```
1   ------------------------------------------------------------------------------
2   -- Hardware AMBA bus IP for image scaling
3   ------------------------------------------------------------------------------
4   -- Entity:      TFG_IMAGE_SCALING
5   -- File:        TFG_IMAGE_SCALING.vhd
6   -- Author:      Luis Miguel Gonzalez Berrocal
7   ------------------------------------------------------------------------------
8   -- VHDL Standard: VHDL'93
9   ------------------------------------------------------------------------------
10  -- Naming Conventions
11  -- active low signals:                  "*_n"
12  -- clock signals:                       "clk", "clk_div#", "clk_#x"
13  -- reset signals:                       "rst", "rst_n"
14  -- generics:                            "C_*"
15  -- user defined types:                  "*_TYPE"
16  -- state machine next state:            "*_ns"
17  -- state machine current state:         "*_cs"
18  -- combinatorial signals:               "*_com"
19  -- pipelined or register delay signals: "*_d#"
20  -- counter signals:                     "*_cnt*"
21  -- clock enable signals:                "*_ce"
22  -- internal version of output port:     "*_i"
23  -- device pins:                         "*_pin"
24  -- ports:                               "Names begin with Uppercase"
25  -- processes:                           "*_PROCESS"
26  -- component instantiations:            "<ENTITY_>I_<#|FUNC>"
27  -- registers:                           "*_REG"
28  ------------------------------------------------------------------------------
29
30  library ieee;
31  use ieee.std_logic_1164.all;
32  library grlib;
33  use grlib.amba.all;
34  use grlib.stdlib.all;
35  use grlib.devices.all;
36  library gaisler;
37  use gaisler.misc.all;
38  use gaisler.libdcom.all;
39  use gaisler.arith.all;
40  library techmap;
41  use techmap.gencomp.all;
42
43  entity TFG_IMAGE_SCALING is
44        generic (FABTECH          : integer := 0;
45                 MEMTECH          : integer := DEFMEMTECH;
46                 PINDEX           : integer := 0;
47                 PADDR            : integer := 0;
48                 PMASK            : integer := 16#fff#;
49                 HINDEX           : integer := 0;
50                 AHBACCSZ         : integer := 32;
51                 BURSTLEN         : integer range 2 to 8 := 8;
52                 AHB_SLV_HINDEX   : integer := 0;
53                 AHB_SLV_HADDR    : integer := 0;
54                 AHB_SLV_HMASK    : integer := 16#fff#;
55                 HIRQ             : integer := 10;
56
57                 INT_ADDRESS_WIDTH : integer := 11;     -- tamaño ram TFG_ROW_BUFFER
```

```
 58                SRC_ADDRESS_WIDTH : integer := 9;      -- tamaño src_ram
 59                DST_ADDRESS_WIDTH : integer := 9;      -- tamaño dst_ram
 60                ADDRESS_HEIGHT    : integer := 11);    -- tamaño contador de filas de
    imagen destino
 61       port (Rst_n : in  std_ulogic;
 62             Clk   : in  std_ulogic;
 63             Apbi  : in  apb_slv_in_type;
 64             Apbo  : out apb_slv_out_type;
 65             Ahbi  : in  ahb_mst_in_type;
 66             Ahbo  : out ahb_mst_out_type);
 67   end TFG_IMAGE_SCALING;
 68
 69   architecture Behavioral of TFG_IMAGE_SCALING is
 70
 71        -- TFG_APB_SLAVE_1
 72        signal irq_user, en, we : std_logic;
 73        signal di, do : std_logic_vector(31 downto 0);
 74        signal addr : std_logic_vector(7 downto 0);
 75
 76        -- TFG_AHB_MASTER_1
 77        signal transfer_type, size : std_logic_vector(2 downto 0);
 78        signal start, dwrite, transfer_end, transfer_active, data_valid : std_logic;
 79        signal addro, dataw, datar : std_logic_vector(31 downto 0);
 80        signal beat_counter : std_logic_vector(9 downto 0);
 81
 82        -- TFG_ROW_INTERPOLATION_1
 83        signal pixel_out, pixel_even, pixel_odd : std_logic_vector (7 downto 0);
 84        signal embedd_xofs : std_logic_vector (INT_ADDRESS_WIDTH - 1 downto 0);
 85        signal embedd_yofs : std_logic_vector (ADDRESS_HEIGHT - 1 downto 0);
 86        signal internal_address : std_logic_vector (INT_ADDRESS_WIDTH - 1 downto 0);
 87        signal image_scaling_state : std_logic_vector (3 downto 0);
 88        signal row_interpolation_state : std_logic_vector (2 downto 0);
 89
 90        -- register bank
 91        signal status_REG, control_REG, address_src_REG, address_dst_REG :
    std_logic_vector (31 downto 0);
 92        signal src_width_REG, src_height_REG, dst_width_REG, dst_height_REG :
    std_logic_vector (31 downto 0);
 93        signal shifted_scale_alpha_REG, shifted_scale_beta_REG : std_logic_vector (31
    downto 0);
 94
 95        -- ram
 96        type src_ram_TYPE is array (0 to 2**SRC_ADDRESS_WIDTH - 1) of std_logic_vector (
    31 downto 0);
 97        type dst_ram_TYPE is array (0 to 2**DST_ADDRESS_WIDTH - 1) of std_logic_vector (
    31 downto 0);
 98        signal ram_enable : std_logic;
 99
100        -- src ram
101        signal src_ram : src_ram_TYPE := (others => X"CAFECAFE");
102        signal write_enable_src : std_logic;
103        signal data_input_src, data_outputA, data_outputB : std_logic_vector (31 downto
    0);
104        signal write_addr : std_logic_vector (SRC_ADDRESS_WIDTH - 1 downto 0);
105        signal read_addr_outputA, read_addr_outputB : std_logic_vector (
    SRC_ADDRESS_WIDTH - 1 downto 0);
106
```

```
107         -- dst ram
108         signal dst_ram : dst_ram_TYPE := (others => X"CAFECAFE");
109         signal write_enable_dst : std_logic;
110         signal data_input_dst, data_output : std_logic_vector (31 downto 0);
111         signal read_write_addr : std_logic_vector (DST_ADDRESS_WIDTH - 1 downto 0);
112
113         -- state machine
114         type state_TYPE is (s0, s1, s2, s3, s4, s5, s6, s7, s8, s9, s10, s11, s12, s13,
    s14, s15);
115         signal current_state : state_TYPE := s0;
116
117         -- delay
118         signal beat_counter_d1 : std_logic_vector (9 downto 0);
119         signal address_index_d1 : std_logic_vector (1 downto 0);
120         signal internal_address_d1, internal_address_d2 : std_logic_vector (
    INT_ADDRESS_WIDTH - 1 downto 0);
121         signal internal_address_d3, internal_address_d4 : std_logic_vector (
    INT_ADDRESS_WIDTH - 1 downto 0);
122         signal internal_address_d5 : std_logic_vector (INT_ADDRESS_WIDTH - 1 downto 0);
123
124         -- counters and acumulators
125         signal inh : std_logic;
126         signal offset_counter : std_logic_vector (1 downto 0);
127         signal dst_offset : std_logic_vector (31 downto 0);
128         signal row_cnt : std_logic_vector (ADDRESS_HEIGHT - 1 downto 0);
129
130         -- mux
131         signal index_offset : std_logic_vector (1 downto 0);
132         signal sel : std_logic;
133
134         -- aux
135         signal control_REG_d1, start_transfer, write_last_word, write_last_word_d1 :
    std_logic;
136         signal byte0, byte1, byte2, byte3 : std_logic_vector (7 downto 0);
137         signal src_width_adjusted, dst_width_adjusted : std_logic_vector (9 downto 0);
138         signal row_address_read, row_address_read_even, row_address_read_odd :
    std_logic_vector (31 downto 0);
139         signal row_address_write : std_logic_vector (31 downto 0);
140         signal address_index : std_logic_vector (INT_ADDRESS_WIDTH - 1 downto 0);
141         signal last_word, compound_word, new_word, last_word_d1 : std_logic_vector (31
    downto 0);
142         signal src_index : integer;
143         signal zeros_internal_address : std_logic_vector (INT_ADDRESS_WIDTH - 1 - 2
    downto 0) := (others => '0');
144         signal zeros_read_addr_outputB : std_logic_vector (SRC_ADDRESS_WIDTH - 1 - 1
    downto 0) := (others => '0');
145         signal zeros_index_offset : std_logic_vector (INT_ADDRESS_WIDTH - 1 - 2 downto 0
    ) := (others => '0');
146         signal zeros_embedd_yofs : std_logic_vector (15 - ADDRESS_HEIGHT downto 0) := (
    others => '0');
147
148         component TFG_APB_SLAVE
149         generic (--FABTECH     : integer := 0;
150                  --MEMTECH     : integer := 0;
151                  NAHBIRQ     : integer := 32;
152                  PINDEX      : integer := 0;
153                  PADDR       : integer := 0;
```

```
154                 PMASK       : integer := 16#fff#;
155                 HIRQ        : integer := 10);
156         port (Rst_n    : in  std_ulogic;
157               Clk      : in  std_ulogic;
158               Apbi     : in  apb_slv_in_type;
159               Apbo     : out apb_slv_out_type;
160               Irq_user : in std_logic;
161               En       : out std_logic;
162               We       : out std_logic;
163               Di       : out std_logic_vector(31 downto 0);
164               Do       : in std_logic_vector(31 downto 0);
165               Addr     : out std_logic_vector(7 downto 0));
166         end component;
167
168         component TFG_AHB_MASTER
169         generic (HINDEX  : integer := 0;
170                  VENID   : integer := 1;
171                  DEVID   : integer := 0;
172                  VERSION : integer := 0);
173         port (Rst_n           : in  std_ulogic;
174               Clk             : in  std_ulogic;
175               Ahbi            : in ahb_mst_in_type;
176               Ahbo            : out ahb_mst_out_type;
177               Transfer_type   : in std_logic_vector(2 downto 0);
178               Size            : in std_logic_vector(2 downto 0);
179               Start           : in std_logic;
180               Addro           : in std_logic_vector(31 downto 0);
181               Dwrite          : in std_logic;
182               Dataw           : in std_logic_vector(31 downto 0);
183               Transfer_end    : out std_logic;
184               Transfer_active : out std_logic;
185               Datar           : out std_logic_vector(31 downto 0);
186               Data_valid      : out std_logic;
187               Beat_counter    : out std_logic_vector(9 downto 0));
188         end component;
189
190         component TFG_ROW_INTERPOLATION
191         generic (ADDRESS_WIDTH             : integer := 11;
192                  ADDRESS_HEIGHT            : integer := 11;
193                  IMSE_EXTRA_PRECISION_BITS : integer := 4;
194                  INTER_RESIZE_COEF_BITS    : integer := 11;
195                  SHIFT                     : integer := 22);
196         port (Clk                    : in std_logic;
197               Rst_n                  : in std_logic;
198               Pixel_even             : in std_logic_vector (7 downto 0);
199               Pixel_odd              : in std_logic_vector (7 downto 0);
200               Dst_width              : in std_logic_vector (31 downto 0);
201               Shifted_scale_alpha    : in std_logic_vector (31 downto 0);
202               Shifted_scale_beta     : in std_logic_vector (31 downto 0);
203               Row_cnt                : in std_logic_vector (ADDRESS_HEIGHT - 1 downto 0
      );
204               Image_scaling_state    : in std_logic_vector (3 downto 0);
205               Row_interpolation_state : out std_logic_vector (2 downto 0);
206               Embedd_xofs            : out std_logic_vector (ADDRESS_WIDTH - 1 downto 0
      );
207               Embedd_yofs            : out std_logic_vector (ADDRESS_HEIGHT - 1 downto
      0);
```

```
208             Internal_address        : out std_logic_vector (ADDRESS_WIDTH - 1 downto 0
    );
209             Pixel_out               : out std_logic_vector (7 downto 0));
210       end component;
211
212   begin
213
214       TFG_APB_SLAVE_1 : TFG_APB_SLAVE
215       generic map (--FABTECH =>
216                    --MEMTECH =>
217                    NAHBIRQ    => NAHBIRQ,
218                    PINDEX     => PINDEX,
219                    PADDR      => PADDR,
220                    PMASK      => PMASK,
221                    HIRQ       => HIRQ)
222       port map (Rst_n   => Rst_n,
223                 Clk     => Clk,
224                 Apbi    => Apbi,
225                 Apbo    => Apbo,
226                 Irq_user => irq_user,
227                 En      => en,
228                 We      => we,
229                 Di      => di,
230                 Do      => do,
231                 Addr    => addr);
232
233       TFG_AHB_MASTER_1 : TFG_AHB_MASTER
234       generic map (HINDEX  => HINDEX,
235                    VENID   => CONTRIB_CORE2,
236                    DEVID   => CONTRIB_CORE2,
237                    VERSION => 1)
238       port map (Rst_n   => Rst_n,
239                 Clk     => Clk,
240                 Ahbi    => Ahbi,
241                 Ahbo    => Ahbo,
242                 Transfer_type    => transfer_type,
243                 Size             => size,
244                 Start            => start,
245                 Addro            => addro,
246                 Dwrite           => dwrite,
247                 Dataw            => dataw,
248                 Transfer_end     => transfer_end,
249                 Transfer_active  => transfer_active,
250                 Datar            => datar,
251                 Data_valid       => data_valid,
252                 Beat_counter     => beat_counter);
253
254       TFG_ROW_INTERPOLATION_1 : TFG_ROW_INTERPOLATION
255       generic map (ADDRESS_WIDTH             => INT_ADDRESS_WIDTH,
256                    ADDRESS_HEIGHT            => ADDRESS_HEIGHT,
257                    IMSE_EXTRA_PRECISION_BITS => 4,
258                    INTER_RESIZE_COEF_BITS    => 11,
259                    SHIFT                     => 22)
260       port map (Clk                      => Clk,
261                 Rst_n                    => Rst_n,
262                 Pixel_even               => pixel_even,
263                 Pixel_odd                => pixel_odd,
```

```
264                   Dst_width                 => dst_width_REG,
265                   Shifted_scale_alpha       => shifted_scale_alpha_REG,
266                   Shifted_scale_beta        => shifted_scale_beta_REG,
267                   Row_cnt                   => row_cnt,
268                   Image_scaling_state       => image_scaling_state,
269                   Row_interpolation_state   => row_interpolation_state,
270                   Embedd_xofs               => embedd_xofs,
271                   Embedd_yofs               => embedd_yofs,
272                   Internal_address          => internal_address,
273                   Pixel_out                 => pixel_out);
274
275         registers_PROCESS : process (Clk, Rst_n)
276         begin
277               if (Rst_n = '0') then
278                     control_REG <= (others => '0');
279                     address_src_REG <= (others => '0');
280                     address_dst_REG <= (others => '0');
281                     src_width_REG <= (others => '0');
282                     src_height_REG <= (others => '0');
283                     dst_width_REG <= (others => '0');
284                     dst_height_REG <= (others => '0');
285                     shifted_scale_alpha_REG <= (others => '0');
286                     shifted_scale_beta_REG <= (others => '0');
287               elsif (Clk = '1' and Clk'event) then
288                     if (en = '1' and we = '1') then
289                           if addr (7 downto 0) = X"01" then
290                                 control_REG <= di (31 downto 0);
291                           end if;
292                           if addr (7 downto 0) = X"02" then
293                                 address_src_REG <= di (31 downto 0);
294                           end if;
295                           if addr (7 downto 0) = X"03" then
296                                 address_dst_REG <= di (31 downto 0);
297                           end if;
298                           if addr (7 downto 0) = X"04" then
299                                 src_width_REG <= di (31 downto 0);
300                           end if;
301                           if addr (7 downto 0) = X"05" then
302                                 src_height_REG <= di (31 downto 0);
303                           end if;
304                           if addr (7 downto 0) = X"06" then
305                                 dst_width_REG <= di (31 downto 0);
306                           end if;
307                           if addr (7 downto 0) = X"07" then
308                                 dst_height_REG <= di (31 downto 0);
309                           end if;
310                           if addr (7 downto 0) = X"08" then
311                                 shifted_scale_alpha_REG <= di (31 downto 0);
312                           end if;
313                           if addr (7 downto 0) = X"09" then
314                                 shifted_scale_beta_REG <= di (31 downto 0);
315                           end if;
316                     end if;
317               end if;
318         end process;
319
320         mux_apb_do_PROCESS: process (addr, status_REG, control_REG, address_src_REG,
```

```
      address_dst_REG, src_width_REG,
321                                     src_height_REG, dst_width_REG, dst_height_REG,
      shifted_scale_alpha_REG,
322                                     shifted_scale_beta_REG)
323         begin
324               case addr (7 downto 0) is
325                     when X"00" => do <= status_REG;
326                     when X"01" => do <= control_REG;
327                     when X"02" => do <= address_src_REG;
328                     when X"03" => do <= address_dst_REG;
329                     when X"04" => do <= src_width_REG;
330                     when X"05" => do <= src_height_REG;
331                     when X"06" => do <= dst_width_REG;
332                     when X"07" => do <= dst_height_REG;
333                     when X"08" => do <= shifted_scale_alpha_REG;
334                     when X"09" => do <= shifted_scale_beta_REG;
335                     when others => do <= X"CAFECAFE";
336               end case;
337         end process;
338
339         ram_enable <= '1';
340         src_ram_PROCESS : process (Clk)
341         begin
342               if (Clk'event and Clk = '1') then
343                     if (ram_enable = '1') then
344                           if (write_enable_src = '1') then
345                                 src_ram (conv_integer(write_addr)) <= data_input_src;
346                           end if;
347                           data_outputA <= src_ram (conv_integer(read_addr_outputA));
348                           data_outputB <= src_ram (conv_integer(read_addr_outputB));
349                     end if;
350               end if;
351         end process;
352
353         dst_ram_PROCESS : process (Clk)
354         begin
355               if (Clk'event and Clk = '1') then
356                     if (ram_enable = '1') then
357                           if (write_enable_dst = '1') then
358                                 dst_ram (conv_integer(read_write_addr)) <=
      data_input_dst;
359                           end if;
360                           data_output <= dst_ram (conv_integer(read_write_addr));
361                     end if;
362               end if;
363         end process;
364
365         signals_PROCESS : process (current_state, beat_counter, datar, data_valid,
      beat_counter_d1, data_output,
366                                     address_index, row_address_read_even,
      row_address_read_odd, row_address_write,
367                                     offset_counter, internal_address_d1,
      internal_address_d2, internal_address_d3,
368                                     internal_address_d4, internal_address_d5,
      zeros_read_addr_outputB, src_index)
369         begin
370               case current_state is
```

```
371                     when s0 =>
372                           transfer_type <= (others => '0');     -- tipo de transferencia
373                           size <= (others => '0');              -- tamaño de datos
374                           start <= '0';                         -- inicio de
    transferencia
375                           addro <= (others => '0');             -- direccion fila
    imagen fuente
376                           dwrite <= '0';                        -- lectura/escritura
    del maestro
377                           dataw <= (others => '0');             -- dato a escribir
378                           data_input_src <= (others => '0');    -- entrada src_ram
379                           write_enable_src <= '0';              -- enable de escritura
    de src_ram
380                           write_addr <= (others => '0');        -- direccion escritura
    de src_ram
381                           read_addr_outputA <= (others => '0'); -- direccion lectura A
    de src_ram
382                           read_addr_outputB <= (others => '0'); -- direccion lectura B
    de src_ram
383                           read_write_addr <= (others => '0');   -- direccion
    lectura/escritura de dst_ram
384                           write_enable_dst <= '0';              -- enable de escritura
    de dst_ram
385                           sel <= '0';                           -- multiplexor para
    index_offset
386                           write_last_word <= '0';               -- enable de escritura
    en registro last_word
387                           inh <= '1';                           -- inhibicion de
    cuenta/acumulacion
388                           write_last_word_d1 <= '0';            -- enable de escritura
    en registro last_word_d1
389                           status_REG <= (others => '0');        -- registro de
    funcionamiento de la maquina de estados
390                           image_scaling_state <= "0000";        -- numeracion de los
    estados de TFG_IMAGE_SCALING
391                     when s1 =>
392                           transfer_type <= (others => '0');
393                           size <= (others => '0');
394                           start <= '0';
395                           addro <= (others => '0');
396                           dwrite <= '0';
397                           dataw <= (others => '0');
398                           data_input_src <= (others => '0');
399                           write_enable_src <= '0';
400                           write_addr <= (others => '0');
401                           read_addr_outputA <= (others => '0');
402                           read_addr_outputB <= (others => '0');
403                           read_write_addr <= (others => '0');
404                           write_enable_dst <= '0';
405                           sel <= '0';
406                           write_last_word <= '0';
407                           inh <= '1';
408                           write_last_word_d1 <= '0';
409                           status_REG <= X"00000001";
410                           image_scaling_state <= "0001";
411                     when s2 =>
412                           transfer_type <= "001";
```

```
413                         size <= "010";
414                         start <= '1';
415                         addro <= row_address_read_even (31 downto 2) & "00";
416                         dwrite <= '0';
417                         dataw <= (others => '0');
418                         data_input_src <= datar;
419                         write_enable_src <= data_valid;
420                         write_addr <= beat_counter_d1 (SRC_ADDRESS_WIDTH - 1 downto 0);
421                         read_addr_outputA <= (others => '0');
422                         read_addr_outputB <= (others => '0');
423                         read_write_addr <= (others => '0');
424                         write_enable_dst <= '0';
425                         sel <= '0';
426                         write_last_word <= '0';
427                         inh <= '1';
428                         write_last_word_d1 <= '0';
429                         status_REG <= X"00000001";
430                         image_scaling_state <= "0010";
431                   when s3 =>
432                         transfer_type <= "001";
433                         size <= "010";
434                         start <= '1';
435                         addro <= row_address_read_even (31 downto 2) & "00";
436                         dwrite <= '0';
437                         dataw <= (others => '0');
438                         data_input_src <= datar;
439                         write_enable_src <= data_valid;
440                         write_addr <= beat_counter_d1 (SRC_ADDRESS_WIDTH - 1 downto 0);
441                         read_addr_outputA <= (others => '0');
442                         read_addr_outputB <= (others => '0');
443                         read_write_addr <= (others => '0');
444                         write_enable_dst <= '0';
445                         sel <= '0';
446                         write_last_word <= '0';
447                         inh <= '1';
448                         write_last_word_d1 <= '0';
449                         status_REG <= X"00000001";
450                         image_scaling_state <= "0011";
451                   when s4 =>
452                         transfer_type <= (others => '0');
453                         size <= (others => '0');
454                         start <= '0';
455                         addro <= (others => '0');
456                         dwrite <= '0';
457                         dataw <= (others => '0');
458                         data_input_src <= (others => '0');
459                         write_enable_src <= '0';
460                         write_addr <= (others => '0');
461                         read_addr_outputA <= address_index (SRC_ADDRESS_WIDTH - 1 +
    src_index downto 2);
462                         if (address_index (1 downto 0) = "11") then
463                               read_addr_outputB <= address_index (SRC_ADDRESS_WIDTH -
    1 + src_index downto 2) +
464
    (zeros_read_addr_outputB & '1');
465                         else
466                               read_addr_outputB <= address_index (SRC_ADDRESS_WIDTH -
```

```
      1 + src_index downto 2);
467                         end if;
468                         read_write_addr <= (others => '0');
469                         write_enable_dst <= '0';
470                         sel <= '0';
471                         write_last_word <= '0';
472                         inh <= '1';
473                         write_last_word_d1 <= '0';
474                         status_REG <= X"00000001";
475                         image_scaling_state <= "0100";
476                   when s5 =>
477                         transfer_type <= "001";
478                         size <= "010";
479                         start <= '1';
480                         addro <= row_address_read_odd (31 downto 2) & "00";
481                         dwrite <= '0';
482                         dataw <= (others => '0');
483                         data_input_src <= datar;
484                         write_enable_src <= data_valid;
485                         write_addr <= beat_counter_d1 (SRC_ADDRESS_WIDTH - 1 downto 0);
486                         read_addr_outputA <= (others => '0');
487                         read_addr_outputB <= (others => '0');
488                         read_write_addr <= (others => '0');
489                         write_enable_dst <= '0';
490                         sel <= '1';
491                         write_last_word <= '0';
492                         inh <= '1';
493                         write_last_word_d1 <= '0';
494                         status_REG <= X"00000001";
495                         image_scaling_state <= "0101";
496                   when s6 =>
497                         transfer_type <= "001";
498                         size <= "010";
499                         start <= '1';
500                         addro <= row_address_read_odd (31 downto 2) & "00";
501                         dwrite <= '0';
502                         dataw <= (others => '0');
503                         data_input_src <= datar;
504                         write_enable_src <= data_valid;
505                         write_addr <= beat_counter_d1 (SRC_ADDRESS_WIDTH - 1 downto 0);
506                         read_addr_outputA <= (others => '0');
507                         read_addr_outputB <= (others => '0');
508                         read_write_addr <= (others => '0');
509                         write_enable_dst <= '0';
510                         sel <= '1';
511                         write_last_word <= '0';
512                         inh <= '1';
513                         write_last_word_d1 <= '1';
514                         status_REG <= X"00000001";
515                         image_scaling_state <= "0110";
516                   when s7 =>
517                         transfer_type <= (others => '0');
518                         size <= (others => '0');
519                         start <= '0';
520                         addro <= (others => '0');
521                         dwrite <= '0';
522                         dataw <= (others => '0');
```

```
523                         data_input_src <= (others => '0');
524                         write_enable_src <= '0';
525                         write_addr <= (others => '0');
526                         read_addr_outputA <= address_index (SRC_ADDRESS_WIDTH - 1 +
    src_index downto 2);
527                         if (address_index (1 downto 0) = "11") then
528                               read_addr_outputB <= address_index (SRC_ADDRESS_WIDTH -
    1 + src_index downto 2) +
529
    (zeros_read_addr_outputB & '1');
530                         else
531                               read_addr_outputB <= address_index (SRC_ADDRESS_WIDTH -
    1 + src_index downto 2);
532                         end if;
533                         if (offset_counter = "00") then
534                               read_write_addr <= internal_address_d5 (
    DST_ADDRESS_WIDTH - 1 + 2 downto 2);
535                         elsif (offset_counter = "01") then
536                               read_write_addr <= internal_address_d4 (
    DST_ADDRESS_WIDTH - 1 + 2 downto 2);
537                         elsif (offset_counter = "10") then
538                               read_write_addr <= internal_address_d3 (
    DST_ADDRESS_WIDTH - 1 + 2 downto 2);
539                         elsif (offset_counter = "11") then
540                               read_write_addr <= internal_address_d2 (
    DST_ADDRESS_WIDTH - 1 + 2 downto 2);
541                         else
542                               read_write_addr <= (others => '0');
543                         end if;
544                         write_enable_dst <= '1';
545                         sel <= '1';
546                         write_last_word <= '1';
547                         inh <= '1';
548                         write_last_word_d1 <= '0';
549                         status_REG <= X"00000001";
550                         image_scaling_state <= "0111";
551                   when s8 =>
552                         transfer_type <= "001";
553                         size <= "010";
554                         start <= '1';
555                         addro <= row_address_write (31 downto 2) & "00";
556                         dwrite <= '1';
557                         dataw <= data_output;
558                         data_input_src <= (others => '0');
559                         write_enable_src <= '0';
560                         write_addr <= (others => '0');
561                         read_addr_outputA <= (others => '0');
562                         read_addr_outputB <= (others => '0');
563                         read_write_addr <= beat_counter (DST_ADDRESS_WIDTH - 1 downto
    0);
564                         write_enable_dst <= '0';
565                         sel <= '1';
566                         write_last_word <= '0';
567                         inh <= '1';
568                         write_last_word_d1 <= '0';
569                         status_REG <= X"00000001";
570                         image_scaling_state <= "1000";
```

```
571                     when s9 =>
572                           transfer_type <= "001";
573                           size <= "010";
574                           start <= '1';
575                           addro <= row_address_write (31 downto 2) & "00";
576                           dwrite <= '1';
577                           dataw <= data_output;
578                           data_input_src <= (others => '0');
579                           write_enable_src <= '0';
580                           write_addr <= (others => '0');
581                           read_addr_outputA <= (others => '0');
582                           read_addr_outputB <= (others => '0');
583                           read_write_addr <= beat_counter (DST_ADDRESS_WIDTH - 1 downto
    0);
584                           write_enable_dst <= '0';
585                           sel <= '1';
586                           write_last_word <= '0';
587                           inh <= '1';
588                           write_last_word_d1 <= '0';
589                           status_REG <= X"00000001";
590                           image_scaling_state <= "1001";
591                     when s10 =>
592                           if (data_valid = '0') then
593                                 transfer_type <= "001";
594                                 size <= "010";
595                                 start <= '1';
596                                 addro <= row_address_write (31 downto 2) & "00";
597                                 dwrite <= '1';
598                                 dataw <= data_output;
599                                 read_write_addr <= beat_counter (DST_ADDRESS_WIDTH - 1
    downto 0);
600                           else
601                                 transfer_type <= (others => '0');
602                                 size <= (others => '0');
603                                 start <= '0';
604                                 addro <= (others => '0');
605                                 dwrite <= '0';
606                                 dataw <= data_output;
607                                 read_write_addr <= (others => '0');
608                           end if;
609                           data_input_src <= (others => '0');
610                           write_enable_src <= '0';
611                           write_addr <= (others => '0');
612                           read_addr_outputA <= (others => '0');
613                           read_addr_outputB <= (others => '0');
614                           write_enable_dst <= '0';
615                           sel <= '1';
616                           write_last_word <= '0';
617                           inh <= '1';
618                           write_last_word_d1 <= '0';
619                           status_REG <= X"00000001";
620                           image_scaling_state <= "1010";
621                     when s11 =>
622                           transfer_type <= (others => '0');
623                           size <= (others => '0');
624                           start <= '0';
625                           addro <= (others => '0');
```

```
626                         dwrite <= '0';
627                         dataw <= (others => '0');
628                         data_input_src <= (others => '0');
629                         write_enable_src <= '0';
630                         write_addr <= (others => '0');
631                         read_addr_outputA <= (others => '0');
632                         read_addr_outputB <= (others => '0');
633                         read_write_addr <= (others => '0');
634                         write_enable_dst <= '0';
635                         sel <= '1';
636                         write_last_word <= '0';
637                         inh <= '0';
638                         write_last_word_d1 <= '0';
639                         status_REG <= X"00000001";
640                         image_scaling_state <= "1011";
641                   when s12 =>
642                         transfer_type <= (others => '0');
643                         size <= (others => '0');
644                         start <= '0';
645                         addro <= (others => '0');
646                         dwrite <= '0';
647                         dataw <= (others => '0');
648                         data_input_src <= (others => '0');
649                         write_enable_src <= '0';
650                         write_addr <= (others => '0');
651                         read_addr_outputA <= (others => '0');
652                         read_addr_outputB <= (others => '0');
653                         read_write_addr <= (others => '0');
654                         write_enable_dst <= '0';
655                         sel <= '1';
656                         write_last_word <= '0';
657                         inh <= '1';
658                         write_last_word_d1 <= '0';
659                         status_REG <= X"00000001";
660                         image_scaling_state <= "1100";
661                   when s13 =>
662                         transfer_type <= (others => '0');
663                         size <= (others => '0');
664                         start <= '0';
665                         addro <= (others => '0');
666                         dwrite <= '0';
667                         dataw <= (others => '0');
668                         data_input_src <= (others => '0');
669                         write_enable_src <= '0';
670                         write_addr <= (others => '0');
671                         read_addr_outputA <= (others => '0');
672                         read_addr_outputB <= (others => '0');
673                         read_write_addr <= (others => '0');
674                         write_enable_dst <= '0';
675                         sel <= '1';
676                         write_last_word <= '0';
677                         inh <= '1';
678                         write_last_word_d1 <= '0';
679                         status_REG <= X"00000001";
680                         image_scaling_state <= "1101";
681                   when s14 =>
682                         transfer_type <= (others => '0');
```

```
683                         size <= (others => '0');
684                         start <= '0';
685                         addro <= (others => '0');
686                         dwrite <= '0';
687                         dataw <= (others => '0');
688                         data_input_src <= (others => '0');
689                         write_enable_src <= '0';
690                         write_addr <= (others => '0');
691                         read_addr_outputA <= (others => '0');
692                         read_addr_outputB <= (others => '0');
693                         read_write_addr <= (others => '0');
694                         write_enable_dst <= '0';
695                         sel <= '1';
696                         write_last_word <= '0';
697                         inh <= '1';
698                         write_last_word_d1 <= '0';
699                         status_REG <= X"00000001";
700                         image_scaling_state <= "1110";
701                   when s15 =>
702                         transfer_type <= (others => '0');
703                         size <= (others => '0');
704                         start <= '0';
705                         addro <= (others => '0');
706                         dwrite <= '0';
707                         dataw <= (others => '0');
708                         data_input_src <= (others => '0');
709                         write_enable_src <= '0';
710                         write_addr <= (others => '0');
711                         read_addr_outputA <= (others => '0');
712                         read_addr_outputB <= (others => '0');
713                         read_write_addr <= (others => '0');
714                         write_enable_dst <= '0';
715                         sel <= '1';
716                         write_last_word <= '0';
717                         inh <= '1';
718                         write_last_word_d1 <= '0';
719                         status_REG <= X"00000001";
720                         image_scaling_state <= "1111";
721             end case;
722       end process;
723
724       state_PROCESS : process (Clk, Rst_n)
725       begin
726             if (Rst_n = '0') then
727                   current_state <= s0;
728             elsif (Clk = '1' and Clk'event) then
729                   case current_state is
730                         when s0 =>                                    -- reset
731                               if (start_transfer = '1') then
732                                     current_state <= s1;
733                               else
734                                     current_state <= s0;
735                               end if;
736                         when s1 =>                                    -- inicializacion
737                               current_state <= s2;
738                         when s2 =>                                    -- acceso al AHB bus
739                               if (transfer_active = '1' and transfer_end = '0') then
```

```
740                                         current_state <= s3;
741                                   else
742                                         current_state <= s2;
743                                   end if;
744                             when s3 =>                              -- transferencia de
    fila even
745                                   if (beat_counter = src_width_adjusted) then
746                                         current_state <= s4;
747                                   else
748                                         current_state <= s3;
749                                   end if;
750                             when s4 =>                              -- entrega de
    pixeles a TFG_ROW_INTERPOLATION
751                                   if (row_interpolation_state = "011") then
752                                         current_state <= s5;
753                                   else
754                                         current_state <= s4;
755                                   end if;
756                             when s5 =>                              -- acceso al AHB bus
757                                   if (transfer_active = '1' and transfer_end = '0') then
758                                         current_state <= s6;
759                                   else
760                                         current_state <= s5;
761                                   end if;
762                             when s6 =>                              -- transferencia de
    fila odd
763                                   if (beat_counter = src_width_adjusted) then
764                                         current_state <= s7;
765                                   else
766                                         current_state <= s6;
767                                   end if;
768                             when s7 =>                              -- entrega de
    pixeles a TFG_ROW_INTERPOLATION
769                                   if (row_interpolation_state = "110") then
770                                         current_state <= s8;
771                                   else
772                                         current_state <= s7;
773                                   end if;
774                             when s8 =>                              -- acceso al AHB bus
775                                   if (transfer_active = '1' and transfer_end = '0') then
776                                         current_state <= s9;
777                                   else
778                                         current_state <= s8;
779                                   end if;
780                             when s9 =>                              -- transferencia de
    fila destino
781                                   if (beat_counter = dst_width_adjusted - 1) then
782                                         current_state <= s10;
783                                   else
784                                         current_state <= s9;
785                                   end if;
786                             when s10 =>                             -- finalizacion de
    transferencia de fila destino
787                                   if (data_valid = '0') then
788                                         current_state <= s10;
789                                   else
790                                         current_state <= s11;
```

```
791                                     end if;
792                               when s11 =>                               -- actualizacion de
    filas procesadas
793                                     current_state <= s12;
794                               when s12 =>                               -- comprobacion de
    funcionamiento
795                                     if (row_cnt >= dst_height_REG (ADDRESS_HEIGHT - 1
    downto 0)) then
796                                           current_state <= s0;
797                                     else
798                                           current_state <= s13;
799                                     end if;
800                               when s13 =>                               -- actualizacion de
    direcciones
801                                     current_state <= s14;
802                               when s14 =>                               -- actualizacion de
    direcciones
803                                     current_state <= s15;
804                               when s15 =>                               -- actualizacion de
    direcciones
805                                     current_state <= s2;
806                         end case;
807                   end if;
808         end process;
809
810         event_detector_PROCESS : process (Clk, control_REG, control_REG_d1)
811         begin
812               if (Clk'event and Clk = '1')  then
813                     control_REG_d1 <= control_REG (0);
814               end if;
815               start_transfer <= control_REG (0) and (not control_REG_d1);
816         end process;
817
818         delay_PROCESS : process (Clk, Rst_n)
819         begin
820               if (Rst_n = '0') then
821                     beat_counter_d1 <= (others => '0');
822                     address_index_d1 <= (others => '0');
823                     internal_address_d1 <= (others => '0');
824                     internal_address_d2 <= (others => '0');
825                     internal_address_d3 <= (others => '0');
826                     internal_address_d4 <= (others => '0');
827                     internal_address_d5 <= (others => '0');
828               elsif (Clk'event and Clk = '1')  then
829                     beat_counter_d1 <= beat_counter;
830                     address_index_d1 <= address_index (1 downto 0);
831                     internal_address_d1 <= internal_address;
832                     internal_address_d2 <= internal_address_d1;
833                     internal_address_d3 <= internal_address_d2;
834                     internal_address_d4 <= internal_address_d3;
835                     internal_address_d5 <= internal_address_d4;
836               end if;
837         end process;
838
839         read_write_address_PROCESS : process (Clk, Rst_n)
840         begin
841               if (Rst_n = '0') then
```

```
842                     row_address_read <= (others => '0');
843                     row_address_write <= (others => '0');
844               elsif (Clk = '1' and Clk'event) then
845                     row_address_read <= address_src_REG + src_width_REG (15 downto 0) *
      (zeros_embedd_yofs & embedd_yofs);
846                     row_address_write <= address_dst_REG + dst_offset;
847               end if;
848         end process;
849
850         read_address_PROCESS : process (Clk, Rst_n)
851         begin
852               if (Rst_n = '0') then
853                     row_address_read_even <= (others => '0');
854                     row_address_read_odd <= (others => '0');
855               elsif (Clk = '1' and Clk'event) then
856                     row_address_read_even <= row_address_read;
857                     row_address_read_odd <= row_address_read + src_width_REG;
858               end if;
859         end process;
860
861         counter_acumulator_PROCESS : process (Clk, current_state)
862         begin
863               if (current_state = s0) then
864                     dst_offset <= (others => '0');
865                     row_cnt <= (others => '0');
866                     offset_counter <= (others => '0');
867               elsif (Clk = '1' and Clk'event) then
868                     if (inh = '0') then
869                           dst_offset <= dst_offset + dst_width_REG;
870                           row_cnt <= row_cnt + 1;
871                           offset_counter <= offset_counter + dst_width_REG (1 downto 0);
872                     end if;
873               end if;
874         end process;
875
876         ajusted_size_PROCESS : process (Clk, Rst_n)
877         begin
878               if (Rst_n = '0') then
879                     src_width_adjusted <= (others => '0');
880                     dst_width_adjusted <= (others => '0');
881               elsif (Clk = '1' and Clk'event) then
882                     if (src_width_REG (1 downto 0) = "00") then
883                           src_width_adjusted <= src_width_REG (11 downto 2);
884                     elsif (src_width_REG (1 downto 0) = "11") then
885                           src_width_adjusted <= src_width_REG (11 downto 2) +
      "0000000010";
886                     else
887                           src_width_adjusted <= src_width_REG (11 downto 2) +
      "0000000001";
888                     end if;
889                     if (dst_width_REG (1 downto 0) = "00") then
890                           dst_width_adjusted <= dst_width_REG (11 downto 2);
891                     elsif (dst_width_REG (1 downto 0) = "11") then
892                           dst_width_adjusted <= dst_width_REG (11 downto 2) +
      "0000000010";
893                     else
894                           dst_width_adjusted <= dst_width_REG (11 downto 2) +
```

```
    "0000000001";
                  end if;
            end if;
      end process;

      index_PROCESS : process (Clk, Rst_n)
      begin
            if (Rst_n = '0') then
                  src_index <= 0;
            elsif (Clk = '1' and Clk'event) then
                  if ((INT_ADDRESS_WIDTH - SRC_ADDRESS_WIDTH) = 1) then
                        src_index <= 1;
                  else
                        src_index <= 2;
                  end if;
            end if;
      end process;

      index_offset_PROCESS : process (Clk, Rst_n)
      begin
            if (Rst_n = '0') then
                  index_offset <= (others => '0');
            elsif (Clk = '1' and Clk'event) then
                  if (sel = '0') then
                        index_offset <= row_address_read_even (1 downto 0);
                  elsif (sel = '1') then
                        index_offset <= row_address_read_odd (1 downto 0);
                  else
                        index_offset <= (others => '0');
                  end if;
            end if;
      end process;

      address_index_PROCESS : process (Clk, Rst_n)
      begin
            if (Rst_n = '0') then
                  address_index <= (others => '0');
            elsif (Clk = '1' and Clk'event) then
                  address_index <= embedd_xofs + (zeros_index_offset & index_offset);
            end if;
      end process;

      pixels_PROCESS : process (Rst_n, Clk)
      begin
            if (Rst_n = '0') then
                  pixel_even <= (others => '0');
                  pixel_odd <= (others => '0');
            elsif (Clk = '1' and Clk'event) then
                  if (address_index_d1 = "00") then
                        pixel_even <= data_outputA (31 downto 24);
                        pixel_odd <= data_outputB (23 downto 16);
                  elsif (address_index_d1 = "01") then
                        pixel_even <= data_outputA (23 downto 16);
                        pixel_odd <= data_outputB (15 downto 8);
                  elsif (address_index_d1 = "10") then
                        pixel_even <= data_outputA (15 downto 8);
                        pixel_odd <= data_outputB (7 downto 0);
```

```
951                     elsif (address_index_d1 = "11") then
952                           pixel_even <= data_outputA (7 downto 0);
953                           pixel_odd <= data_outputB (31 downto 24);
954                     else
955                           pixel_even <= (others => '0');
956                           pixel_odd <= (others => '0');
957                     end if;
958               end if;
959         end process;
960
961         byte0_byte1_byte2_PROCESS : process (Clk, Rst_n)
962         begin
963               if (Rst_n = '0') then
964                     byte0 <= (others => '0');
965                     byte1 <= (others => '0');
966                     byte2 <= (others => '0');
967                     byte3 <= (others => '0');
968               elsif (Clk = '1' and Clk'event) then
969                     if (internal_address_d2 (1 downto 0) = "00") then
970                           byte0 <= pixel_out;
971                     end if;
972                     if (internal_address_d2 (1 downto 0) = "01") then
973                           byte1 <= pixel_out;
974                     end if;
975                     if (internal_address_d2 (1 downto 0) = "10") then
976                           byte2 <= pixel_out;
977                     end if;
978                     if (internal_address_d2 (1 downto 0) = "11") then
979                           byte3 <= pixel_out;
980                     end if;
981               end if;
982         end process;
983
984         last_word_PROCESS : process (Clk, Rst_n)
985         begin
986               if (Rst_n = '0') then
987                     last_word <= (others => '0');
988                     last_word_d1 <= (others => '0');
989               elsif (Clk = '1' and Clk'event) then
990                     if (write_last_word = '1') then
991                           last_word <= data_input_dst;
992                     end if;
993                     if (write_last_word_d1 = '1') then
994                           last_word_d1 <= last_word;
995                     end if;
996               end if;
997         end process;
998
999         words_PROCESS : process (Rst_n, Clk)
1000        begin
1001              if (Rst_n = '0') then
1002                    compound_word <= (others => '0');
1003                    new_word <= (others => '0');
1004              elsif (Clk = '1' and Clk'event) then
1005                    if (offset_counter = "00") then
1006                          compound_word <= byte0 & byte1 & byte2 & byte3;
1007                          new_word <= byte0 & byte1 & byte2 & byte3;
```

```
1008                    elsif (offset_counter = "01") then
1009                          compound_word <= last_word_d1 (31 downto 24) & byte0 & byte1 &
       byte2;
1010                          new_word <= byte3 & byte0 & byte1 & byte2;
1011                    elsif (offset_counter = "10") then
1012                          compound_word <= last_word_d1 (31 downto 16) & byte0 & byte1;
1013                          new_word <= byte2 & byte3 & byte0 & byte1;
1014                    elsif (offset_counter = "11") then
1015                          compound_word <= last_word_d1 (31 downto 8) & byte0;
1016                          new_word <= byte1 & byte2 & byte3 & byte0;
1017                    else
1018                          compound_word <= (others => '0');
1019                          new_word <= (others => '0');
1020                    end if;
1021              end if;
1022        end process;
1023
1024        first_word_PROCESS : process (Rst_n, Clk)
1025        begin
1026              if (Rst_n = '0') then
1027                    data_input_dst <= (others => '0');
1028              elsif (Clk = '1' and Clk'event) then
1029                    if (offset_counter = "00" and internal_address_d4 (INT_ADDRESS_WIDTH
       - 1 downto 2) =
1030
           zeros_internal_address) then
1031                          data_input_dst <= compound_word;
1032                    elsif (offset_counter = "01" and internal_address_d3 (
      INT_ADDRESS_WIDTH - 1 downto 2) =
1033
           zeros_internal_address) then
1034                          data_input_dst <= compound_word;
1035                    elsif (offset_counter = "10" and internal_address_d2 (
      INT_ADDRESS_WIDTH - 1 downto 2) =
1036
           zeros_internal_address) then
1037                          data_input_dst <= compound_word;
1038                    elsif (offset_counter = "11" and internal_address_d1 (
      INT_ADDRESS_WIDTH - 1 downto 2) =
1039
           zeros_internal_address) then
1040                          data_input_dst <= compound_word;
1041                    else
1042                          data_input_dst <= new_word;
1043                    end if;
1044              end if;
1045        end process;
1046
1047        -- pragma translate_off
1048        bootmsg : report_version
1049        generic map ("TFG_IMAGE_SCALING" &
1050        ": Trabajo de Fin de Grado : Hardware AMBA bus IP for image scaling");
1051        -- pragma translate_on
1052
1053   end Behavioral;
1054
1055
```