    entsprechend die Daten
37  -- in den Speicher schreibt, oder aus dem Speicher liest und an den Prozess
    "cacheDriver" weiter-
38  -- gibt. Der Prozess "cacheDriver" empfaengt Daten aus "main" und aus "memDriver"
    und schreibt die
39  -- Daten entsprechend in den Cache.
40  --
41  -- Die Architektur mit drei Prozessen wurde so gewaehlt um vor allem den Zugriff auf
    den unter-
42  -- liegenden Speicher zu schuetzen, da dieser um einiges laenger benoetigt bis er
    eine Anfrage ver-
43  -- arbeitet hat. Durch eine Selbstsperre ("memState") und Synchronisationssignale
    ("memRequest",
44  -- "memContentState") wird sichergestellt, dass der Speicher genug Zeit hat um die
    geforderte
45  -- Aktion auszufuehren.
46  --
47  -- Um den Cache konsistent zu halten und sowohl Cache-Aenderung aus "main" als auch
    aus "memDriver"
48  -- verarbeiten zu koennen - Ein schreibender Zugriff nur aus "main" wuerde nach
    Speicherzugriffen
49  -- einen zusaetzlichen Takt benoetigen und wuerde hoeheren Aufwand bei der
    Synchronisation der
50  -- beiden Prozesse erfordern. - werden alle schreibenden Cachezugriffe durch
    "cacheDriver"
51  -- verwaltet. Zur Synchronisation dienen die Signale "cacheRequestFromMem" und
52  -- "cacheRequestFromMain", ein Lock wird nicht benoetigt, da der Prozess auf
    Aenderungen dieser
53  -- Signale reagiert und beim internen Zugriff auf den Cache keine Verzoegerungen
    beruecksichtigt
54  -- werden muessen. Die zu schreibenden Daten und Adressen werden in Buffern
    zwischengespeichert
55  -- ("main" und "memDriver" haben jeweils ihr eigenen Set von Buffern zum Zugriff auf
    "cacheDriver")
56  ----------------------------------------------------------------------------------
    -------------
57  library IEEE;
58  use IEEE.STD_LOGIC_1164.ALL;
59  USE IEEE.STD_LOGIC_UNSIGNED.all;
60  USE IEEE.NUMERIC_STD.ALL;
61
62  entity CachedMemory is
63    Port (
64      clk     : in  STD_LOGIC;
65      init    : in  STD_LOGIC;
66      dump    : in  STD_LOGIC;
67      reset   : in  STD_LOGIC;
68      re      : in  STD_LOGIC;
69      we      : in  STD_LOGIC;
70      addr    : in  STD_LOGIC_VECTOR (7 downto 0);
71      data_in : in  STD_LOGIC_VECTOR (7 downto 0);
72      output  : out  STD_LOGIC_VECTOR (7 downto 0);
73      ack     : out  STD_LOGIC);
74  end CachedMemory;
75
76  architecture Behavioral of CachedMemory is
77    component Memory
```

```
78        Port (
79          clk: in std_logic;
80          init: in std_logic;
81          dump: in std_logic;
82          reset: in std_logic;
83          re: in std_logic;
84          we: in std_logic;
85          addr: in std_logic_vector(7 downto 0);
86          data_in: in std_logic_vector(7 downto 0);
87          output: out std_logic_vector(7 downto 0)
88          );
89      end component;
90
91      signal mem_clk      : std_logic := '0';
92      signal mem_init     : std_logic := '0';
93      signal mem_dump     : std_logic := '0';
94      signal mem_reset    : std_logic := '0';
95      signal mem_re       : std_logic := '0';
96      signal mem_we       : std_logic := '0';
97      signal mem_addr     : std_logic_vector(7 downto 0) := (others => '0');
98      signal mem_data_in  : std_logic_vector(7 downto 0) := (others => '0');
99      signal mem_output   : std_logic_vector(7 downto 0);
100
101
102     type cacheLine is record
103       v : STD_LOGIC;
104       d : STD_LOGIC;
105       tag : STD_LOGIC_VECTOR (3 downto 0);
106       data : STD_LOGIC_VECTOR (7 downto 0);
107     end record;
108     type cacheStruct is array (0 to 15) of cacheLine;
109     signal cache : cacheStruct;
110
111     -- ==========
112     -- Memory states and buffer
113     -- ==========
114     -- memState is used to lock the main process and to ensure in memDriver that
        access to mem is
115     -- valid (e.g. one mem_clk has finished). It is set via memDriver.
116     type memStates is (memIdle, memBusy);
117     signal memState : memStates := memIdle;
118
119     -- memRequest is used by main to tell memDriver what it has to do. It is set by
        main.
120     type memRequests is ( noMemRequest, memRequestInit, memRequestReset, memRequestRead
        ,
121                           memRequestWB, memRequestWBForDump, memRequestDump);
122     signal memRequest : memRequests := noMemRequest;
123
124     -- memContenState is used by memDriver to tell main whether it has finished a
        certain request or
125     -- not. It is set by memDriver.
126     type memContentStates is (memContentUndefined, memContentInit, memContentReset,
127                               memContentRead, memContentWB, memContentDumped);
128     signal memContentState : memContentStates := memContentUndefined;
129
130     -- the buffer_mem_clk is the clock generated by mem_clk_process for mem. It is not
```

```
    directly
131     -- connected to mem, to enable memDriver to set mem's inputs first.
132     signal buffer_mem_clk   : std_logic := '0';
133     -- buffer_mem_addr is used by main to buffer the line that has to be written back
    during dump.
134     signal buffer_mem_addr  : std_logic_vector(7 downto 0) := (others => '0');
135
136
137     -- ==========
138     -- Cache states and buffer
139     -- ==========
140     -- cacheRequestFromMem is used by memDriver to tell cacheDriver what it has to do.
    It is set by
141     -- memDriver.
142     -- cacheRequestFromMain is used by main to tell cacheDriver what it has to do. It
    is set by main.
143     -- Two signals are needed, since only one process can write to one single signal
144     type cacheRequests is ( noCacheRequest, cacheRequestInit, cacheRequestReset,
145                             cacheRequestWrite, cacheRequestCleanLine);
146     signal cacheRequestFromMem  : cacheRequests := noCacheRequest;
147     signal cacheRequestFromMain : cacheRequests := noCacheRequest;
148
149     -- this set of buffers used by memDriver to tell cacheDriver what and where he has
    to update the
150     -- cache.
151     signal buffer_cacheFromMem_addr : std_logic_vector(7 downto 0) := (others => '0');
152     signal buffer_cacheFromMem_data : std_logic_vector(7 downto 0) := (others => '0');
153     signal buffer_cacheFromMem_d    : std_logic := '0';
154     signal buffer_cacheFromMem_v    : std_logic := '0';
155
156     -- this set of buffers used by main to tell cacheDriver what and where he has to
    update the
157     -- cache.
158     signal buffer_cacheFromMain_addr : std_logic_vector(7 downto 0) := (others => '0');
159     signal buffer_cacheFromMain_data : std_logic_vector(7 downto 0) := (others => '0');
160     signal buffer_cacheFromMain_d    : std_logic := '0';
161     signal buffer_cacheFromMain_v    : std_logic := '0';
162
163     -- these signals are directly connected to the cache's outputs ack and output.
    They are set in
164     -- main.
165     signal bufferAck : STD_LOGIC := '0';
166     signal bufferOut : STD_LOGIC_VECTOR (7 downto 0) := (others => '0');
167
168
169   begin
170
171     -- Clock process definitions
172     -- generates a clock signal in buffer_mem_clk with 1/8 the frequency of clk
173     mem_clk_process :process (clk)
174       variable clkCnt : integer := 0;
175       constant clkDiv : integer := 8; -- muss groeßer 1 und durch 2 teilbar sein.
176     begin
177       if rising_edge(clk) then
178         if clkCnt = (clkDiv / 2 - 1) then
179           clkCnt := 0;
180           buffer_mem_clk <= not(buffer_mem_clk);
```

```
181         else
182           clkCnt := clkCnt + 1;
183         end if;
184       end if;
185     end process;
186
187
188     mem: Memory PORT MAP (
189       clk     => mem_clk,
190       init    => mem_init,
191       dump    => mem_dump,
192       reset   => mem_reset,
193       re      => mem_re,
194       we      => mem_we,
195       addr    => mem_addr,
196       data_in => mem_data_in,
197       output  => mem_output
198     );
199
200
201     -- ====================
202     -- Controlls access to the memory mem.
203     -- ====================
204     memDriver: process (buffer_mem_clk)
205       -- holds the last value of memRequest and is set in idle-mode. It is used in
          busy-mode to
206       -- determine what was done.
207       variable lastMemRequest : memRequests := noMemRequest;
208       -- holds the last value of the cache address. It is also set in idle-mode and
          used in busy-mode
209       -- to tell the cacheDrive which cacheline he has to update.
210       variable lastMemAddr : std_logic_vector(7 downto 0) := (others => '0');
211     begin
212       -- preset cacheDriver request. Located here to make sure the signal is reset
          between mem_clks,
213       -- since the cacheDriver only reacts on changing of the signal and other states
          than
214       -- noCacheRequest.
215       cacheRequestFromMem <= noCacheRequest;
216
217       -- reacts to the slower clock to ensure that the slow mem has enough time to
          process
218       if rising_edge(buffer_mem_clk) then
219         -- if the memDriver is not busy it will check whether an request from main is
            pending.
220         -- the inputs of mem are then set to process the request.
221         if(memState = memIdle) then
222           -- preset mem inputs. if an request is pending, it will set its needed inputs
223           -- initialized here to ensure that if no request is pending nothing is done
              in mem.
224           mem_init    <= '0';
225           mem_dump    <= '0';
226           mem_reset   <= '0';
227           mem_re      <= '0';
228           mem_we      <= '0';
229           mem_addr    <= (others => '0');
230           mem_data_in <= (others => '0');
```

```
231
232           -- save last mem request for later processing in busy
233           lastMemRequest  := memRequest;
234           lastMemAddr     := addr;
235
236           -- *** Init request ***
237           if(memRequest = memRequestInit) then
238             mem_init <= '1';
239             -- tell this process and the main process that mem is busy and its
              contents are undefined.
240             memState <= memBusy;
241             memContentState <= memContentUndefined;
242
243           -- *** Reset request ***
244           elsif(memRequest = memRequestReset) then
245             mem_reset <= '1';
246             -- tell this process and the main process that mem is busy and its
              contents are undefined.
247             memState <= memBusy;
248             memContentState <= memContentUndefined;
249
250           -- *** Read request ***
251           elsif(memRequest = memRequestRead) then
252             mem_re    <= '1';
253             mem_addr  <= lastMemAddr;
254             -- tell this process and the main process that mem is busy and its
              contents are undefined.
255             memState <= memBusy;
256             memContentState <= memContentUndefined;
257
258           -- *** WB request ***
259           elsif(memRequest = memRequestWB) then
260             mem_we    <= '1';
261             mem_addr(3 downto 0) <= lastMemAddr(3 downto 0);
262             mem_addr(7 downto 4) <= cache(to_integer(unsigned(lastMemAddr(3 downto 0
              ))));.tag;
263             mem_data_in <= cache(to_integer(unsigned(lastMemAddr(3 downto 0)))).data;
264
265             -- tell this process and the main process that mem is busy and its
              contents are undefined.
266             memState <= memBusy;
267             memContentState <= memContentUndefined;
268
269           -- *** WB for dump request ***
270           elsif(memRequest = memRequestWBForDump) then
271             -- save the cache line index for later use in busy
272             lastMemAddr := buffer_mem_addr;
273             mem_we    <= '1';
274             mem_addr(3 downto 0) <= lastMemAddr(3 downto 0);
275             mem_addr(7 downto 4) <= cache(to_integer(unsigned(lastMemAddr(3 downto 0
              ))));.tag;
276             mem_data_in <= cache(to_integer(unsigned(lastMemAddr(3 downto 0)))).data;
277
278             -- tell this process and the main process that mem is busy and its
              contents are undefined.
279             memState <= memBusy;
280             memContentState <= memContentUndefined;
```

```
281
282           -- *** Dump request ***
283           elsif(memRequest = memRequestDump) then
284             mem_dump <= '1';
285             -- tell this process and the main process that mem is busy and its
                contents are undefined.
286             memState <= memBusy;
287             memContentState <= memContentUndefined;
288           end if;
289         else
290           -- mem was busy for one mem_clk and should have processed its in-/outputs.
291           -- stop mem from doing unwanted things.
292           mem_init    <= '0';
293           mem_dump    <= '0';
294           mem_reset   <= '0';
295           mem_re      <= '0';
296           mem_we      <= '0';
297           mem_addr    <= (others => '0');
298           mem_data_in <= (others => '0');
299           -- tell main process that mem is idle again.
300           memState <= memIdle;
301
302           -- check last request to call the cacheDriver to update the cache if
              necessary and to tell
303           -- main that it has finished the request
304           -- *** Last request: Init ***
305           if(lastMemRequest = memRequestInit) then
306             -- tell the main process that the content of mem is now initialized.
307             memContentState <= memContentInit;
308             -- no need to set cacheRequestFromMem here, since cache init is controlled
                by main
309
310           -- *** Last request: Reset ***
311           elsif(lastMemRequest = memRequestReset) then
312             -- tell the main process that the content of mem is now reset.
313             memContentState <= memContentReset;
314             -- no need to set cacheRequestFromMem here, since cache init is controlled
                by main
315
316           -- *** Last request: Read ***
317           elsif(lastMemRequest = memRequestRead) then
318             -- check if mem_output is valid
319             if( (mem_output /= "XXXXXXXX") AND (mem_output /= "UUUUUUUU") ) then
320               -- data is valid => write it in the cache via CacheDriver
321               -- buffer cache inputs for cacheDriver
322               buffer_cacheFromMem_addr  <= lastMemAddr;
323               buffer_cacheFromMem_data  <= mem_output;
324               buffer_cacheFromMem_d     <= '0';
325               buffer_cacheFromMem_v     <= '1';
326               -- tell cacheDriver to write data to cache
327               cacheRequestFromMem <= cacheRequestWrite;
328             end if;
329             -- tell the main process that the content of mem is now read (not yet
                used, just here for
330             -- consistency).
331             memContentState <= memContentRead;
332
```

```
333           -- *** Last request: WB ***
334           elsif(lastMemRequest = memRequestWB) then
335             -- update bits in cache line via CacheDriver
336             -- buffer cache inputs for cacheDriver
337             buffer_cacheFromMem_addr  <= lastMemAddr;
338             -- tell cacheDriver to update data in cache
339             cacheRequestFromMem <= cacheRequestCleanLine;
340             -- tell the main process that the content of mem is now wb (not yet used,
                just here for
341             -- consistency).
342             memContentState <= memContentWB;
343
344           -- *** Last request: WB ***
345           elsif(lastMemRequest = memRequestWBForDump) then
346             -- update bits in cache line via CacheDriver
347             -- buffer cache inputs for cacheDriver
348             buffer_cacheFromMem_addr  <= lastMemAddr;
349             -- tell cacheDriver to update data in cache
350             cacheRequestFromMem <= cacheRequestCleanLine;
351             -- tell the main process that the content of mem is now wb (not yet used,
                just here for
352             -- consistency).
353             memContentState <= memContentWB;
354
355           -- *** Last request: Dump ***
356           elsif(lastMemRequest = memRequestDump) then
357             -- tell the main process that the content of mem is now dumped.
358             memContentState <= memContentDumped;
359             -- no need to set cacheRequestFromMem here, since cache dump is controlled
                by main
360           end if;
361         end if;
362       end if;
363
364       -- forward the clock signal to mem after its inputs are set.
365       mem_clk <= buffer_mem_clk;
366     end process;
367
368
369     -- ====================
370     -- Controlls access to the cache.
371     -- ====================
372     cacheDriver: process(cacheRequestFromMem, cacheRequestFromMain)
373       -- variable to temporarily save the index of the addressed cache line to reduce
          the conversion-
374       -- calls from std_logic_vector to integer.
375       variable tmpCacheIndex : integer := 0;
376       -- holds the last value of cacheRequestFromMem and cacheRequestFromMain and is
          set if a
377       -- different request comes in. They are used to determine whether memDriver or
          main has
378       -- triggered the process. If both processes have pending requests, the request
          from main will
379       -- be prioritized.
380       variable prevRequestFromMem  : cacheRequests := noCacheRequest;
381       variable prevRequestFromMain : cacheRequests := noCacheRequest;
382     begin
```

```
383       -- make sure it was cacheRequestFromMem that triggered the process
384       if( cacheRequestFromMem /= prevRequestFromMem) then
385         -- save this request as previous
386         prevRequestFromMem := cacheRequestFromMem;
387         -- buffer the cache index, since the conversion is so long
388         tmpCacheIndex := to_integer(unsigned(buffer_cacheFromMem_addr(3 downto 0)));
389
390         -- *** Request from mem: Write ***
391         if( cacheRequestFromMem = cacheRequestWrite ) then
392           -- write buffered values from mem to cache
393           cache(tmpCacheIndex).tag  <= buffer_cacheFromMem_addr(7 downto 4);
394           cache(tmpCacheIndex).data <= buffer_cacheFromMem_data ;
395           cache(tmpCacheIndex).d    <= buffer_cacheFromMem_d;
396           cache(tmpCacheIndex).v    <= buffer_cacheFromMem_v;
397
398         -- *** Request from mem: CleanLine ***
399         elsif( cacheRequestFromMem = cacheRequestCleanLine ) then
400           -- update dirty bit in cache at buffered index from mem
401           cache(tmpCacheIndex).d <= '0';
402         end if;
403       end if;
404
405
406       -- make sure it was cacheRequestFromMain that triggered the process
407       if( cacheRequestFromMain /= prevRequestFromMain) then
408         -- save this request as previous
409         prevRequestFromMain := cacheRequestFromMain;
410         -- buffer the cache index, since the conversion is so long
411         tmpCacheIndex := to_integer(unsigned(buffer_cacheFromMain_addr(3 downto 0)));
412
413         -- *** Request from main: Init ***
414         if( cacheRequestFromMain = cacheRequestInit ) then
415           -- no need to write back cache, since all values in memory may change.
416           -- no need to set all cacheline values (e.g. tag, d), valid ensures that
              cache access will
417           -- query to mem
418           -- invalidate cache for init
419           for i in 0 to cache'length-1 loop
420             cache(i).v <= '0';
421           end loop;
422
423         -- *** Request from main: Reset ***
424         elsif( cacheRequestFromMain = cacheRequestReset ) then
425           -- no need to write back cache, since all values in memory change
426           -- no need to set all cacheline values (e.g. tag, d), valid ensures that
              cache access will
427           -- query to mem
428           -- invalidate cache for reset
429           for i in 0 to cache'length-1 loop
430             cache(i).v <= '0';
431           end loop;
432
433         -- *** Request from main: Write ***
434         elsif( cacheRequestFromMain = cacheRequestWrite ) then
435           -- write buffered values from main to cache
436           cache(tmpCacheIndex).tag  <= buffer_cacheFromMain_addr(7 downto 4);
437           cache(tmpCacheIndex).data <= buffer_cacheFromMain_data ;
```

```
438           cache(tmpCacheIndex).d    <= buffer_cacheFromMain_d;
439           cache(tmpCacheIndex).v    <= buffer_cacheFromMain_v;
440         end if;
441       end if;
442     end process;
443
444
445
446     -- ====================
447     -- Controlls access to the cache.
448     -- ====================
449     execute: process (clk)
450       -- variable to temporarily save the index and tag of the addressed cache line to
          reduce the
451       -- conversion calls from std_logic_vector to integer and to make code lines
          shorter.
452       variable tmpTag   : STD_LOGIC_VECTOR (3 downto 0) := (others => '0');
453       variable tmpIndex : STD_LOGIC_VECTOR (3 downto 0) := (others => '0');
454       variable tmpIntTag   : integer := 0;
455       variable tmpIntIndex : integer := 0;
456
457       -- help variable used to dump mem and cache. It is set to '0' if not all cache
          lines are
458       -- written back to ensure that mem is only dumped with clean cache
459       variable readyForDump : STD_LOGIC := '1';
460
461     begin
462       -- preset cacheDriver request. Located here to make sure the signal is reset
          between clks,
463       -- since the cacheDriver only reacts on changing of the signal and other states
          than
464       -- noCacheRequest.
465       cacheRequestFromMain <= noCacheRequest;
466
467       if rising_edge(clk) then
468         -- aufteilen und zwischenspeichern der adresse zur leicheren handhabung
469         tmpTag    := addr(7 downto 4);
470         tmpIndex  := addr(3 downto 0);
471         tmpIntTag   := to_integer(unsigned(tmpTag));
472         tmpIntIndex := to_integer(unsigned(tmpIndex));
473
474         -- buffer vorbelegen
475         bufferAck <= '0';
476         bufferOut <= "XXXXXXXX";
477
478         memRequest <= noMemRequest;
479
480         -- *** Init ***
481         -- may take up to 3 mem_clks if there are pending mem operations
482         if(init = '1' AND dump = '0' AND reset = '0' AND re = '0' AND we = '0') then
483           -- invalidate output no matter whether mem is busy or not, since the cache
              will be
484           -- invalidated immediately.
485           bufferAck <= '0';
486           bufferOut <= "XXXXXXXX";
487           -- cache is immediately invalidated, since the content of mem will be
              overwritten.
```

```
488           -- the user has to make sure the init signal is set long enough.
489           cacheRequestFromMain <= cacheRequestInit;
490
491           -- wait until mem has finished its things
492           if((memState = memIdle) AND (memContentState /= memContentInit) ) then
493             -- send request to memDriver to init mem
494             memRequest <= memRequestInit;
495           end if;
496         -- *** Init End ***
497
498
499
500         -- *** Dump ***
501         elsif(init = '0' AND dump = '1' AND reset = '0' AND re = '0' AND we = '0') then
502           readyForDump := '1';
503
504           -- write back cache for mem dump (last cache line first)
505           for i in 0 to cache'length-1 loop
506             if( (cache(i).v = '1') AND (cache(i).d = '1') ) then
507               readyForDump := '0';
508               -- wait until mem has finished its things
509               if(memState = memIdle) then
510                 -- save the addr to provide memDriver with the right cache line
511                 buffer_mem_addr(3 downto 0) <= std_logic_vector(to_unsigned(i, 4));
512                 buffer_mem_addr(7 downto 4) <= (others => '0');
513                 -- send request to memDriver to wb mem
514                 memRequest <= memRequestWBForDump;
515               end if;
516             end if;
517           end loop;
518
519           if( readyForDump = '1' ) then
520             -- wait until mem has finished its things
521             if((memState = memIdle) AND (memContentState /= memContentDumped) ) then
522               -- send request to memDriver to reset mem
523               memRequest <= memRequestDump;
524             end if;
525           end if;
526         -- *** Dump End ***
527
528
529
530         -- *** Reset ***
531         elsif(init = '0' AND dump = '0' AND reset = '1' AND re = '0' AND we = '0') then
532           -- invalidate output no matter whether mem is busy or not, since the cache
              will be
533           -- invalidated immediately.
534           bufferAck <= '0';
535           bufferOut <= "XXXXXXXX";
536           -- cache is immediately invalidated, since the content of mem will be
              overwritten.
537           -- the user has to make sure the reset signal is set long enough.
538           cacheRequestFromMain <= cacheRequestReset;
539
540           -- wait until mem has finished its things
541           if((memState = memIdle) AND (memContentState /= memContentReset) ) then
542             -- send request to memDriver to reset mem
```

```
543             memRequest <= memRequestReset;
544           end if;
545         -- *** Reset End ***
546
547
548
549         -- *** Read ***
550         elsif(init = '0' AND dump = '0' AND reset = '0' AND re = '1' AND we = '0') then
551           -- invalidate output no matter whether data is in cache or not. Will be set
552           -- if data is in cache.
553           bufferAck <= '0';
554           bufferOut <= "XXXXXXXX";
555
556           -- check whether data is in cache
557           if(cache(tmpIntIndex).v = '1') then
558             if(tmpTag /= cache(tmpIntIndex).tag) then
559               if(cache(tmpIntIndex).d = '1') then
560                 -- valid, other data, dirty => wb the dirty cacheline
561                 -- wait until mem has finished its things
562                 if(memState = memIdle) then
563                   -- no need to save the addr to ensure that only the value at rising
                      clk is
564                   -- used, since the mem_clk has its rising edge at the same time
                      (disregarding gate
565                   -- latency).
566                   -- send request to memDriver to wb mem
567                   memRequest <= memRequestWB;
568                 end if;
569               else
570                 -- valid, other data, clean => read data from cache
571                 -- wait until mem has finished its things
572                 if(memState = memIdle) then
573                   -- no need to save the addr to ensure that only the value at rising
                      clk is
574                   -- used, since the mem_clk has its rising edge at the same time
                      (disregarding gate
575                   -- latency).
576                   -- send request to memDriver to read mem
577                   memRequest <= memRequestRead;
578                 end if;
579               end if;
580             else
581               -- valid, same data => output data in cache. no need to write back yet
                  if dirty
582               bufferAck <= '1';
583               bufferOut <= cache(tmpIntIndex).data;
584             end if;
585           else
586             -- invalid => read data from cache (if invalid, then no need to check the
                other values)
587             -- wait until mem has finished its things
588             if(memState = memIdle) then
589               -- no need to save the addr to ensure that only the value at rising clk is
590               -- used, since the mem_clk has its rising edge at the same time
                  (disregarding gate
591               -- latency).
592               -- send request to memDriver to read mem
```

```
593               memRequest <= memRequestRead;
594             end if;
595           end if;
596
597
598
599         -- *** Write ***
600         elsif(init = '0' AND dump = '0' AND reset = '0' AND re = '0' AND we = '1') then
601           -- check addressed cacheline
602           if(cache(tmpIntIndex).v = '1') then
603             if(tmpTag /= cache(tmpIntIndex).tag) then
604               if(cache(tmpIntIndex).d = '1') then
605                 -- valid, other data, dirty => wb the dirty cacheline
606                 -- wait until mem has finished its things
607                 if(memState = memIdle) then
608                   -- no need to save the addr to ensure that only the value at rising
                      clk is
609                   -- used, since the mem_clk has its rising edge at the same time
                      (disregarding gate
610                   -- latency).
611                   -- send request to memDriver to wb mem
612                   memRequest <= memRequestWB;
613                 end if;
614               else
615                 -- valid, other data, clean => write data to cache
616                 -- check if input is valid
617                 if( (data_in /= "XXXXXXXX") AND (data_in /= "UUUUUUUU") ) then
618                   -- data is valid => write it in the cache via CacheDriver
619                   -- buffer cache inputs for cacheDriver
620                   buffer_cacheFromMain_addr  <= addr;
621                   buffer_cacheFromMain_data  <= data_in;
622                   buffer_cacheFromMain_d     <= '1';
623                   buffer_cacheFromMain_v     <= '1';
624                   -- tell cacheDriver to write data to cache
625                   cacheRequestFromMain <= cacheRequestWrite;
626                   -- if needed, an write-ack can be send here
627                   -- NOTE OPTIONAL ACK (email)
628                 end if;
629               end if;
630             else
631               -- valid, same data => overwrite write data in cache (no need to check
                  dirty,
632               -- since wb only needed when other data)
633               -- check if input is valid
634               if( (data_in /= "XXXXXXXX") AND (data_in /= "UUUUUUUU") ) then
635                 -- data is valid => write it in the cache via CacheDriver
636                 -- buffer cache inputs for cacheDriver
637                 buffer_cacheFromMain_addr  <= addr;
638                 buffer_cacheFromMain_data  <= data_in;
639                 buffer_cacheFromMain_d     <= '1';
640                 buffer_cacheFromMain_v     <= '1';
641                 -- tell cacheDriver to write data to cache
642                 cacheRequestFromMain <= cacheRequestWrite;
643                 -- if needed, an write-ack can be send here
644                 -- NOTE OPTIONAL ACK (email)
645               end if;
646             end if;
```

```
647           else
648             -- invalid, same data => overwrite write data in cache (if invalid,
649             -- then no need to check the other values)
650             -- check if input is valid
651             if( (data_in /= "XXXXXXXX") AND (data_in /= "UUUUUUUU") ) then
652               -- data is valid => write it in the cache via CacheDriver
653               -- buffer cache inputs for cacheDriver
654               buffer_cacheFromMain_addr  <= addr;
655               buffer_cacheFromMain_data  <= data_in;
656               buffer_cacheFromMain_d     <= '1';
657               buffer_cacheFromMain_v     <= '1';
658               -- tell cacheDriver to write data to cache
659               cacheRequestFromMain <= cacheRequestWrite;
660               -- if needed, an write-ack can be send here
661               -- NOTE OPTIONAL ACK (email)
662             end if;
663           end if;
664
665
666
667         -- *** Nothing ***
668         elsif(init = '0' AND dump = '0' AND reset = '0' AND re = '0' AND we = '0') then
669           -- same outputs if no request to cache is forwarded to hold the current state
670           bufferAck <= bufferAck;
671           bufferOut <= bufferOut;
672
673
674         -- *** Fehlerhafte Eingabe ***
675         else
676           -- disable outputs if erroneous inputs
677           bufferAck <= '0';
678           bufferOut <= "XXXXXXXX";
679
680         end if;
681       end if;
682     end process;
683
684     -- Ausgabenetz
685     ack     <= bufferAck;
686     output  <= bufferOut;
687
688   end Behavioral;
```

```
1  ------------------------------------------------------------------------------------------
   -------------
2  -- Rechnerarchitektur und Eingebettete Systeme
3  -- Uebungszettel 2 - Aufgabe 2: Direct Mapped Cache Testbench
4  --
5  -- Die Testfaelle wurden so gewahlt, dass zum Einen die Anforderungen aus der
   Aufgabestellung ge-
6  -- prueft werden und zum Anderen wurden weitere Testsfaelle gewaehlt, die die
   verschiedenen
7  -- Zustaende des Caches pruefen (Init, Reset, Dump, Read, Write, KeineEingabe).
8  -- Weiterhin wurde die Funktionalitaet WriteBack, ausgeloest sowohl durch read als
   auch durch write
9  -- getestet und das Verhalten der Beschreibung am Beginn der CachedMemory.vhd
   entspricht.
10 -- Es wurde geprueft ob alle 80 Byte des Speichers addressierbar sind.
11 --
12 -- Zuordnung von Testfall zu Aufgabenstellung:
13 -- 1. "Der obige Speicher soll um einen direct mapped Cache erweitert werden, der 16
   Adressen
14 --    vorhalten kann."
15 --    Wird durch Testfall 2 und 3 getestet indem zwei Sets von 16 Adressen
   hintereinander gelesen
16 --    werden. Das Ueberpruefen von ack laesst darauf schliessen ob der 2. set aus
   dem Speicher
17 --    geladen wurde (zeit unterschied bis ack). Weiterhin muessen die gelesenen
   Cachelines mit
18 --    den erwarteten Werten im Speicher ueber einstimmen.
19 --
20 -- 2. "Der Ausgang ack signalisiert hierbei, dass die gewünschten Daten am Ausgang
   anliegen. Der
21 --    Wert wird auf 0 gesetzt, sobald eine Leseanfrage eingeht."
22 --    Wird durch Testfall 8 geprueft.
23 --
24 -- 3. "Beachte hierbei, dass es wichtig ist, dass ack nicht auf 1 steht, bevor die
   Daten wirklich
25 --    vorliegen."
26 --    Wird durch fast alle Tests geprueft (z.B. Testfall 1 und 2) da direkt nach
   rising_edge(ack)
27 --    der output geprueft wird.
28 --
29 -- 4. "Es gilt zu beachten, dass der Cache über eine achtmal schnellere clock
   betrieben wird, als
30 --    der eigentliche Speicher (da sonst ein Cache auch ziemlich sinnlos wäre). Es
   wird also eine
31 --    zweite clock domain benötigt. Hierzu muss ein entsprechender clock divider
   implementiert
32 --    werden."
33 --    Wird durch fast alle Tests geprueft (z.B. Testfall 1) ueber ack = '0' bei
   Speicher
34 --    zugriffen.
35 --
36 -- 5. "Beachte, dass beim dumpen des Caches natürlich die Einträge im Cache selbst
   berücksichtigt
37 --    werden müssen."
38 --    Wird durch Testfall 9 und 10 getestet indem zuerst der Speicher mit Werten
   gefuellt wird,
39 --    wovon einige dirty sind und der Speicher danach gedumpt wird. Das dumpfile
```

```
    dump.dat wurde
40  --    danach manuell ueberpruft, es entsprach den erwarteten Werten.
41  --
42  -- Weitere Tests:
43  -- 6.   Init wird vor Testfall 1 ausgefuehrt und dann durch Testfall 1 geprueft.
    Weiterhin wurde in
44  --      Isim der interne Speicher von mem manuel ueberprueft (Entsprach den
    Vorstellungen)
45  -- 7.   Reset wird vor Testfall 1 ausgefuehrt und dann manuell in Isim ueberprueft
    ob alle Cachelines
46  --      invalid sind.
47  -- 8.   Read mit CacheMiss wird in Testfall 1 geprueft.
48  -- 9.   Read mit CacheHit wird in Testfall 2 geprueft.
49  -- 10.  Read mit WriteBack wird in Testfall 3 geprueft.
50  -- 11.  Mehrfacher Read nach WriteBack wird in Testfall 4 geprueft.
51  -- 12.  Schreiben ohne WriteBack wird in Testfall 5 geprueft.
52  -- 13.  Ob zuvor geschriebene Werte durch Schreiben anderer Daten zurueckgeschrieben
53  --      (writeback) werden, wird in Testfall 6 geprueft.
54  -- 14.  Ob zuvor geschriebene Werte durch Lesen anderer Daten zurueckgeschrieben
55  --      (writeback) werden, wird in Testfall 7 geprueft.
56  -- 15.  Addressierbarkeit aller 80 Speicheradressen, wird in Testfall 9 geprueft.
57  --------------------------------------------------------------------------------
    -------------
```