```vhdl
-- Okay in dieser Aufgabe ist es Ziel, ein Register [oder Arbeitsspeicher] in VHDL zu schreiben.
-- Das Register wird intern mit einem Array aus 80 x 8 Bit Vektoren dargestellt. Die
-- Adressierung wird über die Position des Arrayeleemntes vorgenommen. Da Arrayelemente allerdings
-- mit einem Integer addressiert werden und wir hier einen 'addr' Eingang haben der aus einem 8 Bit
-- Vektor besteht, müssen wir diesen Vektor in einen Integer konvertieren. Später müssen wir die
-- Arraypositon in eine Datei schreiben,hier müssen wir den Integer wieder zurück in einen Bitvektor
-- konvertieren.

-- Für diese Typkonvertierungen laden wir die Bibliotheken IEEE.numeric_std.ALL und IEEE.STD_LOGIC_UNSIGNED.ALL,
-- da diese über entsprechende Methoden verfügen um diese Konvertierungen durchzuführen.

-- Die Libraries use IEEE.STD_LOGIC_TEXTIO.ALL; und use STD.TEXTIO.ALL benötigen wir um aus Dateien zu lesen
-- und auch um in Dateien zu schreiben. sie bieten uns also Methoden um Dateioperationen auszuführen.
-- Diese benötigen wir, weil wir aus der Datei 'memory.dat' in das Register schreiben wollen und
-- aus dem Register in die Datei dum.dat

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.STD_LOGIC_TEXTIO.ALL;

library STD;
use STD.TEXTIO.ALL;

-- Die schnittstellen waren vorgegeben und müssen daher hier nicht explizit kommentiert werden!

entity Memory is
port(
  clk: in std_logic;
  init: in std_logic;
  dump: in std_logic;
  reset: in std_logic;
  re: in std_logic;
  we: in std_logic;
  addr: in std_logic_vector(7 downto 0);
  data_in: in std_logic_vector(7 downto 0);
  output: out std_logic_vector(7 downto 0)
);
end memory;

architecture memory_impl of memory is

-- Komponenten

-- Typendefinitonen
```

```
47  -- Wir erstellen uns einen neuen Datentypen, der aus einem Array aus 8-Bit Wort
    besteht.
48  -- Bei 80Byte Nutzdaten und 8-Bit Worten [1Byte = 8 Bit] benötigen wir
49  -- einen 80x1Byte großen Array. Die Addresszuordnung wird über die Arrayposition
    umgesetzt.
50  -- Zusätzlich haben wir ein Arrayelement, Element 80, dass für falsche Eingabe dient!
51  type storage is array (0 to 80) of std_logic_vector(7 downto 0);
52
53  -- Signale
54  -- Wir erstellen unser 'Register' vom Type storage. Initial werden alle
    Registerelemente mit einem
55  -- leeren Bitvektor gefüllt. Nur Element 80 im Array wird vordefiniert für falsche
    Adresswerte!
56  signal reg : storage := (80 => "XXXXXXXX", others => "00000000");
57
58  signal bufferOut : std_logic_vector(7 downto 0) := (others => '0');
59
60  begin
61  -- Der Prozess execute reagiert auf die steigende Taktflanke (clk)
62  -- und schreibt die internen Signale entsprechend der Eingänge
63  -- re und we in das Register oder legt den Wert am Ausgang an.
64  -- Zusätzich reagiert er auf die Eingänge init und dump und liest entweder
65  -- aus der Datei memory.dat in den Array oder gibt den Inhalt des
66  -- Arrays in der Datei dump.dat aus
67  -- Ein synchroner Reset, der auf den Eingang 'reset' reagiert ist ebenfalls in diesem
68  -- Prozess umgesetzt.
69  -- Wir haben anhand der IF-Klauseln eine Fallunterscheidung eingeführt, sodass nur
    gültige
70  -- Eingabgen an den Eingängen auch zu einem Ergebnis am Ausgang führen können,
71  -- in allen Fällen die wir nicht anhand der fallunterscheidung betrachten geben wir
72  -- am Ausgang 'output' einen Bitvektor aus, der mit 'X' gefüllt ist, um einen
73  -- Fehler zu signalisieren
74  -- Hinweis: Auch in den Lese- und Schreibphasen in und aus memory.dat und dump.dat
75  -- liegt, wie in der Aufgabenstellung gefordert ein Bitvektor aus 'X' an
76  -- output an.
77  execute: process (clk)
78  -- Deklaration der Hilfsvariablen zum Einlesen und Schreiben der Dateien
79  -- memory.dat und dump.dat
80  file iofile : text;
81  variable ioline : line;
82  variable ioaddr : integer;
83  variable iodata : std_logic_vector(7 downto 0);
84  variable iotmp : std_logic_vector(7 downto 0);
85
86  -- Wir erstellen die lokale Variable 'iaddr', in den wir den 8 Bit Eingangsvektor
    addr als Integer
87  -- abspeichern wollen.
88  variable iaddr : integer range 0 to 255 := 0;
89
90
91  begin
92    -- Wie in der Aufgabenstellung gefordert reagieren wir nur auf die Steigende
      Taktflanke von clk
93    -- wir bauen also eine synchrone Schaltung!
94    if rising_edge(clk) then
95      -- Routine, die den Bitvektor addr in einen Integer wandelt,
96      -- damit er für die Adressierung anhand der Array Position genutzt werden kann
```

```
97        -- Dabei werden nur Interger Werte angenommen, die kleiner als 80 sind
98        -- Somit werden OutofBounds Exception verhindert!
99        iaddr := to_integer(unsigned(addr));
100       if(iaddr >= 80) then
101         iaddr := 80;
102         report "Array Index out of Bounds: " & integer'image(iaddr);
103       end if;
104       -- Wenn am Eingang 'init' eine '1' anliegt, soll der Inhalt der Datei memory.dat
          in das register
105       -- geladen werden. Die Fallunterscheidung verbietet eine '1' an einem anderen
          Eingang.
106       if (init = '1' AND dump = '0' AND reset = '0' AND re = '0' AND we = '0') then
107         -- Wir öffnen die Datei 'memory.dat' im rademode und lesen Sie Zeilenweise ein,
108         -- die ersten 8 Bits jeder Zeile konvertieren wir on the fly in einen
            Integerwert,
109         -- der und die Speicherposition im Array angibt und die zweiten 8 Bits
110         -- sind dann die Daten, die wir in die betreffende Speicherzelle schreiben.
111         -- Diesen Vorgang wiederholen wir so lange, bis wir das Ende der Datei
            erreicht haben.
112         file_open(iofile,"memory.dat",read_mode);
113           while not endfile(iofile) loop
114             readline(iofile,ioline);
115             read(ioline,iotmp);
116             ioaddr := to_integer(unsigned(iotmp));
117             read(ioline,iotmp);
118             iodata := iotmp;
119             reg(ioaddr) <= iodata;
120           end loop;
121         -- Wenn wir alles eingelesen haben müssen wir die Datei wieder schließen.
122         file_close(iofile);
123         -- Nach Abschluss des Einlesevorgangsgeben legen wir einen Bitvektor auf den
            Ausgang. Dies müssen wir tun,
124         -- da wir in Xilinx normalerweise keine Dateieingaben und Ausgaben schreiben
            können, da wir ja Hardware bauen.
125         -- Da die Dateieingabe und Ausgabe also nicht als Signale gelten würden wir
            die Eingänge init und dump nicht
126         -- benutzen und sie würden bei der Synthese wegoptimiert. Daher ein 8 Bit
            Vektor bestehend aus 'X' am Ausgang.
127         bufferOut <= "XXXXXXXX";
128       -- Wenn am Eingang 'dump' eine '1' anliegt, soll der Inhalt des Registers in die
          Datei dump.dat geschrieben
129       -- werden. Die Fallunterscheidung verbietet eine '1' an einem anderen Eingang.
130       elsif(dump = '1' AND init = '0' AND reset = '0' AND re = '0' AND we = '0') then
131         -- Wir öffnen die Datei 'dump.dat' im writemode und schreiben den Inhalt
            unseres Registers
132         -- Zeilenweise in die Datei. Dazu verwenden wir eine for-Schleife die bei null
            beginnt zu
133         -- zählen und verwenden diesen counter als Adresse, da wir das Array
            sequentiell durchlaufen.
134         -- Dabei geben wir durch die Typkonvertierung immer einen vollen 8 Bit Vektor
            als Adresse
135         -- in die datei ein.
136         -- Den Inhalt der betreffenden Speicherzelle an der Counter-Position des Arrays
137         -- schreiben wir dann in die selbe Zeile. Wir schreiben immer den gesamten
138         -- Inhalt des Arrays in die Datei.
139         -- Diesen Vorgang wiederholen wir so lange, bis wir das Ende des Arrays
            erreicht haben.
```

```
140         file_open(iofile,"dump.dat",write_mode);
141           for i in 0 to reg'length-1 loop
142             iotmp := std_logic_vector(to_unsigned(i, iotmp'length));
143             write(ioline,iotmp);
144             iotmp := reg(i);
145             write(ioline,iotmp);
146             writeline(iofile,ioline);
147           end loop;
148         -- Wenn wir alles geschrieben haben müssen wir die Datei wieder schließen.
149         file_close(iofile);
150         -- Nach Abschluss des Schreibevorgangs legen wir einen Bitvektor auf den
            Ausgang. Dies müssen wir tun,
151         -- da wir in Xilinx normalerweise keine Dateieingaben und Ausgaben schreiben
            können, da wir ja Hardware bauen.
152         -- Da die Dateieingabe und Ausgabe also nicht als Signale gelten würden wir
            die Eingänge init und dump nicht
153         -- benutzen und sie würden bei der Synthese wegoptimiert. Daher ein 8 Bit
            Vektor bestehend aus 'X' am Ausgang.
154         bufferOut <= "XXXXXXXX";
155       -- Wenn am Eingang re eine '1' anliegt, sollen die Daten, die in der Speicherzelle
156       -- "addr" am Ausgang "output" angelegt werden!
157       -- Die Fallunterscheidung verbietet eine '1' an einem anderen Eingang.
158       elsif(re = '1' AND dump = '0' AND init = '0' AND reset = '0' AND we = '0') then
159         bufferOut <= reg(iaddr);
160       -- Wenn am Eingang "we" eine '1' anliegt, sollen die Daten, die am Eingang
          "data_in"
161       -- anliegen in die Speicherzelle geschrieben werden, die durch "addr" definiert
          ist.
162       -- Die Fallunterscheidung verbietet eine '1' an einem anderen Eingang.
163       elsif(we = '1' AND re = '0' AND dump = '0' AND init = '0' AND reset = '0') then
164         if(iaddr = 80) then
165           bufferOut <= "XXXXXXXX";
166         else
167           reg(iaddr) <= data_in;
168         end if;
169       -- Wenn der Reseteingang aktiviert ist, werden alle Einträge des Registers auf 0
          zurückgesetzt.
170       -- Dabei wird das Array sequentiell durchlaufen.
171       -- Die Fallunterscheidung verbietet eine '1' an einem anderen Eingang.
172       elsif(reset = '1' AND we = '0' AND re = '0' AND dump = '0' AND init = '0') then
173         reg <= (80 => "XXXXXXXX", others => "00000000");
174         bufferOut <= "00000000";
175       -- Wenn keiner der oben genannten Fälle abgefangen wurde, muss es sich um eine
          fehlerhafte Belegung der
176       -- Eingänge handeln. In diesem Fall verändern wir den Inhalt des Registers nicht
          und geben nur einen
177       --  Bitvektor gefüllt mit 'X' an den Ausgang.
178       elsif(reset = '0' AND we = '0' AND re = '0' AND dump = '0' AND init = '0') then
179         bufferOut <= bufferOut;
180       else
181         bufferOut <= "XXXXXXXX";
182       end if;
183     end if;
184   end process;
185
186     output <= bufferOut;
187
```

```
188   end memory_impl;
```

```
1  ------------------------------------------------------------------------------
   -------------
2  -- Rechnerarchitektur und Eingebettete Systeme
3  -- Uebungszettel 2 - Aufgabe 1: Memory 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 Speichers pruefen:
8  --
9  -- Zuordnung von Testfall zu Aufgabenstellung:
10 -- 1. "Der Speicher soll 80 Byte speichern können. Diese Bytes sollen gelesen und
   geschrieben
11 --    werden können."
12 --    Geprueft durch Testfall 2.
13 --
14 -- 2. "Der Prozess des Schreibens oder Lesens soll bei einer steigenden Flanke der
   Clock clk
15 --    begonnen werden."
16 --    Geprueft durch Testfall 6.
17 --
18 -- 3. "Nach insgesamt einem Taktschritt soll die Operation abgeschlossen sein."
19 --    Geprueft durch Testfall 2.
20 --
21 -- 4. "Liegt am Signal reset eine 1, so wird der gesamte Speicher gelöscht und alle
   Werte werden
22 --    auf 0 gesetzt."
23 --    Geprueft durch Testfall 1. Weiterhin wird reset durch weitere Testfaelle
   verwendet und somit
24 --    geprueft.
25 --
26 -- 5. "Die Bits re (read enable) und we (write enable) geben an, ob gelesen oder
   geschrieben werden
27 --    soll. Für Lesen und Schreiben steht die betroffene Adresse des Speichers in
   addr. Soll
28 --    geschrieben werden, so steht der zu schreibende Wert in data_in. Soll gelesen
   werden, so wird
29 --    der entsprechendeWert in output ausgegeben..."
30 --    Geprueft durch Testfall 2. Weiterhin werden diese Bedingungen als
   Vorraussetzung fuer die
31 --    Funktion folgender Testfaelle benoetigt.
32 --
33 -- 8. "... und bis zum nächsten Lesebefehl gehalten."
34 --    Geprueft durch Testfall 6.
35 --
36 -- 9. "Für ein einfaches Debugging wird der Speicher mit der Möglichkeit
   ausgestattet, seinen
37 --    Inhalt aus einer Datei mit dem Namen memory.dat einzulesen. Der
   Einlese-Vorgang soll dann
38 --    beginnen, wenn der Eingang init gesetzt ist. Dieser Möglichkeit ist
   insbesondere für den
39 --    Sorter von Bedeutung, da man diesen anderweitig nicht mit Daten füttern könnte."
40 --    Geprueft durch CachedMemory und dessen Testbench.
41 --
42 -- 10. "Entsprechend steuert das Signal dump die Ausgabe des Speichers in die Datei
   dump.dat."
```

```
43  --    Geprueft durch CachedMemory und dessen Testbench.
44  --
45  -- 11. "Im Fehlerfall soll XXXXXXXX am Ausgang anliegen."
46  --    Geprueft durch Testfall 5.
47  --
48  -- Beschreibung der einzelnen Testfaelle:
49  -- Testfall 1:
50  --  - Pueft den Zustand nach Init und Reset (Output jeder Speicherzelle "00000000").
51  --  - Prueft das Lesen ("re") des Wertes "00000000".
52  -- Testfall 2:
53  --  - Prueft das Schreiben ("we") in jede Speicherzelle.
54  --  - Prueft das "output" seinen Wert beim Schreiben nicht aendert.
55  --  - Prueft das Lesen ("re") jeder Speicherzelle.
56  --  - Prueft das "output" beim Lesen den gelesenen Wert nicht beeinflusst.
57  --  - Prueft das jeder geschriebene Wert der richtigen Speicherzelle zugeordnet wird.
58  --  - Prueft das jeder Schreib- oder Lesevorgang nach einem Takt abgeschlossen ist.
59  --  - Prueft das Wirken der Signale "addr", "data_in", "we", "re" und "output"
60  --    (z.B. "addr" bestimmt die zu schreibende/lesende Adresse").
61  --  - Prueft das mindestens 80 Byte gespeichert werden koennen (Adresse "00000000" -
    "01001111").
62  -- Testfall 3:
63  --  - Prueft das jedes Bit jeder Speicherzelle mit 0 und 1 beschrieben werden kann.
64  --  - Prueft das nebeneinanderer liegende Bit einer Speicherzelle nicht
    ueberschieben werden.
65  -- Testfall 4:
66  --  - Prueft das eine Speicherzelle jeden moeglichen 8-Bit-Wert annehmen kann.
67  --  - Es wird nur drei Speicherzellen geprüft: 1. Speicherzelle, n. Speicherzelle
    und eine
68  --    beliebige (2.) dazwischen.
69  -- Testfall 5:
70  --  - Prueft das maximal an Adresse n-1 -also 80 Speicherzellen- geschrieben
71  --    oder gelesen werden kann. Ansonsten soll "XXXXXXXX" an "output" anliegen.
72  --  - Prueft das danach wieder geschrieben und gelesen werden kann
73  -- Testfall 6:
74  --  - Prueft das Lese- und Schreiboperationen nur bei steigender Flanke uebernommen
    wird.
75  --  - Prueft das ein Abbrechen des Lesevorgangs "output" nicht ueberschreibt und
    "output"
76  --    bis zum naechsten Lesebefehl gehalten wird (ein Takt ohne Lesebefehl)
77  ------------------------------------------------------------------------------
    -------------
```