```
1  ----------------------------------------------------------------------------------
2  --Aufgabe 3 konnten wir nicht rechtzeitig in eine funktionierende Form bringen.
3  --Unsere Testbench demonstriert in der Konsole recht gut die Idee unseres
   Lösungsansatzes,
4  --zeigt allerdings die zahlreichen Fehler, die beim Schreiben in den Speicher
   auftreten.
5  -- Company:
6  -- Engineer:
7  --
8  -- Create Date:    20:24:49 11/14/2015
9  -- Design Name:
10 -- Module Name:    Sorter - Behavioral
11
12 library IEEE;
13 use IEEE.STD_LOGIC_1164.ALL;
14 use ieee.numeric_std.ALL;
15 use ieee.std_logic_unsigned.ALL;
16
17
18 entity Sorter is
19     Port ( clk : in  STD_LOGIC;
20            addr_start : in  STD_LOGIC_VECTOR (7 downto 0);
21            addr_end : in  STD_LOGIC_VECTOR (7 downto 0);
22            start : in  STD_LOGIC;
23            done : out  STD_LOGIC);
24 end Sorter;
25
26 architecture Behavioral of Sorter is
27
28
29   component CachedMemory
30    Port (
31     clk     : in  STD_LOGIC;
32     init    : in  STD_LOGIC;
33     dump    : in  STD_LOGIC;
34     reset   : in  STD_LOGIC;
35     re      : in  STD_LOGIC;
36     we      : in  STD_LOGIC;
37     addr    : in  STD_LOGIC_VECTOR (7 downto 0);
38     data_in : in  STD_LOGIC_VECTOR (7 downto 0);
39     output  : out  STD_LOGIC_VECTOR (7 downto 0);
40     ack     : out  STD_LOGIC
41    );
42    end component;
43
44   signal mem_clk      : std_logic := '0';
45   signal mem_init     : std_logic := '0';
46   signal mem_dump     : std_logic := '0';
47   signal mem_reset    : std_logic := '0';
48   signal mem_re       : std_logic := '0';
49   signal mem_we       : std_logic := '0';
50   signal mem_addr     : std_logic_vector(7 downto 0) := (others => '0');
51   signal mem_data_in  : std_logic_vector(7 downto 0) := (others => '0');
52   signal mem_output   : std_logic_vector(7 downto 0);
53   signal mem_ack      : std_logic := '0';
54
55 begin
```

```
56     mem: CachedMemory PORT MAP (
57       clk     => clk,
58       init    => mem_init,
59       dump    => mem_dump,
60       reset   => mem_reset,
61       re      => mem_re,
62       we      => mem_we,
63       addr    => mem_addr,
64       data_in => mem_data_in,
65       output  => mem_output,
66       ack     => mem_ack);
67
68   execute: process (clk)
69     --Verzögert Sortierung, damit init den Speicher füllen kann.
70     --Erinnerung: Muss bei dump wieder auch Ursprungswert gestellt werden!
71     variable initDelay : Integer := 500;
72
73     variable isRunning   : STD_LOGIC                  := '0';--Gibt an, ob
       Sortierungg läuft.
74     variable pointer     : std_logic_vector(7 downto 0):= addr_start;
       --Speicheradresse für Vergleiche im Speicher.
75
76     variable currentValue: std_logic_vector(7 downto 0);--Aktueller
       zwischengespeicherter Vergleichswert
77     variable nextValue   : std_logic_vector(7 downto 0);--Nächster
       zwischengespeicherter Vergleichswert (zwischengespeichert)
78     variable currentValueValid: STD_LOGIC             := '0';--Gibt an, ob
       currentValue aktualisiert werden muss.
79     variable nextValueValid: STD_LOGIC                := '0';--Gibt an, ob nextValue
       aktualisiert werden muss.
80
81     variable getOutput: STD_LOGIC                     := '0';--Gibt an, ob der
       Output des Speichers bereit zum auslesen ist.
82     variable waitForOutput: STD_LOGIC                 := '0';--Gibt an, ob auf den
       Output des Speichers überhaupt zu warten ist.
83     variable amountCorrectLastDigits: Integer         := 0; --Anzahl korrekter
       Adressen am Ende des Adressraums. Erhöht sich nach und nach.
84     variable sortAgain: STD_LOGIC                     := '0';--Gibt an, ob im
       aktuellen Sortierdurchlauf zwei Werte getauscht wurden.
85     variable replaceCurrentValue: STD_LOGIC           := '0';--Gibt an, ob
       currentValue einen Wert enthält, der noch in den Speicher geschrieben werden muss.
86
87     variable saveValue    : std_logic_vector(7 downto 0);--Gibt den zu rettenden Wert
       (ehemals in currentValue) an.
88     variable saveAddress   : std_logic_vector(7 downto 0);--Gibt die Adresse des zu
       rettenden Wertes an.
89     variable saveLastValuePreviousSort : std_logic    := '0';--Gibt an, ob
       currentValue am Ende des letztes Sortierdurchlaufes
90                                                          --*nicht* in den
                                                            Speicher geschrieben
                                                            konnte und daher
                                                            anschließend
91                                                          --noch gespeichert
                                                            werden muss.
92
93     variable internalDone : std_logic                 := '0'; --zur Verzögerung der
       Umstellung von done
```

```
94
95  begin
96   if rising_edge(clk)
97   then
98
99     if(start = '1' AND isRunning = '0')
100    then
101       --Initialisiere zahlreiche Werte.
102       mem_dump <= '0';
103       pointer := addr_start;
104       isRunning := '1';
105       initDelay := 500;
106       done <= '0';
107       saveLastValuePreviousSort := '0';
108       internalDone := '0';
109    end if;
110
111    if(isRunning = '1')
112    then
113       --initDelay wird genutzt, um dem Speicher genug Zeit zu geben, init auszuführen.
114       if (initDelay = 500)
115       then
116          report "Habe mem_init auf 1 gesetzt und warte jetzt einen Takt.";
117          mem_init <= '1';
118       end if;
119
120       if (initDelay = 499)
121       then
122          report "Habe mem_init wieder auf 0 gesetzt.";
123          report "Warte noch ein paar hundert Zyklen, um init Zeit zu lassen.";
124          mem_init <= '0';
125       end if;
126
127       --delay runerzählen.
128       if (initDelay > 0)
129       then
130          initDelay := initDelay-1;
131       end if;
132
133       --Nach Verzögerung:
134       if(isRunning ='1' AND initDelay = 0)
135       then
136
137          --Wir laden, wenn nötig, den aktuellen Wert aus dem Speicher.
138          --currentValue wird nur zu Beginn eines Suchdurchlaufes abgerufen,
139          --da nextValue genügt, um currentValue für den jeweils nächsten Schritt
140          --festzulegen.
141          if(currentValueValid = '0')
142          then
143             if (getOutput = '0')
144             then
145                mem_addr <= pointer;
146                mem_we <= '0';
147                mem_re   <= '1';
148                getOutput := '1';
149
150             else
```

```
151                  if (getOutput = '1' AND mem_ack = '1')
152                  then
153                     --Speicher hat ack auf 1 gesetzt, daher ist der Wert nun abrufbar.
154                     report "currentValue: " &integer'image(to_integer(unsigned(
                        mem_output)));
155                     currentValue := mem_output;
156                     currentValueValid := '1';
157                     getOutput := '0';
158                  end if;
159               end if;
160            end if;
161
162            --Ähnlich wie für currentValue wird der benötigte Wert aus dem Speicher
               geladen.
163            if(currentValueValid = '1' AND nextValueValid = '0')
164            then
165               if (getOutput = '0')
166               then
167                  mem_addr <= std_logic_vector(unsigned(pointer) + 1);
168                  mem_we <= '0';
169                  mem_re   <= '1';
170                  getOutput := '1';
171
172               else
173                  if (getOutput = '1' AND mem_ack = '1')
174                  then
175                     report "nextValue: " &integer'image(to_integer(unsigned(mem_output
                        )));
176                     nextValue := mem_output;
177                     nextValueValid := '1';
178                     getOutput := '0';
179                  end if;
180               end if;
181            end if;
182
183            --Wenn im vorherigen Durchlauf als letzter Schritt eine Sortierung statt
               fand,
184            --wird der 'gerettete' letzte Wert nun noch kurz gespeichert.
185            --Dies ist vonnöten, da in einem solchen Fall 2 writes im gleichen Clock
               cycle
186            --ausgeführt werden würden. Das wird so vermieden.
187            if(saveLastValuePreviousSort='1' AND currentValueValid = '1' AND
               nextValueValid = '1')
188            then
189               report "Saving last value.";
190               report "#Write " &integer'image(to_integer(unsigned(saveValue))) & " to
                  address" &integer'image(to_integer(unsigned(saveAddress)));
191               mem_addr <= saveAddress;
192               mem_data_in<= saveValue;
193               mem_re <= '0';
194               mem_we <= '1';
195               saveLastValuePreviousSort:='0';
196               initDelay := 100;
197            end if;
198
199            --Wenn wir 'intern* fertig sind und kein Wert mehr zu retten ist, können
               wir den Wert dumpen.
```

```
200             if (internalDone = '1' AND saveLastValuePreviousSort='0')
201             then
202                report "Set done to true.";
203                initDelay := 5;
204                mem_re <= '0';
205                mem_we <= '0';
206                mem_reset <= '0';
207                mem_init <= '0';
208                mem_dump <= '1';
209                done <= '1';
210                isRunning := '0';
211             end if;
212
213
214             --Im folgenden Codeblock wird festgestellt, was sortiert wird.
215             --Allerdings wird bei jedem Durchlauf nur 1 Schreibvorgang durchgeführt!
216             --Dies spart Zeit, da der Speicher 1 Clock cycle zum speichern benötigt.
217             --
218             --1. Fall
219             --Wenn der aktuelle Wert größer ist als der nächste Wert, wird der aktuelle
                Wert
220             --durch den nächsten ersetzt.
221             --currentValue wird nicht verändert. Somit hat der Speicher zu diesem
                Zeitpunkt
222             --einen doppelten Eintrag! Der 2. Fall oder das Erreichen des Endes des
                Adressraumes
223             --behebt diese Inkonsistenz.
224             --
225             --2. Fall
226             --Wenn die beiden Werte in der korrekten Reihenfolge sind, wird die
                aktuelle Adresse
227             --mit dem zwischengespeicherten aktuellen Wert überschrieben, wenn der 1.
                Fall unmittel-
228             --bar zuvor auftrat.
229             if(currentValueValid = '1' AND nextValueValid = '1' AND
                saveLastValuePreviousSort = '0' AND internalDone = '0')
230             then
231                --In jedem Fall muss der nächste Wert gleich neu geladen werden.
232                nextValueValid := '0';
233                report "pointer: " & integer'image(to_integer(unsigned(pointer)));
234                report "currentValue: " &integer'image(to_integer(unsigned(currentValue
                   )))& ", nextValue: " &integer'image(to_integer(unsigned(nextValue)));
235                mem_re <= '0'; --Auf keinen Fall etwas lesen!
236
237                if (currentValue>nextValue)
238                then  --1. Fall
239                --Werte tauschen
240                   report "#Tausche currentValue und nextValue, weil nextValue kleiner
                      war.";
241                   report "#Write " &integer'image(to_integer(unsigned(nextValue)))& "
                      (next) to adress" &integer'image(to_integer(unsigned(pointer)));
242
243                   initDelay := 100;
244                   mem_addr <= pointer;
245                   mem_data_in <= nextValue;
246                   mem_we <= '1';
247                   sortAgain := '1';
```

```
248                  replaceCurrentValue := '1';
249                  --currentValue bleibt gleich, da es nach vorne verschoben wurde!
250                  --Daher wird currentValue kein neuer Wert zugewiesen.
251
252               else --2. Fall
253                  --Werte bereits in richtiger Reihefolge
254                  report "#Die Werte waren bereits in korrekter Reihenfolge.";
255                  report "#Write " &integer'image(to_integer(unsigned(currentValue)))
256                         & " (curr) to adress " &integer'image(to_integer(unsigned(
                           pointer)))
257                         & " (just in case.)";
258                  initDelay := 100;
259                  replaceCurrentValue := '0';
260                  mem_addr <= pointer;
261                  mem_data_in <= currentValue;
262                  mem_we <= '1';
263                  mem_re <= '0';
264
265                  currentValue := nextValue;
266
267               end if;
268
269               --Pointer für nächsten Schritt vorbereiten.
270               pointer := std_logic_vector(unsigned(pointer)+1 );
271
272               --Wir überprüfen, ob der Pointer den Adressraum verlassen würde.
273               --Wir berücksichtigen, dass mit jedem Durchlauf des Bubble Sort der
                  Suchlauf
274               --eine Adresse früher beendet werden kann, da pro Durchlauf ein hoher Wert
275               --das Ende des Adressraums füllt-
276               if (to_integer(unsigned(pointer) +1 ) > to_integer(unsigned(addr_end))  -
                   amountCorrectLastDigits)
277               then
278                  report "Ende des Sortierbereiches erreicht.";
279                  --Wir sparen uns einen Vergleich pro Suchdurchlauf.
280                  amountCorrectLastDigits := amountCorrectLastDigits +1;
281
282                  --Wenn der Sortiervorgang fertig ist, ist currentValue für die nächste
283                  --Adresse noch nicht gespeichert worden, wenn replaceCurrentValue
                     wahr ist.
284                  --Das muss allerdings geschehen, bevor weitersortiert werden kann, #
285                  --da der Speicher sonst falsch ist.
286                  if (replaceCurrentValue = '1')
287                  then
288                     report "Scheduled to save last value edit.";
289                     replaceCurrentValue := '0';
290                     saveLastValuePreviousSort := '1';
291                     saveAddress := std_logic_vector(unsigned(pointer));
292                     saveValue := std_logic_vector(unsigned(currentValue));
293                  end if;
294
295                  --Für erneute Sortierung vorbereiten:
296                  if (sortAgain = '1')
297                  then
298                     report "Starte Sortiervorgang neu.";
299                     sortAgain := '0';
300                     currentValueValid:= '0';
```

```
301                     nextValueValid := '0';
302
303                     report "--------------";
304                     pointer := addr_start; --Wichtig!
305
306                  --FERTIG SORTIERT!
307                  else
308                     report "Heureka! Sortierung ist fertig.";
309                     nextValueValid := '1';
310                     currentValueValid := '1';
311
312                     internalDone := '1';
313                  end if;
314               end if;
315            end if;
316         end if;
317      end if;
318   end if;
319  end process;
320  end Behavioral;
321
322
```

```
1  -- Diese Test ist eine einfache Möglichkeit um zu überprüfen, ob die eingabe
   memory.dat im vergleich zu
2  -- dump.dat sortiert wurde. Die Überprüfung muss händisch durchgeführt werden, ist
   aber auch relativ einfach
3  -- da dass erste eleemnt von memory.dat dass größte ist und an der 32. Stelle in
   dujmp.dat stehen muss, wenn die
4  -- Sortierung erfolgreich war! Ein testdurchlauf ist genügend, auf Grenzwerte wird
   hier nicht getestet,
5  -- da dies die Leisutngsmöglichkeiten des Testgerätes überschreitet!
```

```
0000000011111111
0000000100000100
0000001000000101
0000001100000110
0000010000000111
0000010100001000
0000011000001001
0000011100001010
0000100000001011
0000100100001100
0000101000001101
0000101100001110
0000110000001111
0000110100010000
0000111000010001
0000111100010010
0001000000010111
0001000100011000
0001001000011001
0001001100011010
0001010000011011
0001010100011100
0001011000011101
0001011100011110
0001100000011111
0001100100100000
0001101000100001
0001101100100010
0001110000100011
0001110100100100
0001111000100101
0001111100100110
0010000011111111
0010000111111111
0010001011111111
0010001111111111
0010010011111111
0010010111111111
0010011011111111
0010011111111111
0010100011111111
0010100111111111
0010101011111111
0010101111111111
0010110011111111
0010110111111111
0010111011111111
0010111111111111
0011000000000000
0011000100000000
0011001000000000
0011001100000000
0011010000000000
0011010100000000
0011011000000000
0011011100000000
0011100000000000
```

```
0011100100000000
0011101000000000
0011101100000000
0011110000000000
0011110100000000
0011111000000000
0011111100000000
0100000000000000
0100000100000000
0100001000000000
0100001100000000
0100010000000000
0100010100000000
0100011000000000
0100011100000000
0100100000000000
0100100100000000
0100101000000000
0100101100000000
0100110000000000
0100110100000000
0100111000000000
```

```
0000000000000100
0000000100000101
0000001000000110
0000001100000111
0000010000001000
0000010100001001
0000011000001010
0000011100001011
0000100000001100
0000100100001101
0000101000001110
0000101100001111
0000110000010000
0000110100010001
0000111000010010
0000111100010111
0001000011111111
0001000100011000
0001001000011001
0001001100011010
0001010000011011
0001010100011100
0001011000011101
0001011100011110
0001100000011111
0001100100100000
0001101000100001
0001101100100010
0001110000100011
0001110100100100
0001111000100101
0001111100100110
0010000011111111
0010000111111111
0010001011111111
0010001111111111
0010010011111111
0010010111111111
0010011011111111
0010011111111111
0010100011111111
0010100111111111
0010101011111111
0010101111111111
0010110011111111
0010110111111111
0010111011111111
0010111111111111
0011000000000000
0011000100000000
0011001000000000
0011001100000000
0011010000000000
0011010100000000
0011011000000000
0011011100000000
0011100000000000
```

```
0011100100000000
0011101000000000
0011101100000000
0011110000000000
0011110100000000
0011111000000000
0011111100000000
0100000000000000
0100000100000000
0100001000000000
0100001100000000
0100010000000000
0100010100000000
0100011000000000
0100011100000000
0100100000000000
0100100100000000
0100101000000000
0100101100000000
0100110000000000
0100110100000000
0100111000000000
0100111100000000
01010000XXXXXXXX

```