Laboratory Report

Han Liu
UoG:2289217L

CONTENTS

1 INTRODUCTION

In digital circuit field, we usually need to tackle and coordinate the communication between different modules. Generally, communication between different modules in a computing system makes use of a common bus. However, lots of different modules might need to communicate simultaneously, so a common bus is not efficient as only one piece of data can be transmitted on the bus at any time and many clock cycles need to waste time in negotiating which modules can access the bus. To tackle this problem, we design a crossbar switch system to route data packets. In details, the data packet will be given enough routing information to the crossbar switch on how data from different inputs should be routed, in this case, the lowest order bit of header byte is used to determine the routing direction.
In this report, we firstly design the overall operation of the whole system, then we design each block individually. Secondly, we create different test vertors for different scenerios. Thirdly, we will show the simulation results for different test vectors to examinate whether the design is correct. Fourly, we will discuss some problems we met in the design process as well as some existing problems and ways to neutralise them. Lastly, we draw our conlcusion.

2 DESIGN AND METHODOLOGY

2.1 OVERALL DESIGN SCHEME

In order to route different packets to $outx$ and $outy$, we firstly need to determine the routing methods. As there are totally four routing directions: $x \Rightarrow y$, $x \Rightarrow x$, $y \Rightarrow x$, $y \Rightarrow y$, so we need two different assignments for a single signal. Therefore, signal $available$ need to be divided into $stim\_xavail$ and $stim\_yavail$, when x-input wants to transfer a packet, it will activate $stim\_xavail$ and $controlx$ component; when y-input wants to transfer a packet, it will activate $stim\_yavail$ and $controly$ component. Similarly, $data\_in$ and $enable$ should be divided into $stimx\_data$, $stimy\_data$ and $xenable$, $yenable$. Also, $reqx$ and $reqy$ should also have two different assigments. More details will be given in "design of different blocks" section.
Then we could configure the whole data flow of the system.
Firstly, stimulus block will receive a data, if x-input receive the data, then data flag $stim\_xavail$ will be raised; if y-input receive the data, then data flag $stim\_yavail$ will be raised.
Secondly, it will communicate with $control$ block. If x-input receives the data, then it will communicate with $controlx$; if y-input receives the data, then it will communicate with $controly$. The control block will examinate the lowest bit of the header work, if the bit is 0, then it will send instructions to the output block to let $xdata\_out$ equal to input; if the bit is 1, then it will send instructions to the output block to let $ydata\_out$ equal to input.
Lastly, output block needs to receive the instructions of control block to export correct output.
After configuring the whole operational process, we now design each block individually.

2.2 DESIGN OF DIFFERENT BLOCKS

2.2.1 STIMULUS BLOCK DESIGN

The main function of stimulus block is to receive input data, so the design of stimulus block should be determined by input. As illustrated before, if x-input receives the data, then $stim\_xavail$ will be raised; if y-input receives the data, then $stim\_yavail$ will be raised.
Also, the whole operation of this block should satisfy synchronous design methodology. As the given reset signal $reset$ is activated for $2450ns$, so in this case we set a delay to stimulus block for synchrounous system operation between stimulus block and control block. Therefore, stimulus block have waited for

five periods to transfer the data.
Then the stimulus block will use a kind of "handshaking protocol" to communicate with control block. Suppose y-input transfers the data. In this case, $stim\_yavail = 1$ is transfered to the $controly$ block, then $controly$ will send $yenable$ signal to the stimulus block. After receiving $yenable$ signal, then stimulus block will transfer the header work. When stimulus block receives $yenable$ signal again, it will trasfer the data packet.
For describing the whole operation more clearly, we can plot the functional timing diagram of this block as following: (suppose y-input transfers the data)

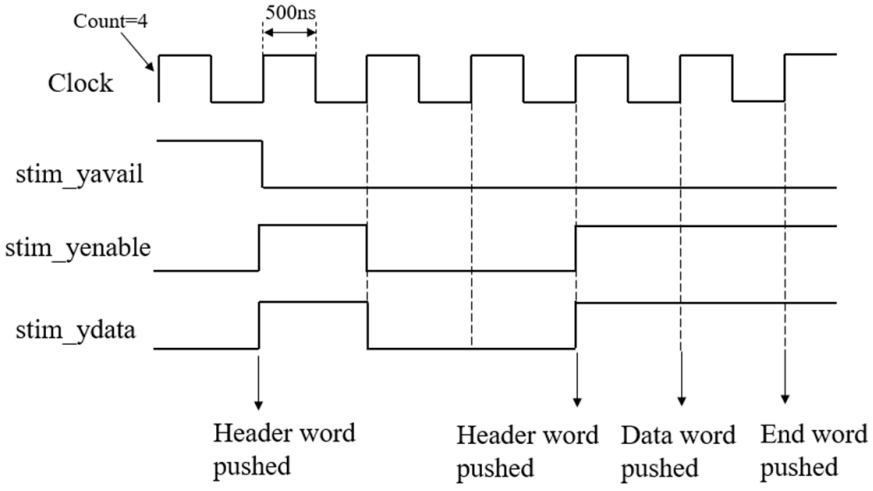

Figure 2.1: Timing diagram of stimulus block

The time between two raises of enable is not three periods actually, but the basic operation of this block is the same with the figure. Then we translate the figure into VHDL code to accomplish our design.

2.2.2 Output block design

The main function of the output block is to receive the instructions of control block and export the output. Firstly, it needs to determine which output port should export the data. As there are two types of outputs: x-output and y-output, so we need to consider these two situations. In the first situation, data is transfered to x-output regardless of whether the input comes from x or y, which can be seen as the same situation. The second situation is to transfer the data to y-output regardless of whether input comes from x or y.
Secondly, it needs to export the data when receiving the instructions. Also, it is required to consider two situations: exporting x-output and exporting y-output. Same with above analysis, when the output port is determined, whether output instructions come from x-input or y-input can be seen as the same case. Therefore, we can design a finite state machine of four states to accomplish this design as following:

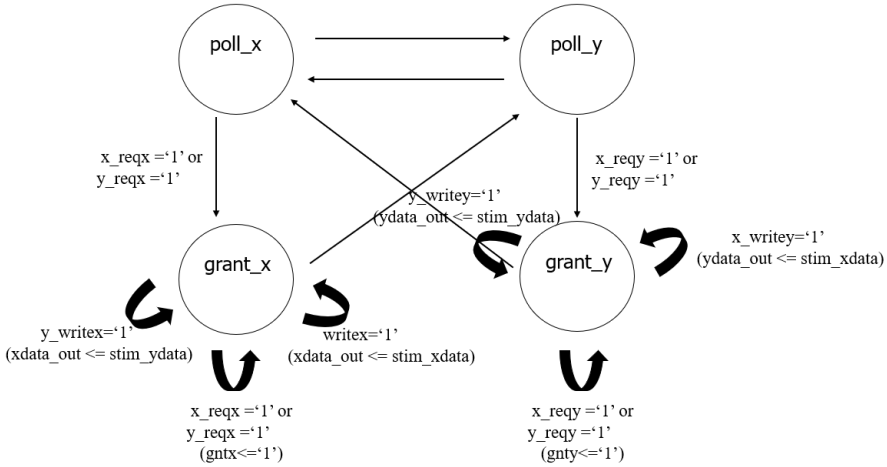

Figure 2.2: Finite State Machine of output block

Then we can translate the diagram of finite state machine into VHDL to accomplish the design.

2.2.3 CONTROL BLOCK DESIGN

The main function of the control block is to determine where to send the data packet and send instructions to output block to export the data.
Firstly, it needs to communicate with stimulus block and get header work. When the stimulus block sends $availabel$ signal, control block will reply with $enable$ signal, then stimulus block will push the header work. When the control block receives header work, it will examinate the lowest bit of header word: when it's 0, it will transfer the data packet to x-output; when it's 1, it will transfer the data packet to y-output.
Secondly, it needs to communicate with output block to give transferring instructions in terms of routing direction and export data packet. The communication is like a "handshaking" protocol. When the lowest bit of header word is 0, it will send $reqx = 1$ to output block; when it's 1, it will send $reqy = 1$ to output block. Then the output block will reply with $gntx = 1$ or $gnty = 1$. After receiving reply, control block will send writing instructions to the output block to let data be exported.
The whole operation can be illustrated in a finite state machine as following:

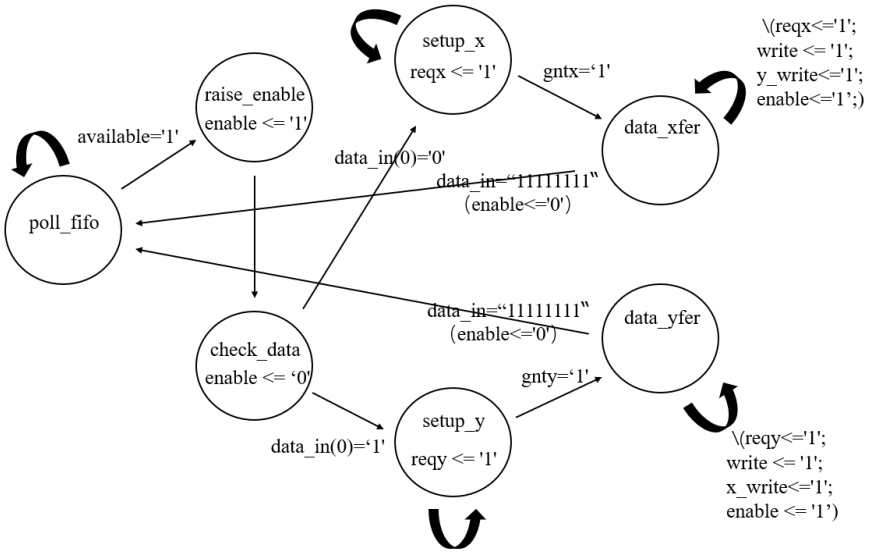

Figure 2.3: Finite State Machine of control block

Then we can translate the diagram of finite state machine into VHDL to achieve the design.

3 TESTING

As there are totally four routing directions: $x \Rightarrow y$, $x \Rightarrow x$, $y \Rightarrow x$, $y \Rightarrow y$, so we need to test these four cases individually. We can configure test vectors as following:

1. $x \Rightarrow x$: we create header word to be 00000010, data word to be 11100111 and end word to be 11111111.
2. $x \Rightarrow y$: we create header word to be 00000001, data word to be 10101011 and end word to be 11111111.
3. $y \Rightarrow x$: we create header word to be 00000010, data word to be 10111011 and end word to be 11111111.
4. $y \Rightarrow y$: we create header word to be 00000001, data word to be 10101010 and end word to be 11111111.

4 RESULTS

The simulation results for each test vector is shown in the following part.
The simulation result for "$x \Rightarrow x$" test vectors is shown as following:

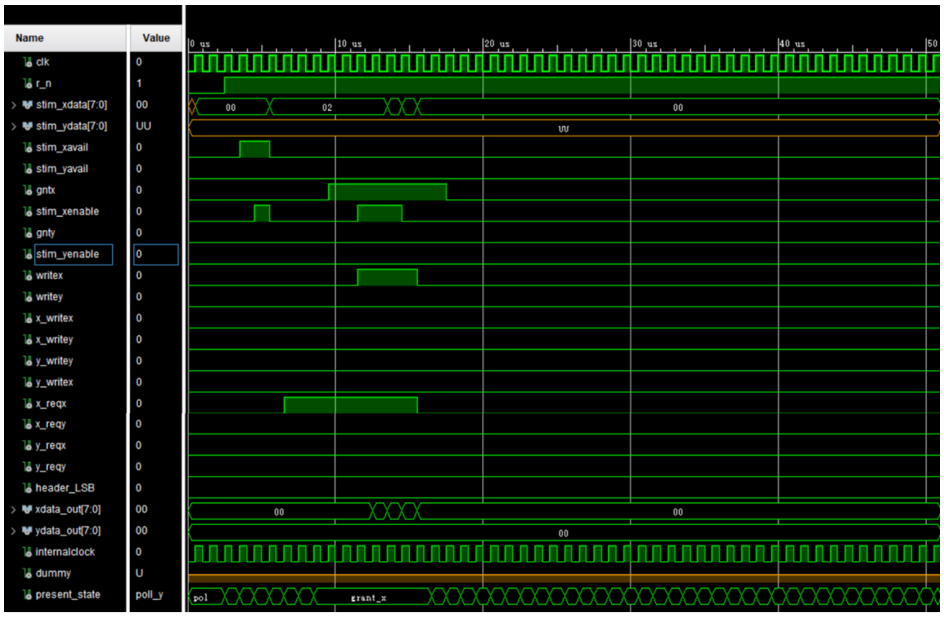

Figure 4.1: Simulation result for "$x \Rightarrow x$"

The simulation result for "$x \Rightarrow y$" test vectors is shown as following:

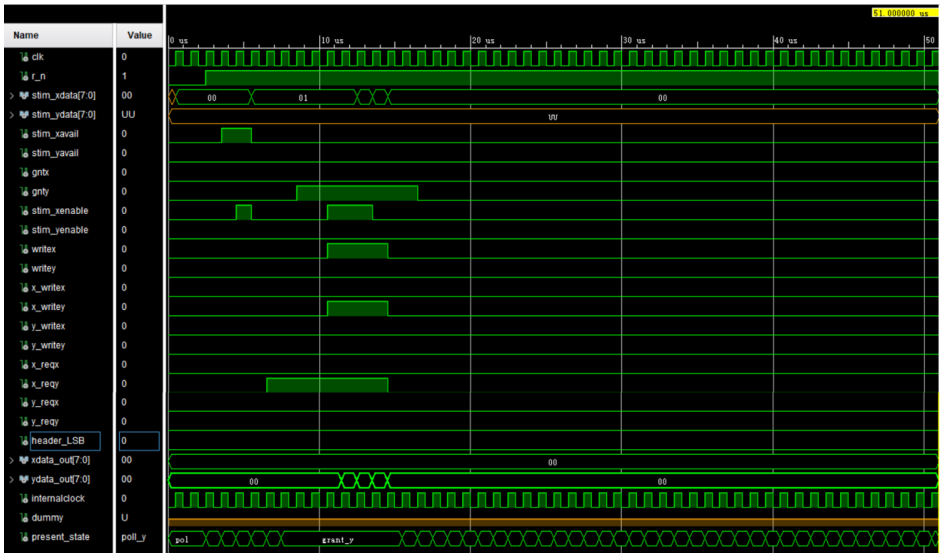

Figure 4.2: Simulation result for "$x \Rightarrow y$"

The simulation result for "$y \Rightarrow x$" test vectors is shown as following:

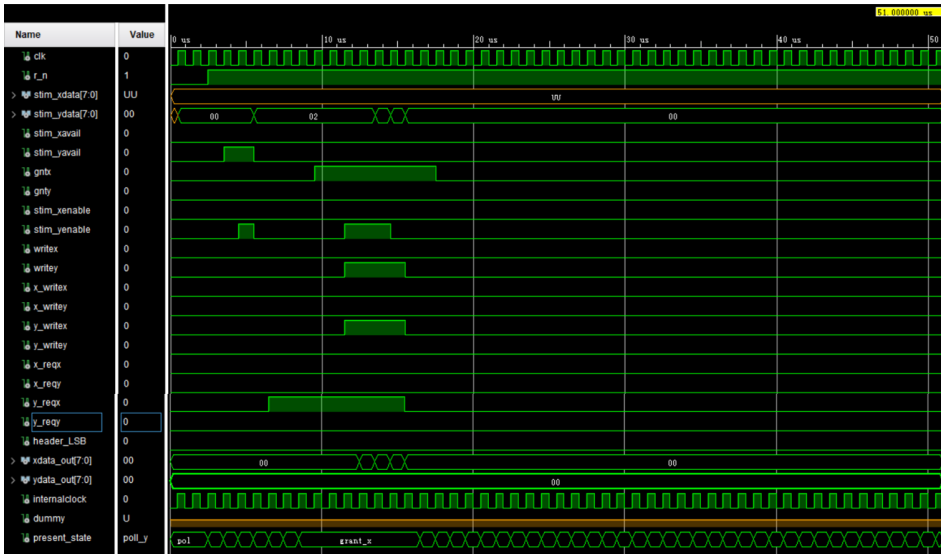

Figure 4.3: Simulation result for "$y \Rightarrow x$"

The simulation result for "$y \Rightarrow y$" test vectors is shown as following:

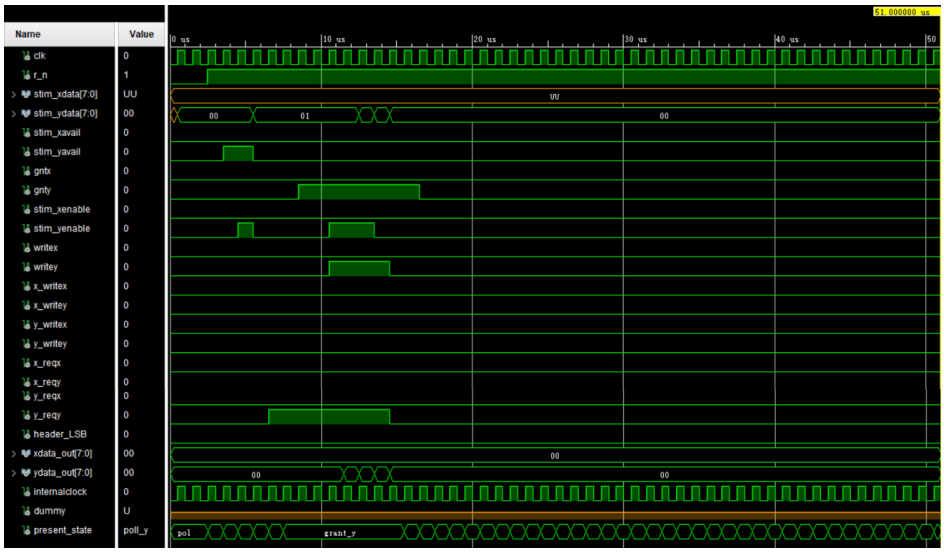

Figure 4.4: Simulation result for "$y \Rightarrow y$"

5 Problems

5.1 Encountered problems in design process

The problems that I encounter in the design process is mostly related to the synchronous operation of the system. For example, at the beginning, I always can't export the header word but the remaining data word and end word can be exported normally. After a detailed analysis, I found that this phenomenon

is caused by non-synchronous operation of stimulus block and control block. Because when header work are initially pushed, control block is not ready for routing the packets. Therefore, I transmit the header work again when control block and output block are ready to transmit. Then this probelms are successfully fixed.

5.2 Existing problems in current design

The existing problems mainly consists of two parts.
Firstly, as we only configure a single output process, so it can only tansmit a single output at a time. Therefore, $x \Rightarrow x$ and $y \Rightarrow y$ or $x \Rightarrow y$ and $y \Rightarrow x$ can not be transmitted simutaneously.
Secondly, because of the same reason, it can not achieve that let one input be broadcasted to both outputs.
In the next section, we will present some approaches to tackle these two problems.

6 Improvements

There are two improvements of the experiment:

1. $x \Rightarrow x$ and $y \Rightarrow y$ or $x \Rightarrow y$ and $y \Rightarrow x$ can be transmitted simutaneously
2. Enable one input to be broadcasted to both outputs

We will present corresponding approach to each of them.

6.1 Improvements for transmitting simutaneously

6.1.1 Design methodology

In order to let two inputs can be transmitted simutaneously, we just need to revise the output module. Orginally, the output module is a single process, thus it can only tansmitte a single word at a single time. We can decompose it into two independent processes, with one process tackling x_output and the other process tackling y_output. The deisgn process is quite trival, we just need to decompose the original finite state machine into two parts, the first part consists of $poll\_x$ and $grant\_x$, the other part consists of $poll\_y$ and $grant\_y$. Two state machines can be seen in the following figures:

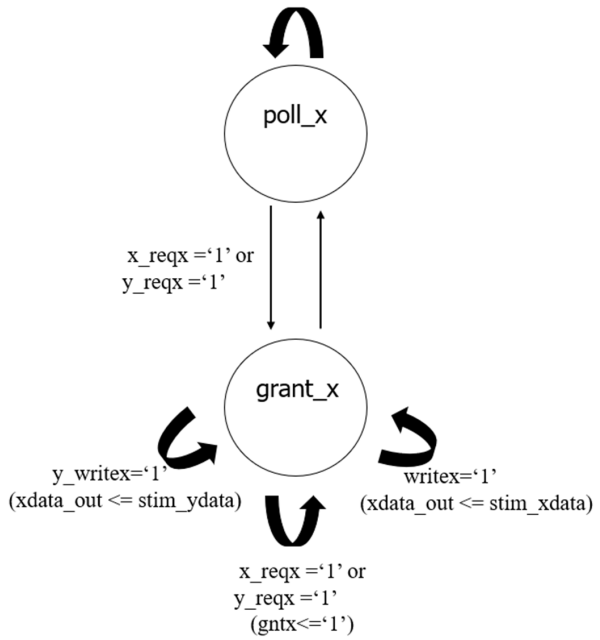

Figure 6.1: Finite state machine of x_output

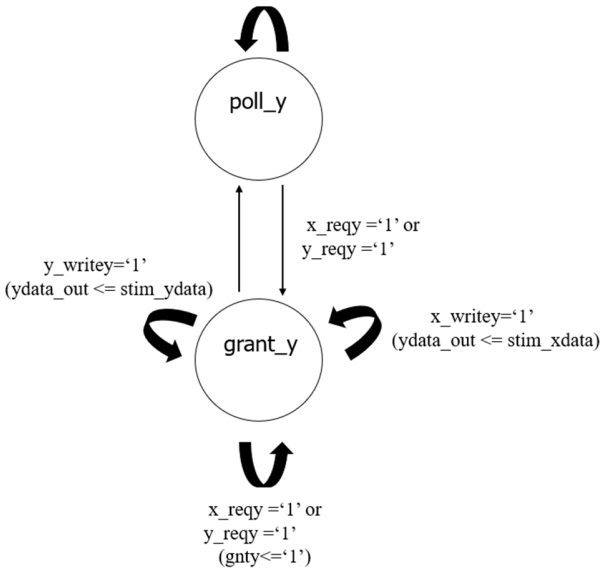

Figure 6.2: Finite state machine of y_output

6.1.2 SIMULATION RESULTS

Then we could create test vectors to test the function of the system.
Firstly, we need to test $x \Rightarrow x$ and $y \Rightarrow y$ could transmitte simutaneously. We choose the test test vectors same with "TESTING" section. The simulation result is as following:

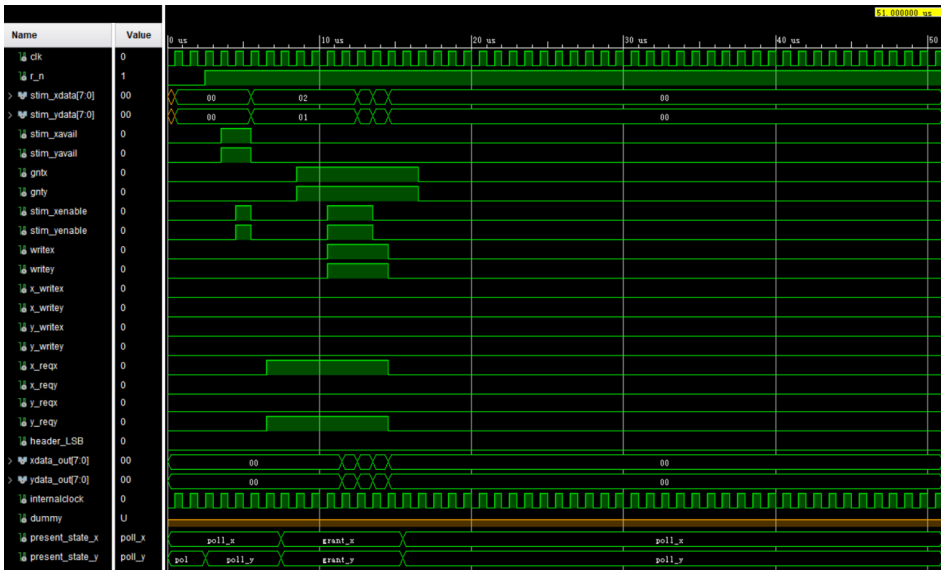

Figure 6.3: Simulation result for $x \Rightarrow x$ and $y \Rightarrow y$ transmitting simutaneously

Secondly, we need to test $x \Rightarrow y$ and $y \Rightarrow x$ could transmitte simutaneously. We choose the test test vectors same with "TESTING" section. The simulation result is as following:

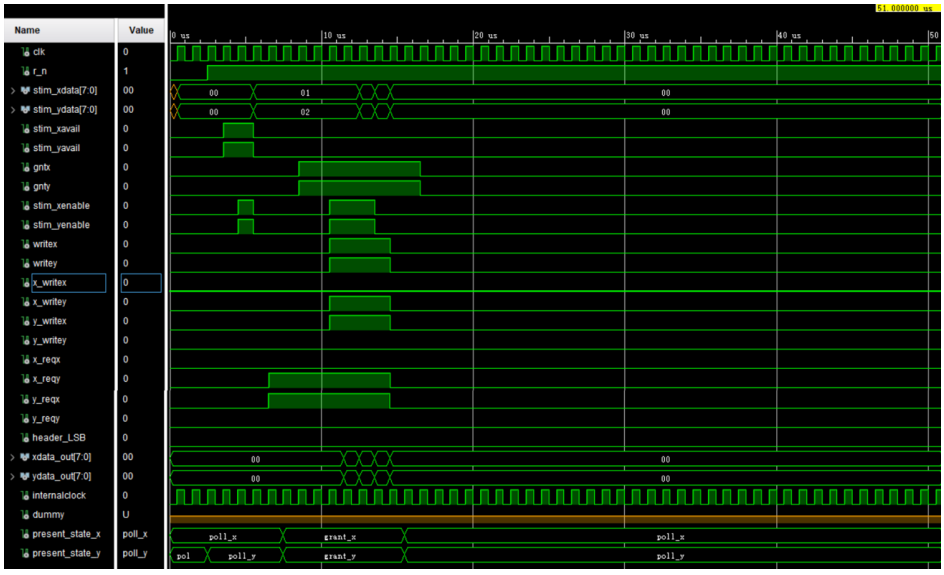

Figure 6.4: Simulation result for $x \Rightarrow y$ and $y \Rightarrow x$ transmitting simutaneously

6.2 IMPROVEMENTS FOR INPUT BROADCASTING

6.2.1 DESIGN METHODOLOGY

In order to let one input be broadcasted to both outputs, we just need to revise the control block. We could create a new rule: if the highest order of header word is '1', then the data packet will broadcast to both outputs, otherwise it will export to only one output. Therefore, we could add two more states in the output block called $setup\_xy$ and $data\_xyfer$. The new state diagram can be seen as following:

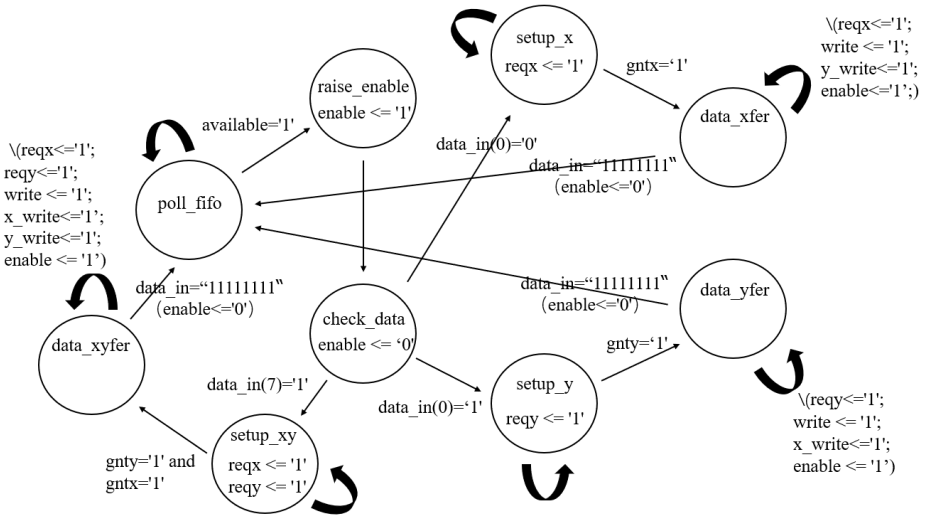

Figure 6.5: Finite state machine of broadcasting-enable control block

6.2.2 SIMULATION RESULTS

We could also create test vectors to test the function of the system.
Firstly, we test x-input can transmitte to x-output and y-output simutaneously. We choose the test vectors same with "TESTING" section. The simulation result is as following:

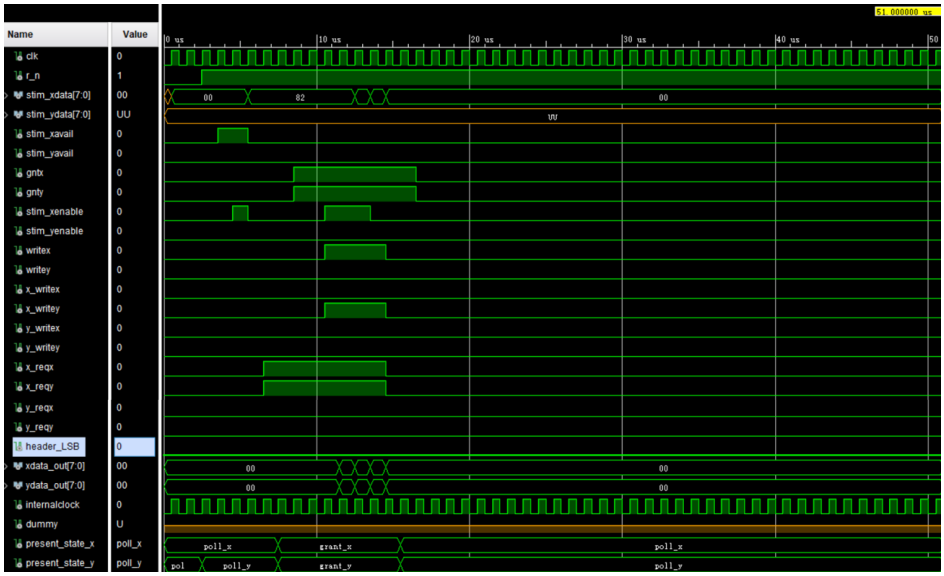

Figure 6.6: Simulation result for x-input broadcasting

Secondly, we test y-input can transmitte to x-output and y-output simutaneously. We choose the test test vectors same with "TESTING" section. The simulation result is as following:

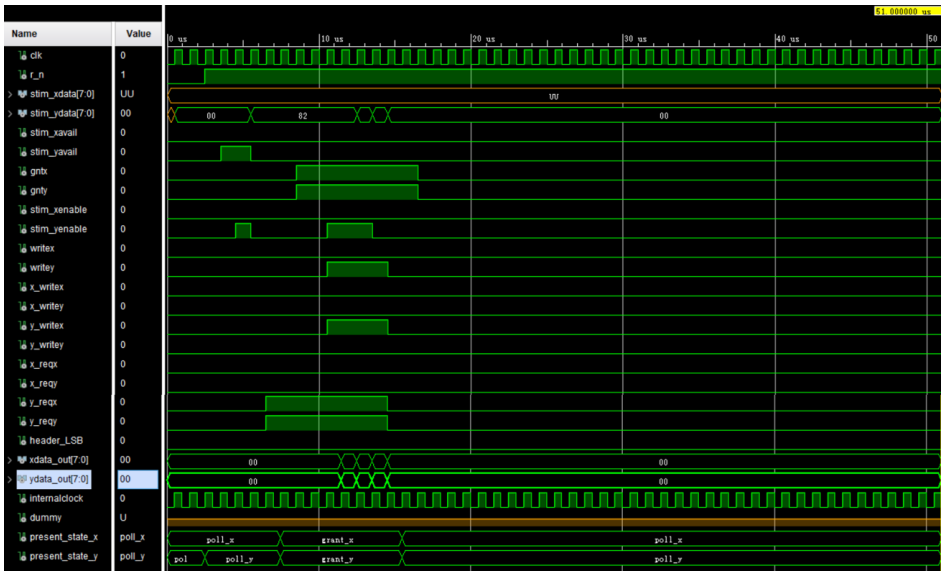

Figure 6.7: Simulation result for y-input broadcasting

7 CONCLUSION

In this laboratory report, we design a crossbar switch system to route data packets according to the lowest order bit of header byte. The routing is achieved by three different blocks: stimulus block, output block and control block. The main function of stimulus block is to receive input data packet, and control block is used to determine where to send the data packet and send instructions to output block to export the data. And the main function of the output block is to receive the instructions of control block and export the output. We draw timing diagrams and state diagrams to analyze and design each block. After completing the design process, we need to create test vectors. As there are totally four routing directions: $x \Rightarrow y$, $x \Rightarrow x$, $y \Rightarrow x$, $y \Rightarrow y$, so we need to create four different test vectors individually. Through simulation, we can confirm that our design is correct. Finally, there are some existing problems in our design: it can't export two data simutaneously and can't broadcast one input to both outputs. Therefore, we can revise the design of control block and output block to tackle these problems.

8 CODE LISTINGS

- VHDL code for *top_tb.vhd*

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all; -- for internal counter etc.
4
5  -- Uncomment the following library declaration if using
6  -- arithmetic functions with Signed or Unsigned values
7  --use IEEE.NUMERIC_STD.ALL;
8
9  -- Uncomment the following library declaration if instantiating
10 -- any Xilinx leaf cells in this code.
11 --library UNISIM;
12 --use UNISIM.VComponents.all;
13
14 entity top_tb is
15 end top_tb;
16
17 architecture behavioral of top_tb is
18 component control
19 PORT(
20 enable : OUT std_logic;
21 reqx : OUT std_logic;
22 reqy : OUT std_logic;
23 write: OUT std_logic;
24 x_write : OUT std_logic;
25 y_write : OUT std_logic;
26 available : IN std_logic;
27 clk : IN std_logic;
28 data_in : IN std_logic_vector(7 DOWNTO 0);
29 gntx : IN std_logic;
30 gnty : IN std_logic;
31 reset : IN std_logic;
32 header_LSB : IN std_logic
33 );
34
35 end component;
36 --INPUT
37 signal clk: std_logic;
38 signal r_n: std_logic;
39 signal stim_xdata: std_logic_vector(7 downto 0);
40 signal stim_ydata: std_logic_vector(7 downto 0);
41 signal stim_xavail: std_logic :='0';
42 signal stim_yavail: std_logic :='0';
43 signal gntx: std_logic :='0';
44 signal gnty: std_logic :='0';
45 --OUTPUT
46 signal stim_xenable: std_logic :='0';
47 signal stim_yenable: std_logic :='0';
48 signal writex: std_logic :='0';
49 signal writey: std_logic :='0';
50 signal x_writex: std_logic :='0';
51 signal x_writey: std_logic :='0';
52 signal y_writex: std_logic :='0';
53 signal y_writey: std_logic :='0';
```

```
signal x_reqx: std_logic :='0';
signal x_reqy: std_logic :='0';
signal y_reqx: std_logic :='0';
signal y_reqy: std_logic :='0';
signal header_LSB: std_logic :='0';
--    signal data_out: std_logic_vector(7 downto 0);

signal xdata_out: std_logic_vector(7 downto 0);
signal ydata_out: std_logic_vector(7 downto 0);
--VARIABLE
----stim_xbar.vhd
signal internalclock : std_logic; -- internal clock
signal dummy : std_logic; -- dummy signal
----output.vhd
type states is (poll_x, poll_y, grant_x, grant_y);
-- Present state.
signal present_state : states;


begin

controlx:control port map(
--Input
clk=>clk,
reset=>r_n,
data_in=>stim_xdata,
available=>stim_xavail,
gntx=>gntx,
gnty=>gnty,
--Output
enable=>stim_xenable,
write=>writex,
x_write=>x_writey,
reqx=>x_reqx,
reqy=>x_reqy,
header_LSB=>header_LSB
);
controly:control port map(
--Input
clk=>clk,
reset=>r_n,
data_in=>stim_ydata,
available=>stim_yavail,
gntx=>gntx,
gnty=>gnty,
--Output
enable=>stim_yenable,
write=>writey,
y_write=>y_writex,
reqx=>y_reqx,
reqy=>y_reqy,
header_LSB=>header_LSB
);

resetgen : process
begin -- process resetgen
r_n <= '0';
wait for 2450 ns;
r_n <= '1';
```

```
113 -- Add more entries if required at this point !
114
115 wait until dummy'event;
116 end process resetgen;
117
118 -- purpose: Generates the internal clock source. This is a 50% duty
119 -- cycle clock source, period 1000ns, with the first half of
120 -- the cycle being logic '0'.
121 --
122 -- outputs: internalclock
123 clockgen : process
124 begin -- process clockgen
125 internalclock <= '0';
126 wait for 500 ns;
127 internalclock <= '1';
128 wait for 500 ns;
129 end process clockgen;
130 -- purpose: Generates the stimuli for the data, using a counter based
131 -- approach which is much cleaner than explicit timings
132 --
133 -- outputs: stim_inst
134 -- stim_a
135 -- stim_b
136
137
138 --For datagen process, each time you could only run one process, and needs to ...
        comment out other three processes
139 -- x -> x
140 datagen1 : process
141 -- Since we're going to output a number of different test
142 -- data we need an internal counter to keep track of things.
143
144 variable count : unsigned (5 downto 0) := "000000";
145 begin -- process datagen1
146 wait until internalclock'event and internalclock = '1';
147 count := count + 1;
148 if count = 4 then -- simple data transfer x -> x
149 stim_xavail <= '1'; -- data flag raised
150 wait until internalclock'event and internalclock = '1'
151 and stim_xenable = '1';
152
153 stim_xdata <= "00000010"; -- header word pushed when system
154 stim_xavail <= '0'; -- requests transfer, and flag back down
155
156 wait until internalclock'event and internalclock = '1'
157 and stim_xenable = '1';
158
159 stim_xdata <= "00000010";
160
161 wait until internalclock'event and internalclock = '1'
162 and stim_xenable = '1';
163
164 stim_xdata <= "11100111"; -- data word pushed
165
166 wait until internalclock'event and internalclock = '1'
167 and stim_xenable = '1';
168
169 stim_xdata <= "11111111"; -- end word pushed
170 elsif count = 5 then    -- NO OPERATION
```

```
171 stim_xavail ≤ '0';
172 stim_xdata ≤ "00000000";
173 else  -- NO OPERATION
174 stim_xavail ≤ '0';
175 stim_xdata ≤ "00000000";
176 end if;
177 end process datagen1;
178
179 -- x -> y
180 datagen2 : process
181 -- Since we're going to output a number of different test
182 -- data we need an internal counter to keep track of things.
183
184 variable count : unsigned (5 downto 0) := "000000";
185 begin -- process datagen1
186 wait until internalclock'event and internalclock = '1';
187 count := count + 1;
188 if count = 4 then -- simple data transfer x -> y
189 stim_xavail ≤ '1'; -- data flag raised
190 wait until internalclock'event and internalclock = '1'
191 and stim_xenable = '1';
192
193 stim_xdata ≤ "00000001"; -- header word pushed when system
194 stim_xavail ≤ '0'; -- requests transfer, and flag back down
195
196 wait until internalclock'event and internalclock = '1'
197 and stim_xenable = '1';
198 stim_xdata ≤ "00000001";
199
200 wait until internalclock'event and internalclock = '1'
201 and stim_xenable = '1';
202 stim_xdata ≤ "10101011"; -- data word pushed
203
204 wait until internalclock'event and internalclock = '1'
205 and stim_xenable = '1';
206
207 stim_xdata ≤ "11111111"; -- end word pushed
208 elsif count = 5 then    -- NO OPERATION
209 stim_xavail ≤ '0';
210 stim_xdata ≤ "00000000";
211 else  -- NO OPERATION
212 stim_xavail ≤ '0';
213 stim_xdata ≤ "00000000";
214 end if;
215 end process datagen2;
216
217 -- y -> x
218 datagen3 : process
219 -- Since we're going to output a number of different test
220 -- data we need an internal counter to keep track of things.
221
222 variable count : unsigned (5 downto 0) := "000000";
223 begin -- process datagen1
224 wait until internalclock'event and internalclock = '1';
225 count := count + 1;
226 if count = 4 then -- simple data transfer y -> x
227 stim_yavail ≤ '1'; -- data flag raised
228 wait until internalclock'event and internalclock = '1'
229 and stim_yenable = '1';
```

```
230
231 stim_ydata ≤ "00000010"; -- header word pushed when system
232 stim_yavail ≤ '0'; -- requests transfer, and flag back down
233
234
235 wait until internalclock'event and internalclock = '1'
236 and stim_yenable = '1';
237
238 stim_ydata ≤ "00000010"; -- header word pushed when system
239
240 wait until internalclock'event and internalclock = '1'
241 and stim_yenable = '1';
242
243 stim_ydata ≤ "10111010"; -- data word pushed
244
245 wait until internalclock'event and internalclock = '1'
246 and stim_yenable = '1';
247
248 stim_ydata ≤ "11111111"; -- end word pushed
249 elsif count = 5 then    -- NO OPERATION
250 stim_yavail ≤ '0';
251 stim_ydata ≤ "00000000";
252 else  -- NO OPERATION
253 stim_yavail ≤ '0';
254 stim_ydata ≤ "00000000";
255 end if;
256 end process datagen3;
257
258
259
260 -- y -> y
261 datagen4 : process
262 -- Since we're going to output a number of different test
263 -- data we need an internal counter to keep track of things.
264
265 variable count : unsigned (5 downto 0) := "000000";
266 begin -- process datagen1
267 wait until internalclock'event and internalclock = '1';
268 count := count + 1;
269 if count = 4 then -- simple data transfer y -> y
270 stim_yavail ≤ '1'; -- data flag raised
271 wait until internalclock'event and internalclock = '1'
272 and stim_yenable = '1';
273
274 stim_ydata ≤ "00000001"; -- header word pushed when system
275 stim_yavail ≤ '0'; -- requests transfer, and flag back down
276
277 wait until internalclock'event and internalclock = '1'
278 and stim_yenable = '1';
279 stim_ydata ≤ stim_ydata; -- header word pushed again
280
281 wait until internalclock'event and internalclock = '1'
282 and stim_yenable = '1';
283
284 stim_ydata ≤ "10101010"; -- data word pushed
285
286 wait until internalclock'event and internalclock = '1'
287 and stim_yenable = '1';
288
```

```
289 stim_ydata ≤ "11111111"; -- end word pushed
290
291
292 elsif count = 5 then    -- NO OPERATION
293 stim_yavail ≤ '0';
294 stim_ydata ≤ "00000000";
295 else  -- NO OPERATION
296 stim_yavail ≤ '0';
297 stim_ydata ≤ "00000000";
298 end if;
299 end process datagen4;
300
301
302
303
304 -- ***********************************
305
306
307 -- now drive the output clock, this is simply the internal clock
308 -- nb: could also invert the clock if desired to allow for signals
309 -- to be generated pseudo-asynchronously
310
311 clk ≤ internalclock;
312
313 --output.vhd
314 process (clk, r_n)
315 begin
316 -- Activities triggered by asynchronous reset (active low).
317 if (r_n = '0') then
318 -- Set the default state and outputs.
319 present_state ≤ poll_x;
320 gntx ≤ '0';
321 gnty ≤ '0';
322 xdata_out ≤ "00000000";
323 ydata_out ≤ "00000000";
324 elsif (clk'event and clk = '1') then
325 -- Set the default state and outputs.
326 present_state ≤ poll_x;
327 gntx ≤ '0';
328 gnty ≤ '0';
329 xdata_out ≤ "00000000";
330 ydata_out ≤ "00000000";
331 case present_state is
332
333 when poll_x =>
334 if (x_reqx = '1' or y_reqx = '1') then
335 present_state ≤ grant_x;
336 else
337 present_state ≤ poll_y;
338 end if;
339
340
341 when poll_y =>
342 if (x_reqy = '1' or y_reqy = '1') then
343 present_state ≤ grant_y;
344 else
345 present_state ≤ poll_x;
346 end if;
347
```

```
348
349 when grant_x =>
350 if (y_writex = '1') then
351 xdata_out ≤ stim_ydata;
352 elsif (writex = '1') then
353 xdata_out ≤ stim_xdata;
354 end if;
355 if (x_reqx = '1' or y_reqx='1') then
356 present_state ≤ grant_x;
357 else
358 present_state ≤ poll_y;
359 end if;
360 gntx ≤ '1';
361
362
363 when grant_y =>
364 if (x_writey = '1') then
365 ydata_out ≤ stim_xdata;
366 elsif (writey = '1') then
367 ydata_out ≤ stim_ydata;
368 end if;
369 if (x_reqy = '1' or y_reqy = '1') then
370 present_state ≤ grant_y;
371 else
372 present_state ≤ poll_x;
373 end if;
374 gnty ≤ '1';
375
376
377
378 when others =>
379 -- Set the default state and outputs.
380 present_state ≤ poll_x;
381 gntx ≤ '0';
382 gnty ≤ '0';
383 xdata_out ≤ "00000000";
384 ydata_out ≤ "00000000";
385 end case;
386 end if;
387 end process;
388
389 end behavioral;
```

- VHDL code for *control.vhd*

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity control is
6  port(
7  enable : OUT std_logic;
8  reqx : OUT std_logic;
9  reqy : OUT std_logic;
10 write: OUT std_logic;
11 x_write : OUT std_logic;
```

```
y_write : OUT std_logic;
available : IN std_logic;
clk : IN std_logic;
data_in : IN std_logic_vector(7 DOWNTO 0);
gntx : IN std_logic;
gnty : IN std_logic;
reset : IN std_logic;
header_LSB : IN std_logic

);
end control;


architecture behavior of control is

-- Behaviour follows the 'classic' state machine method
-- Possible states.

type states is (poll_fifo, raise_enable, check_data, setup_x, setup_y, ...
    data_xfer, data_yfer);

-- Present state.

signal present_state : states;

begin

-- Main process.

process (clk, reset)

begin

-- Activities triggered by asynchronous reset (active low).

if (reset = '0') then

-- Set the default state and outputs.

present_state <= poll_fifo;
enable <= '0';
reqx <= '0';
reqy <= '0';
write <= '0';
x_write <= '0';
y_write <= '0';


elsif (clk'event and clk = '1') then

-- Set the default state and outputs.

present_state <= poll_fifo;
enable <= '0';
reqx <= '0';
reqy <= '0';
write <= '0';
x_write <= '0';
y_write <= '0';
```

```
70
71  case present_state is
72
73  when poll_fifo =>
74  if(available='1')then
75  present_state ≤ raise_enable;
76  else
77  present_state ≤ poll_fifo;
78  end if;
79
80
81
82  when raise_enable =>
83  enable ≤ '1';
84  present_state ≤ check_data;
85
86
87
88  when check_data =>
89  enable ≤ '0';
90  if(data_in(0)='0') then
91  present_state ≤ setup_x;
92  elsif(data_in(0) = '1') then
93  present_state ≤ setup_y;
94  end if;
95
96
97  when setup_x =>
98  reqx≤'1';
99  if(gntx='1')then
100 present_state≤data_xfer;
101 else
102 present_state≤setup_x;
103 end if;
104
105
106
107 when setup_y =>
108 reqy≤'1';
109 if(gnty='1')then
110 present_state≤data_yfer;
111 else
112 present_state≤setup_y ;
113 end if;
114
115
116
117 when data_xfer =>
118 reqx≤'1';
119 write ≤ '1';
120 y_write≤'1';
121 enable≤'1';
122 if(data_in = "11111111")then
123 enable≤'0';
124 present_state ≤ poll_fifo;
125 else
126 present_state ≤ data_xfer;
127 end if;
128
```

```
129
130 when data_yfer =>
131 reqy<='1';
132 write <= '1';
133 x_write<='1';
134 enable <= '1';
135 if(data_in="11111111")then
136 enable<='0';
137 present_state<poll_fifo;
138 else
139 enable<='1';
140 present_state<data_yfer;
141 end if;
142
143
144 when others =>
145 present_state <= poll_fifo;
146 enable <= '0';
147 reqx <= '0';
148 reqy <= '0';
149 x_write <= '0';
150 y_write <= '0';
151
152
153
154 end case;
155
156 end if;
157
158 end process;
159
160 end behavior;
```

- VHDL code of *top_tb.vhd* for enabling to transmitte simutaneously

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all; -- for internal counter etc.
4
5  -- Uncomment the following library declaration if using
6  -- arithmetic functions with Signed or Unsigned values
7  --use IEEE.NUMERIC_STD.ALL;
8
9  -- Uncomment the following library declaration if instantiating
10 -- any Xilinx leaf cells in this code.
11 --library UNISIM;
12 --use UNISIM.VComponents.all;
13
14 entity top_tb is
15 end top_tb;
16
17 architecture behavioral of top_tb is
18 component control
19 PORT(
20 enable : OUT std_logic;
21 reqx : OUT std_logic;
```

```
reqy : OUT std_logic;
write: OUT std_logic;
x_write : OUT std_logic;
y_write : OUT std_logic;
available : IN std_logic;
clk : IN std_logic;
data_in : IN std_logic_vector(7 DOWNTO 0);
gntx : IN std_logic;
gnty : IN std_logic;
reset : IN std_logic;
header_LSB : IN std_logic
);

end component;
--INPUT
signal clk: std_logic;
signal r_n: std_logic;
signal stim_xdata: std_logic_vector(7 downto 0);
signal stim_ydata: std_logic_vector(7 downto 0);
signal stim_xavail: std_logic :='0';
signal stim_yavail: std_logic :='0';
signal gntx: std_logic :='0';
signal gnty: std_logic :='0';
--OUTPUT
signal stim_xenable: std_logic :='0';
signal stim_yenable: std_logic :='0';
signal writex: std_logic :='0';
signal writey: std_logic :='0';
signal x_writex: std_logic :='0';
signal x_writey: std_logic :='0';
signal y_writex: std_logic :='0';
signal y_writey: std_logic :='0';
signal x_reqx: std_logic :='0';
signal x_reqy: std_logic :='0';
signal y_reqx: std_logic :='0';
signal y_reqy: std_logic :='0';
signal header_LSB: std_logic :='0';
--    signal data_out: std_logic_vector(7 downto 0);

signal xdata_out: std_logic_vector(7 downto 0);
signal ydata_out: std_logic_vector(7 downto 0);
--VARIABLE
----stim_xbar.vhd
signal internalclock : std_logic; -- internal clock
signal dummy : std_logic; -- dummy signal
----output.vhd
type states is (poll_x, poll_y, grant_x, grant_y);
-- Present state.
signal present_state_x : states;
signal present_state_y : states;


begin

controlx:control port map(
--Input
clk=>clk,
reset=>r_n,
data_in=>stim_xdata,
```

```
81  available=>stim_xavail,
82  gntx=>gntx,
83  gnty=>gnty,
84  --Output
85  enable=>stim_xenable,
86  write=>writex,
87  x_write=>x_writey,
88  reqx=>x_reqx,
89  reqy=>x_reqy,
90  header_LSB=>header_LSB
91  );
92  controly:control port map(
93  --Input
94  clk=>clk,
95  reset=>r_n,
96  data_in=>stim_ydata,
97  available=>stim_yavail,
98  gntx=>gntx,
99  gnty=>gnty,
100 --Output
101 enable=>stim_yenable,
102 write=>writey,
103 y_write=>y_writex,
104 reqx=>y_reqx,
105 reqy=>y_reqy,
106 header_LSB=>header_LSB
107 );
108
109 resetgen : process
110 begin -- process resetgen
111 r_n ≤ '0';
112 wait for 2450 ns;
113 r_n ≤ '1';
114 -- Add more entries if required at this point !
115
116 wait until dummy'event;
117 end process resetgen;
118
119 -- purpose: Generates the internal clock source. This is a 50% duty
120 -- cycle clock source, period 1000ns, with the first half of
121 -- the cycle being logic '0'.
122 --
123 -- outputs: internalclock
124 clockgen : process
125 begin -- process clockgen
126 internalclock ≤ '0';
127 wait for 500 ns;
128 internalclock ≤ '1';
129 wait for 500 ns;
130 end process clockgen;
131 -- purpose: Generates the stimuli for the data, using a counter based
132 -- approach which is much cleaner than explicit timings
133 --
134 -- outputs: stim_inst
135 -- stim_a
136 -- stim_b
137
138
139 --  For testing the simutaneous transmission, you need to run two datagen ...
```

```
    processes and comment the other two processes
--  For testing the input broadcast, you need to run a datagen process at a ...
    time, and comment the other three processes
-- x -> x
datagen1 : process
-- Since we're going to output a number of different test
-- data we need an internal counter to keep track of things.

variable count : unsigned (5 downto 0) := "000000";
begin -- process datagen1
wait until internalclock'event and internalclock = '1';
count := count + 1;
if count = 4 then -- simple data transfer x -> x
stim_xavail <= '1'; -- data flag raised
wait until internalclock'event and internalclock = '1'
and stim_xenable = '1';

stim_xdata <= "10000010"; -- header word pushed when system
stim_xavail <= '0'; -- requests transfer, and flag back down

wait until internalclock'event and internalclock = '1'
and stim_xenable = '1';

stim_xdata <= "10000010";

wait until internalclock'event and internalclock = '1'
and stim_xenable = '1';

stim_xdata <= "11100111"; -- data word pushed

wait until internalclock'event and internalclock = '1'
and stim_xenable = '1';

stim_xdata <= "11111111"; -- end word pushed
elsif count = 5 then    -- NO OPERATION
stim_xavail <= '0';
stim_xdata <= "00000000";
else  -- NO OPERATION
stim_xavail <= '0';
stim_xdata <= "00000000";
end if;
end process datagen1;

-- x -> y
datagen2 : process
-- Since we're going to output a number of different test
-- data we need an internal counter to keep track of things.

variable count : unsigned (5 downto 0) := "000000";
begin -- process datagen1
wait until internalclock'event and internalclock = '1';
count := count + 1;
if count = 4 then -- simple data transfer x -> y
stim_xavail <= '1'; -- data flag raised
wait until internalclock'event and internalclock = '1'
and stim_xenable = '1';

stim_xdata <= "00000001"; -- header word pushed when system
stim_xavail <= '0'; -- requests transfer, and flag back down
```

```
197
198 wait until internalclock'event and internalclock = '1'
199 and stim_xenable = '1';
200 stim_xdata ≤ "00000001";
201
202 wait until internalclock'event and internalclock = '1'
203 and stim_xenable = '1';
204 stim_xdata ≤ "10101011"; -- data word pushed
205
206 wait until internalclock'event and internalclock = '1'
207 and stim_xenable = '1';
208
209 stim_xdata ≤ "11111111"; -- end word pushed
210 elsif count = 5 then    -- NO OPERATION
211 stim_xavail ≤ '0';
212 stim_xdata ≤ "00000000";
213 else  -- NO OPERATION
214 stim_xavail ≤ '0';
215 stim_xdata ≤ "00000000";
216 end if;
217 end process datagen2;
218
219 -- y -> x
220 datagen3 : process
221 -- Since we're going to output a number of different test
222 -- data we need an internal counter to keep track of things.
223
224 variable count : unsigned (5 downto 0) := "000000";
225 begin -- process datagen1
226 wait until internalclock'event and internalclock = '1';
227 count := count + 1;
228 if count = 4 then -- simple data transfer y -> x
229 stim_yavail ≤ '1'; -- data flag raised
230 wait until internalclock'event and internalclock = '1'
231 and stim_yenable = '1';
232
233 stim_ydata ≤ "10000010"; -- header word pushed when system
234 stim_yavail ≤ '0'; -- requests transfer, and flag back down
235
236
237 wait until internalclock'event and internalclock = '1'
238 and stim_yenable = '1';
239
240 stim_ydata ≤ "10000010"; -- header word pushed when system
241
242 wait until internalclock'event and internalclock = '1'
243 and stim_yenable = '1';
244
245 stim_ydata ≤ "10111010"; -- data word pushed
246
247 wait until internalclock'event and internalclock = '1'
248 and stim_yenable = '1';
249
250 stim_ydata ≤ "11111111"; -- end word pushed
251 elsif count = 5 then    -- NO OPERATION
252 stim_yavail ≤ '0';
253 stim_ydata ≤ "00000000";
254 else  -- NO OPERATION
255 stim_yavail ≤ '0';
```

```
256 stim_ydata ≤ "00000000";
257 end if;
258 end process datagen3;
259
260
261
262 -- y -> y
263 datagen4 : process
264 -- Since we're going to output a number of different test
265 -- data we need an internal counter to keep track of things.
266
267 variable count : unsigned (5 downto 0) := "000000";
268 begin -- process datagen1
269 wait until internalclock'event and internalclock = '1';
270 count := count + 1;
271 if count = 4 then -- simple data transfer y -> y
272 stim_yavail ≤ '1'; -- data flag raised
273 wait until internalclock'event and internalclock = '1'
274 and stim_yenable = '1';
275
276 stim_ydata ≤ "00000001"; -- header word pushed when system
277 stim_yavail ≤ '0'; -- requests transfer, and flag back down
278
279 wait until internalclock'event and internalclock = '1'
280 and stim_yenable = '1';
281 stim_ydata ≤ stim_ydata; -- header word pushed again
282
283 wait until internalclock'event and internalclock = '1'
284 and stim_yenable = '1';
285
286 stim_ydata ≤ "10101010"; -- data word pushed
287
288 wait until internalclock'event and internalclock = '1'
289 and stim_yenable = '1';
290
291 stim_ydata ≤ "11111111"; -- end word pushed
292
293
294 elsif count = 5 then    -- NO OPERATION
295 stim_yavail ≤ '0';
296 stim_ydata ≤ "00000000";
297 else  -- NO OPERATION
298 stim_yavail ≤ '0';
299 stim_ydata ≤ "00000000";
300 end if;
301 end process datagen4;
302
303
304
305
306 -- ***********************************
307
308
309 -- now drive the output clock, this is simply the internal clock
310 -- nb: could also invert the clock if desired to allow for signals
311 -- to be generated pseudo-asynchronously
312
313 clk ≤ internalclock;
314
```

```
315 --output.vhd
316 --  x-output
317 x_output: process (clk, r_n)
318 begin
319 -- Activities triggered by asynchronous reset (active low).
320 if (r_n = '0') then
321 -- Set the default state and outputs.
322 present_state_x ≤ poll_x;
323 gntx ≤ '0';
324 xdata_out ≤ "00000000";
325 elsif (clk'event and clk = '1') then
326 -- Set the default state and outputs.
327 present_state_x ≤ poll_x;
328 gntx ≤ '0';
329 xdata_out ≤ "00000000";
330 case present_state_x is
331
332 when poll_x =>
333 if (x_reqx = '1' or y_reqx = '1') then
334 present_state_x ≤ grant_x;
335 else
336 present_state_x ≤ poll_x;
337 end if;
338
339 when grant_x =>
340 if (y_writex = '1') then
341 xdata_out ≤ stim_ydata;
342 elsif (writex = '1') then
343 xdata_out ≤ stim_xdata;
344 end if;
345 if (x_reqx = '1' or y_reqx='1') then
346 present_state_x ≤ grant_x;
347 else
348 present_state_x ≤ poll_x;
349 end if;
350 gntx ≤ '1';
351
352 when others =>
353 -- Set the default state and outputs.
354 present_state_x ≤ poll_x;
355 gntx ≤ '0';
356 xdata_out ≤ "00000000";
357 end case;
358 end if;
359 end process;
360 --  y-output
361 y_output: process (clk, r_n)
362 begin
363 -- Activities triggered by asynchronous reset (active low).
364 if (r_n = '0') then
365 -- Set the default state and outputs.
366 present_state_y ≤ poll_x;
367 gnty ≤ '0';
368 ydata_out ≤ "00000000";
369 elsif (clk'event and clk = '1') then
370 -- Set the default state and outputs.
371 present_state_y ≤ poll_y;
372 gnty ≤ '0';
373 ydata_out ≤ "00000000";
```

```
374 case present_state_y is
375
376
377 when poll_y =>
378 if (x_reqy = '1' or y_reqy = '1') then
379 present_state_y ≤ grant_y;
380 else
381 present_state_y ≤ poll_y;
382 end if;
383
384
385
386 when grant_y =>
387 if (x_writey = '1') then
388 ydata_out ≤ stim_xdata;
389 elsif (writey = '1') then
390 ydata_out ≤ stim_ydata;
391 end if;
392 if (x_reqy = '1' or y_reqy = '1') then
393 present_state_y ≤ grant_y;
394 else
395 present_state_y ≤ poll_y;
396 end if;
397 gnty ≤ '1';
398
399 when others =>
400 -- Set the default state and outputs.
401 present_state_y ≤ poll_y;
402 gnty ≤ '0';
403 ydata_out ≤ "00000000";
404 end case;
405 end if;
406 end process;
407
408 end behavioral;
```

- VHDL code of *control.vhd* for enabling to broadcast

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4
5 entity control is
6 port(
7 enable : OUT std_logic;
8 reqx : OUT std_logic;
9 reqy : OUT std_logic;
10 write: OUT std_logic;
11 x_write : OUT std_logic;
12 y_write : OUT std_logic;
13 available : IN std_logic;
14 clk : IN std_logic;
15 data_in : IN std_logic_vector(7 DOWNTO 0);
16 gntx : IN std_logic;
17 gnty : IN std_logic;
18 reset : IN std_logic;
```

```
19 header_LSB : IN std_logic
20
21 );
22 end control;
23
24
25 architecture behavior of control is
26
27 -- Behaviour follows the 'classic' state machine method
28 -- Possible states.
29
30 type states is (poll_fifo, raise_enable, check_data, setup_x, setup_y, ...
       setup_xy, data_xfer, data_yfer, data_xyfer);
31
32 -- Present state.
33
34 signal present_state : states;
35
36 begin
37
38 -- Main process.
39
40 process (clk, reset)
41
42 begin
43
44 -- Activities triggered by asynchronous reset (active low).
45
46 if (reset = '0') then
47
48 -- Set the default state and outputs.
49
50 present_state ≤ poll_fifo;
51 enable ≤ '0';
52 reqx ≤ '0';
53 reqy ≤ '0';
54 write ≤ '0';
55 x_write ≤ '0';
56 y_write ≤ '0';
57
58
59 elsif (clk'event and clk = '1') then
60
61 -- Set the default state and outputs.
62
63 present_state ≤ poll_fifo;
64 enable ≤ '0';
65 reqx ≤ '0';
66 reqy ≤ '0';
67 write ≤ '0';
68 x_write ≤ '0';
69 y_write ≤ '0';
70
71 case present_state is
72
73 when poll_fifo =>
74 if(available='1')then
75 present_state ≤ raise_enable;
76 else
```

```
77  present_state ≤ poll_fifo;
78  end if;
79
80
81
82  when raise_enable =>
83  enable ≤ '1';
84  present_state ≤ check_data;
85
86
87
88  when check_data =>
89  enable ≤ '0';
90  if data_in(7)='1' then
91  present_state ≤ setup_xy;
92  elsif(data_in(0)='0') then
93  present_state ≤ setup_x;
94  elsif(data_in(0) = '1') then
95  present_state ≤ setup_y;
96  end if;
97
98
99  when setup_x =>
100 reqx≤'1';
101 if(gntx='1')then
102 present_state≤data_xfer;
103 else
104 present_state≤setup_x;
105 end if;
106
107
108
109 when setup_y =>
110 reqy≤'1';
111 if(gnty='1')then
112 present_state≤data_yfer;
113 else
114 present_state≤setup_y ;
115 end if;
116
117 when setup_xy =>
118 reqy≤'1';
119 reqx≤'1';
120 if(gnty='1' and gntx='1')then
121 present_state≤data_xyfer;
122 else
123 present_state≤setup_xy ;
124 end if;
125
126 when data_xfer =>
127 reqx≤'1';
128 write ≤ '1';
129 y_write≤'1';
130 enable≤'1';
131 if(data_in = "11111111")then
132 enable≤'0';
133 present_state ≤ poll_fifo;
134 else
135 present_state ≤ data_xfer;
```

```
136  end if;
137
138
139  when data_yfer =>
140  reqy<='1';
141  write <= '1';
142  x_write<='1';
143  enable <= '1';
144  if(data_in="11111111")then
145  enable<='0';
146  present_state<poll_fifo;
147  else
148  enable<='1';
149  present_state<data_yfer;
150  end if;
151
152  when data_xyfer =>
153  reqy<='1';
154  reqx<='1';
155  write <= '1';
156  x_write<='1';
157  y_write<='1';
158  enable <= '1';
159  if(data_in="11111111")then
160  enable<='0';
161  present_state<poll_fifo;
162  else
163  enable<='1';
164  present_state<data_xyfer;
165  end if;
166
167
168  when others =>
169  present_state <= poll_fifo;
170  enable <= '0';
171  reqx <= '0';
172  reqy <= '0';
173  x_write <= '0';
174  y_write <= '0';
175
176
177
178  end case;
179
180  end if;
181
182  end process;
183
184  end behavior;
```