Experiment E4303
Asynchronous Sequential Logic and VHDL

Sven Richter

September 9, 2004

Contents

1 Introduction

The aim of this practical is to reinforce the Asynchronous Sequential Logic design procedures and to provide some experience at realizing these circuits in CPLD.

Task

Part 1: Design the sequential logic circuit for a digital phase modulater or digital single sideband mixer, with the waveform shown in figure 1. Ensure that all possible transitions are specified. (The frequency of signal A is much higher than that of signal B.)

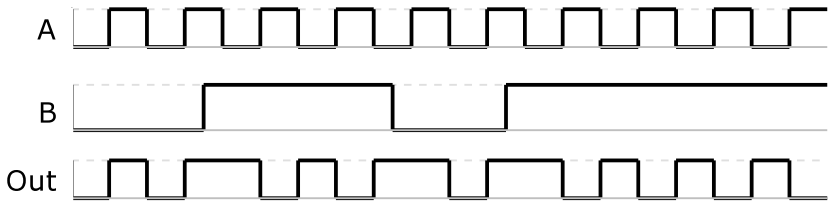

Figure 1: Digital Mixer waveforms.

Realize the circuit using VHDL, fully test the circuit by observing the waveforms.

Part 2: Write the VHDL code for a divide by 100 circuit and test your code by observing the waveforms. Combine the code for these two circuits into one VHDL module, such that the Output from the digital mixer is used as clock (ie input) for the divider. This combined logic can then form the basis of a PLL circuit.

2 Design

To design the digital phase modulator it is necessary to regard all states and transitions the circuits can change to. Therefore, the waveform from figure 1 must be extended to figure 2. In the new waveform all states and transition are noted to which the circuit could change. To fill them in, reasonable assumptions where made.

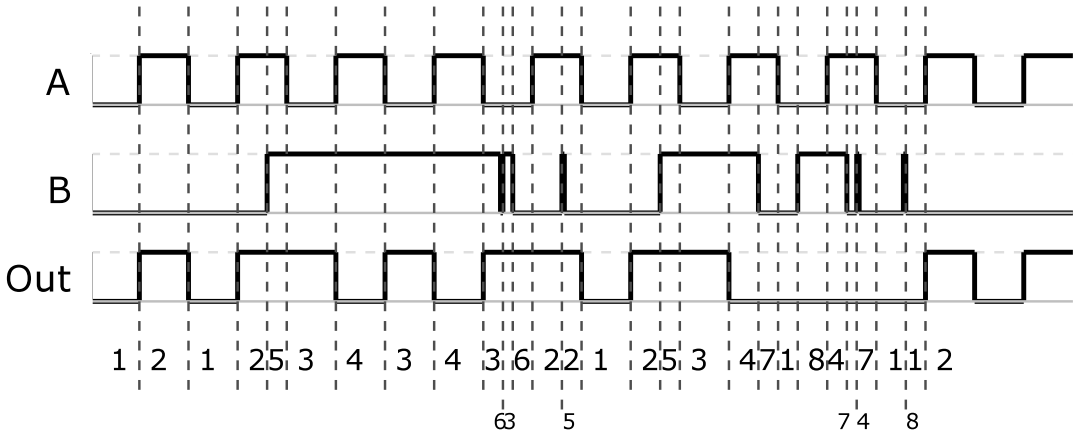

Figure 2: Digital Mixer waveforms with states.

AB									
00		01		11		10		00	Out
1	↔	8	→	4	↔	7	→	1	0
			↗↙				↗↙		
6	↔	3	←	5	↔	2	←	6	1

Table 1: Logic Signal Flow Graph

With all states defined is is possible to create the Logic Signal Flow Graph which is shown in table 1. Here all stables states occur and the transition between them is marked by an arrow.

From this Flow Graph the Primitive Flow Table can be developed. It is

shown in table 2.

		AB		
00	01	11	10	Out
1	8		2	0
1		5	**2**	1
6	**3**	4		1
	3	**4**	7	0
	3	**5**	2	1
6	3		2	1
1		4	**7**	0
1	**8**	4		0

Table 2: Primitive Flow Table

The Bold numbers, in the boxes, in the Primitive Flow Table are stable states. All other numbers describe transitional states and unfilled cells are don't cares.

In this table it is easy to locate states which can be merged. As a result the merger Diagram (figure 3) is synthesized. In the Merger Diagram states that can be merged are connected by a line.

Therefore, two possible ways to merge the states exist:

1. 1 with 2, 5 with 6, 3 with 4, 7 with 8
2. 2 with 5, 6 with 3, 4 with 7, 8 with 1

The second merging possibility has much easier output coding but will lead to much more complicated boolean expressions. Therefore the first one will be used for all further design steps.

The results from the second merging possibility are not shown here to keep clearness.

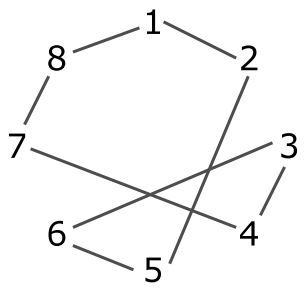

Figure 3: Merger Diagram

		AB		
00	01	11	10	Out
1	8	5	**2**	0/1
6	**3**	**4**	7	1/0
6	3	**5**	2	1
1	**8**	4	**7**	0

Table 3: Merged flow table

The resulting Merged flow table is shown in table 3. In this table all stable states also marked bold and got a frame. All other states are transitional states and don't cares do not exist.

But this table contains a critical race condition in column three (AB = 11) and four (AB = 10). For example if the circuit is currently in the stable state eight and the input changes to "11" it will change to the transitional state four but to reach the stable state four it would have to go over the stable state five or the transitional state five. The similar problem occurs by transition from state two to state five and so on.

To solve this problem and to remove the critical race condition it is required to swap row two with row three. The resulting Merged flow table without

the critical races is shown in table 4.

		AB		
00	01	11	10	Out
1	8	5	**2**	0/1
6	3	**5**	2	1
6	**3**	**4**	7	1/0
1	**8**	4	**7**	0

Table 4: Merged flow table without critical race

From the Merged Flow table the Excitation Matrix (table 5) is constructed. Two output variables f and g are introduced because the Merged Flow table consists of four rows so a two bit value is needed.

All stable states are replaced by the corresponding value of the output variables. Therefore the values for the transitional states is known too and is replaced.

		AB		
00	01	11	10	fg
00	10	01	**00**	00
01	11	**01**	00	01
01	**11**	**11**	10	11
00	**10**	11	**10**	1 0

Table 5: Excitation Matrix

From the Excitation Matrix can the Karnaugh maps for F (table 6) and G (table 6) be build.

	AB			
00	01	11	10	fg
0	1	0	0	00
0	1	0	0	01
0	1	1	1	11
0	1	1	1	10

Table 6: Karnaugh map for F

	AB			
00	01	11	10	fg
0	0	1	0	00
1	1	1	0	01
1	1	1	0	11
0	0	1	0	10

Table 7: Karnaugh map for G

Out of the Karnaugh maps the boolean expressions for F and G are calculated as followed

$$F = \overline{A}B \vee Af \quad (1)$$

$$G = \overline{A}g \vee AB \quad (2)$$

	AB			
00	01	11	10	fg
0	0	1	1	00
1	1	1	1	01
1	1	0	0	11
0	0	0	0	10

Table 8: Karnaugh map for out

Because, of the choice for the first possibility to merge, it is also needed to calculate the output value "out" of the Inputs A and B and the values of f and g. The Karnaugh map for "out" is shown in table 8.

This resulting expression is:

$$out = \overline{A}g \vee A\overline{f} \quad (3)$$

To build the circuit with NAND's it is needed to solve equations 1, 2 and 3 as bellow:

$$F = \overline{\overline{\overline{A}B} \wedge \overline{Af}} \quad (4)$$

$$G = \overline{\overline{\overline{A}g} \wedge \overline{AB}} \quad (5)$$

$$out = \overline{\overline{\overline{A}g} \wedge \overline{A\overline{f}}} \quad (6)$$

Now the network can be created and it is shown in figure 4.

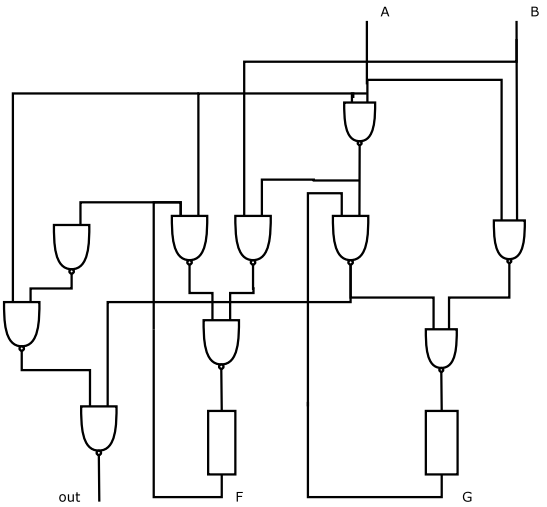

Figure 4: Circuit

To prevent static hazards in the circuit it would be better to derive a little bit more complex expression for F,G and out for a practical realization.

$$F = \overline{A}B \vee Af \vee Bf \quad (7)$$

$$G = \overline{A}g \vee AB \vee Bg \quad (8)$$

$$out = \overline{A}g \vee A\overline{f} \vee \overline{f}g \quad (9)$$

But because of the VHDL compiler which will just remove this additional terms it is easier use the simple one's to implement in VHDL and it will not change the results of the testing.

3 Realization and testing

3.1 Part A

The circuit was realized in VHDL and the sourcecode can be found at section A. All Boolean equations for F and G are programed in the mixer module (section A.1). To generate the expected output it is necessary to implement also the Boolean algebraic expression for "out", this is done in module circmixer (section A.3).

In the version of the circmixer module provided in the appendix the divide100 module (section A.2) is used too and is connected to the output of the phase-mixer. But this will be discussed in Part 2.

To test the mixer module it was neccessary to add an additional signal "rst" on the circuit, to reset the initial values. This is not needed if the program is written into an real processor but mandatory for testing.

A abel test vector was created to test the mixer module. This vector did contain all possible transitions to produce an error message in the simulation window if any expression are not correct.

```
Test_vectors
([siga,sigb,rst] -> [sigout])
[0,0,0] ->[0]; //6
[1,0,1] ->[1]; //2
[0,0,1] ->[0]; //1
.....
[0,0,1] ->[1]; //6
[1,0,1] ->[1]; //2
END
```

The test did run without any errors and the result is shown figure 5 and figure 6 is the resulting waveform.

Additionally, I generated a waveform similar to figure 1 to compare the designed circuit with the requirements expressed in the task formulation. Figure 7 does show this one and the result of the test. They are both similar.

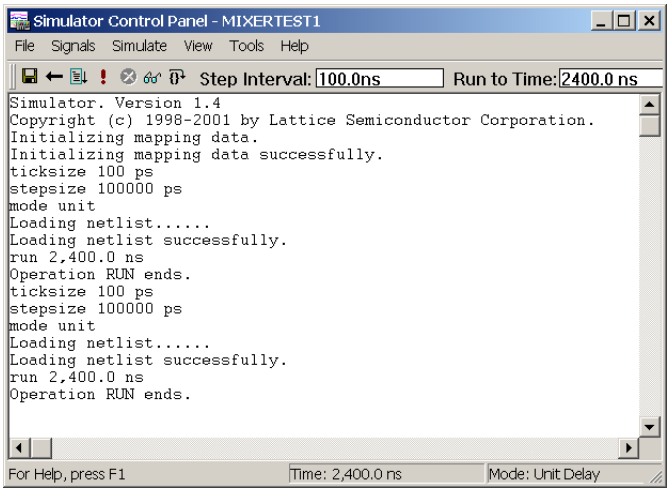

Figure 5: Test-Vector result

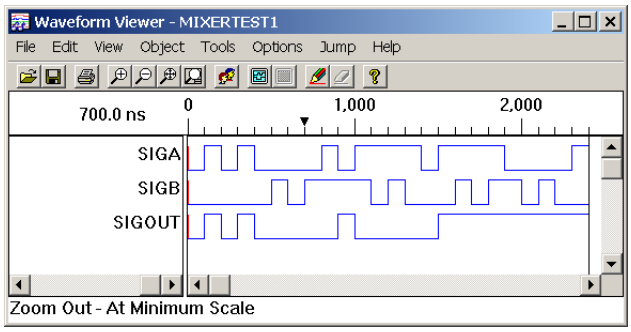

Figure 6: Test-Vector waveform

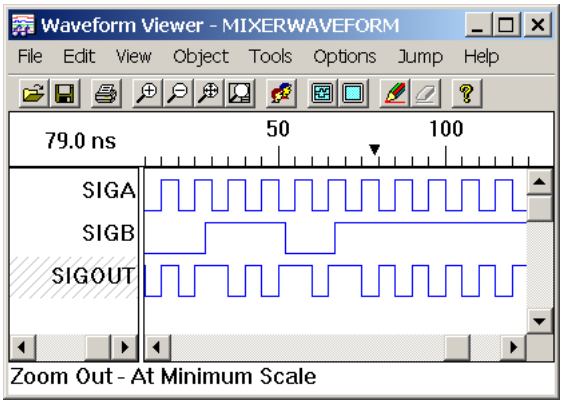

Figure 7: Waveform

3.2 Part B

The VHDL code for the the divide by 100 circuit is attached in section A.2. It is a simple counter which counts fifty clock pulses and switches then the output. It was tested completely and behaved like expected. The results can be seen in figure 8, 9 and 10.

Because there was no EPLD test board available as the code was finished it was not possible to test it on an real device. But If it can be assumed that the signal A, like described in the task formulation, has a very high frequency compared to signal B the circuit will end up with a nearly stable clock signal. Only if signal B starts to change very often it can change the frequency of the output signal more significant.

A simulation of the complete program was done and the results are shown in figure 11 and 12.

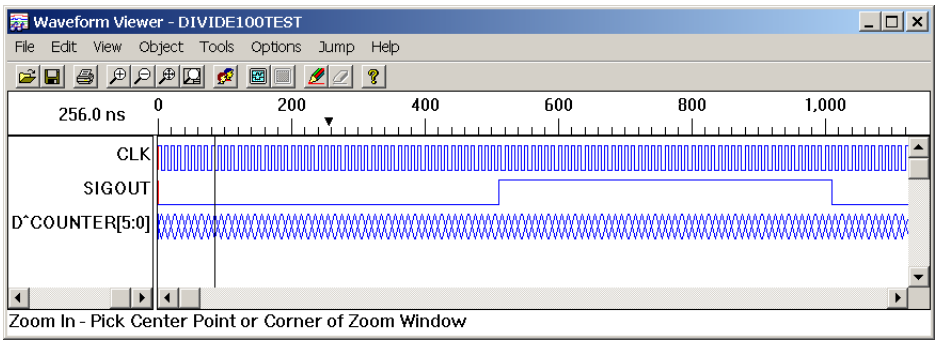

Figure 8: Divider waveform - full

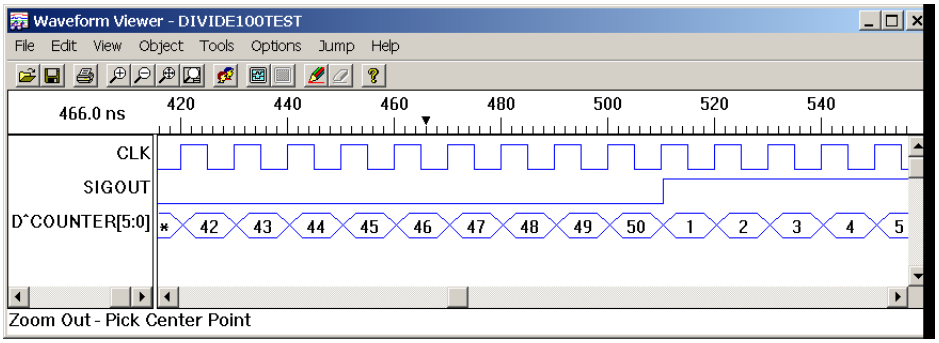

Figure 9: Divider waveform - begin

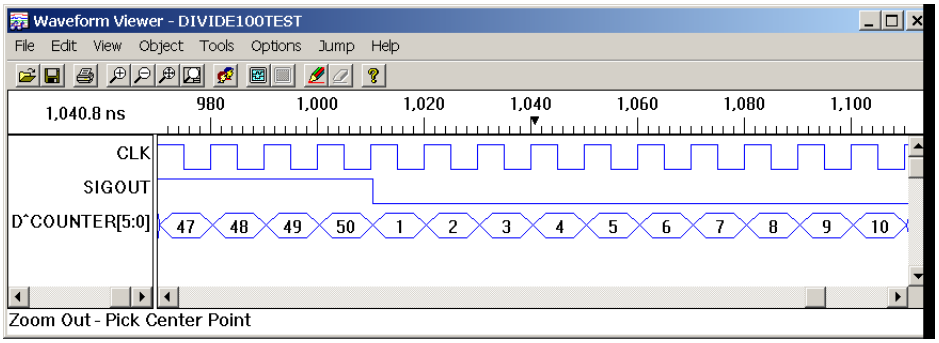

Figure 10: Divider waveform - end

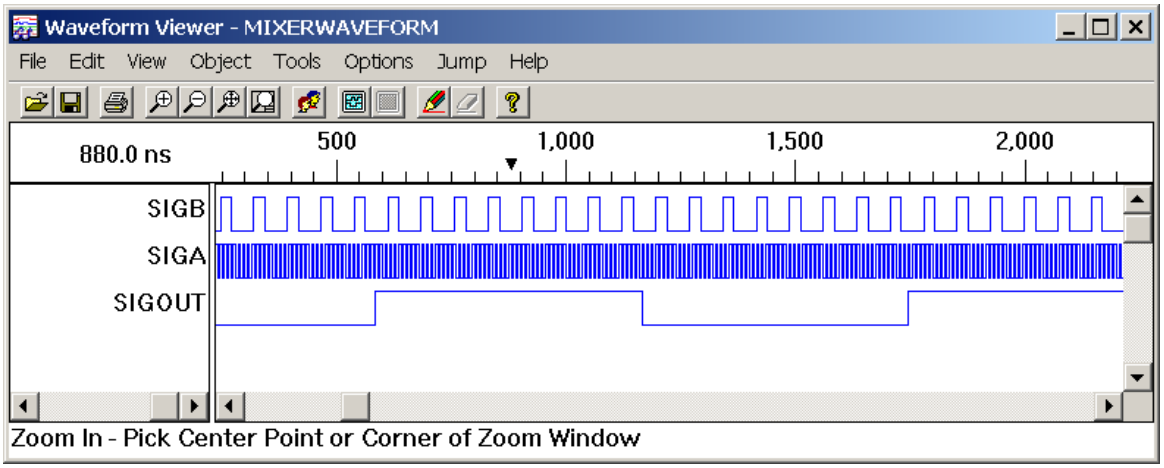

Figure 11: Divider + mixer waveform - full

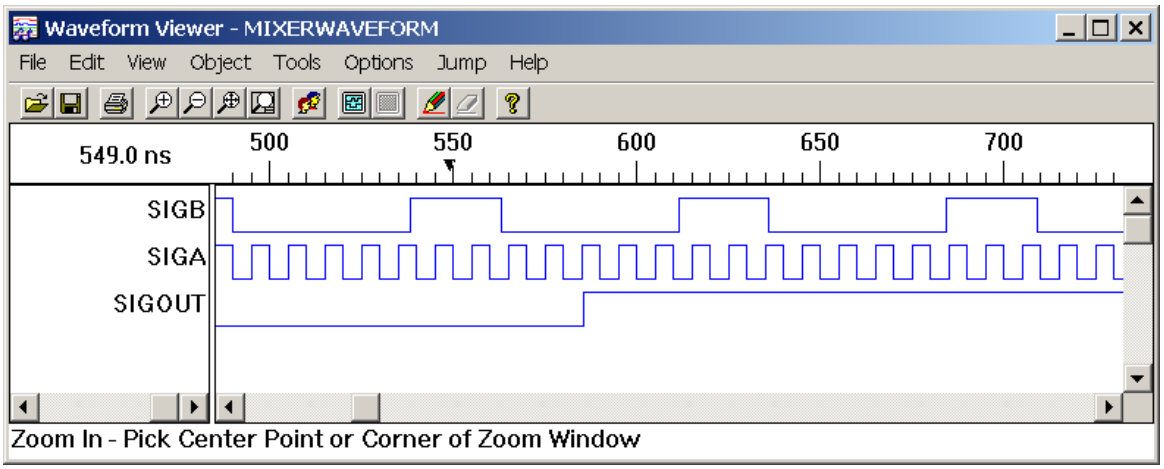

Figure 12: Divider + mixer waveform - center

A Sourcecode

A.1 mixer

```
1  -- Experiment E4303
2  --
3  --    file: mixer.vhd
4  --     author: Sven Richter
5  --
6  --    digital phase mixer
7  --
8
9
10 library ieee;
11 use ieee.std_logic_1164.all;
12
13 entity mixer is
14    port (A,B,rst :in std_logic;
15    -- rst is just to be able to simulate
16        Fn,Gn :buffer std_logic
17    );
18
19 end;
20
21 architecture mixer_arc of mixer is
22
23    attribute syn_keep : boolean;
24    signal F,G : std_logic := '0';
25    attribute syn_keep of F,G : signal is true;
26 begin
27    run: process(A, B, F, G) begin
28           Fn <= ((not A) and B) or (A and f);
29           Gn <= ((not A) and G) or (A and B);
30    end process run;
31
32    outrun: process (Fn, Gn) begin
33       F <= Fn and rst;
34       G <= Gn and rst ;
35    end process outrun;
```

```
36
37 end mixer_arc;
38
```

A.2 divide100

```
1 -- Experiment E4303
2 --
3 --    file: divide100.vhd
4 --     author: Sven Richter
5 --
6 --    Divider 1 by 100
7 --
8
9 library ieee;
10 use ieee.std_logic_1164.all;
11
12 entity divide100 is
13     port( clk :in std_logic;
14         sigout :inout std_logic
15     );
16 end;
17
18 architecture divide100_arc of divide100 is
19     signal counter : integer range 1 to 50;
20 begin
21
22     run: process begin
23         wait until rising_edge(clk);
24         if (counter < 50) then
25             counter <= counter + 1;
26         else
27             counter <= 1;
28             sigout <= not sigout;
29         end if ;
30
31     end process run;
32
33 end divide100_arc;
```

34

A.3 circmixer

```
1 -- Experiment E4303
2 --
3 --    file: circmixer.vhd
4 --     author: Sven Richter
5 --
6 --    the complete circus to
7 --    do the asynchron logic
8
9
10 library ieee;
11 use ieee.std_logic_1164.all;
12
13 entity circmixer is
14    port (siga, sigb,rst : in std_logic;
15        sigout : inout std_logic
16    );
17
18    attribute loc : string;
19    attribute loc of rst: signal is "P16";
20    attribute loc of siga: signal is "P15";
21    attribute loc of sigb: signal is "P17";
22    attribute loc of sigout: signal is "P29";
23
24 end;
25
26 architecture circmixer_arc of circmixer is
27    component  mixer
28        port (A,B,rst :in std_logic;
29            Fn,Gn :buffer std_logic
30        );
31    end component;
32
33    component divide100
34        port( clk :in std_logic;
35            sigout :inout std_logic
```

```
36         );
37     end component;
38 
39     signal sigf, sigg, sigmixer : std_logic ;
40 
41 begin
42     m0: mixer port map (siga, sigb, rst , sigf, sigg);
43 
44     d0: divide100 port map (sigmixer, sigout);
45 
46     run: process (sigf, sigg) begin
47         sigmixer <= ((not siga) and sigg) or ((not sigf) and siga);
48     end process run;
49 
50 
51 end circmixer_arc;
52 
```