

Fachhochschule Dortmund, University of Applied Sciences and Arts – Summer Semester 2020

Microelectronics & HW/SW-Co-Design

Dortmund, 04/10/2020

ASSIGNMENT REPORT

AUDIO MIXER

TEAM MEMBER:	Truong Hoang Xuan	7207032
	Bheeshmaraya Bheemasamudra Vruksharaj	7206909
	Mukesh Gowda Channarayapatna Nanjegowda	7206931

Contents

1	INTRODUCTION	2
1.1	Hardware Description Language	2
1.2	Field Programmable Gate Arrays	2
1.3	MIXER: What is it? How does it work?	2
2	IMPLEMENTATION	3
2.1	Tools	3
2.1.1	ModelSim PE Student Edition 10.4a	3
2.1.2	Lattice Diamond 3.11	3
2.1.3	Microsoft® Excel® for Office 365	3
2.1.4	GNU Octave, version 5.2.0	3
2.1.5	Code::Blocks 17.12	3
2.2	VHDL implementation for MIXER	4
2.2.1	System behavior	5
2.2.2	Gain - loss calculation	6
2.2.3	Saturation algorithm	7
2.2.4	Clock calculation and auxiliary functions	10
3	VERIFICATION	11
3.1	Unit test - Test for Gain - loss calculation and limit functions	11
3.2	Unit test - Test for overflow detection	11
3.3	System test – Test circuit behavior	11
3.4	Acceptance test - Test with audio data	12
4	SYNTHESIS	14
4.1	Verify Functionality with Simulation	15
4.2	Location pin setting	16
4.3	Examine result and enhancement	16
4.4	Floor plan and physical view	17
5	CONCLUSION	18
5.1	Gained knowledge	18
5.2	Open points and improvement	18
5.3	Extension	18
6	REFERENCES	19

1 INTRODUCTION

1.1 Hardware Description Language

Hardware description Language (HDL) is a specialized computer language used to describe the structure and behavior of electronic circuits, and most commonly, digital logic circuits.

A hardware description language enables a precise, formal description of an electronic circuit that allows automated analysis and simulation of an electronic circuit. It also allows for the synthesis of an HDL description into a netlist, which can then be placed and routed to produce the set of masks used to create an integrated circuit.

HDLs form an integral part of Electronic Design Automation (EDA) systems, especially for complex circuits, such as application-specific integrated circuits, microprocessors, and programmable logic devices.

1.2 Field Programmable Gate Arrays

Field Programmable Gate Arrays (FPGAs) are semiconductor devices that are based around a matrix of Configurable Logic Blocks (CLBs) connected via programmable interconnects.

The FPGA configuration is generally specified using an HDL, like that used for an Application-Specific Integrated Circuit (ASIC).

FPGAs contain an array of programmable logic blocks, and a hierarchy of "reconfigurable interconnects" that allow the blocks to be "wired together", like many logic gates that can be inter-wired in different configurations. FPGAs can be reprogrammed to desired application or functionality requirements after manufacturing.

1.3 MIXER: What is it? How does it work?

Mixer is an electronic device used for mixing, balancing and combining different sounds and audio signals, sources like microphones, instruments, and synthesizers or previously recorded audio.

A mixer takes various audio sources through its multiple input channels, adjusts levels and other attributes of the sound, then usually combines them to a lesser number of outputs.

This involves taking the audio from performers in a live situation, tweaking and adding effects, then combining these to a stereo or mono output which can be amplified with a PA system. A mixer allows the user to take lots of audio sources and manipulate them before sending them to output channels.

At its core function, an audio mixer takes two or more audio signals, merges them together and provides one or more output signals.

The audio mixer that we have implemented here has four input channels (16 bits) from a Time Division Multiplexing (TDM) where each input channel is multiplied with gain factors (10 bit) for each target. Finally, each input channel summed up after multiplying with gain factor to produce two channels (24 bits) output TDM stream.

2 IMPLEMENTATION

2.1 Tools

This section briefly mentions the tools are mainly used to do this assignment. All most tools are open source or license free for student.

2.1.1 ModelSim PE Student Edition 10.4a

The ModelSim PE Student Edition is used to writing VHDL code and run simulation for testing. The wave forms in the manual test results are taken by this software.

2.1.2 Lattice Diamond 3.11

To synthesize the circuit on FPGA board, this tool is used. The tool provides many features from design, analysis, verification the code until configuration, generate report, download data to the FPGA boards.

2.1.3 Microsoft® Excel® for Office 365

To generate test data automatically, MS Excel is selected. This SW provides many build-in functions for calculation as well as support automation test. After generating a big amount of test data, the manual calculation to compare with result from VHDL seems to be impossible. Hence, a Visual Basic macro is developed on this software to calculate with the generated test data automatically, to ensure that all the calculation with VHDL code is properly.

2.1.4 GNU Octave, version 5.2.0

In order to generate the audio file, GNU Octave is used. This is a free software and very useful for signal processing.

2.1.5 Code::Blocks 17.12

This software is used to writing C++ code, to read and write audio wave file for testing purpose. This is open source and cross-platform IDE.

2.2 VHDL implementation for MIXER

To complete this assignment, the following process is applied.

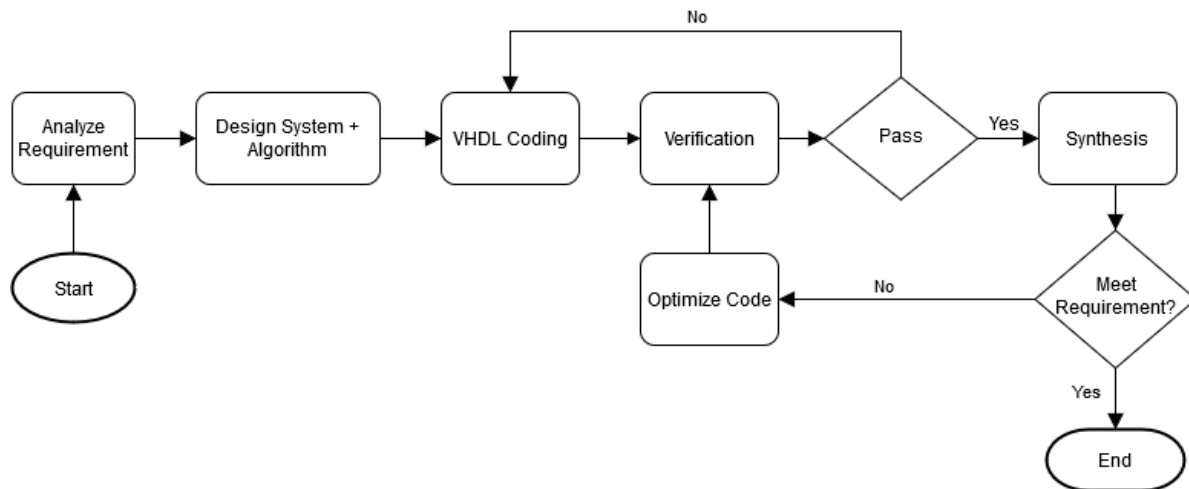


Figure 1: Main process

The outcome is the following files and the functionality for each file in this assignment. They are including VHDL codes as well as test artifacts.



Figure 2: VHDL code and test artifacts

Along with that is the files which are related to audio test. They can be found under `AudioMixer\bin\Release`. The details for this test are mentioned in [3.1.4 Acceptance test - Test with audio data](#).

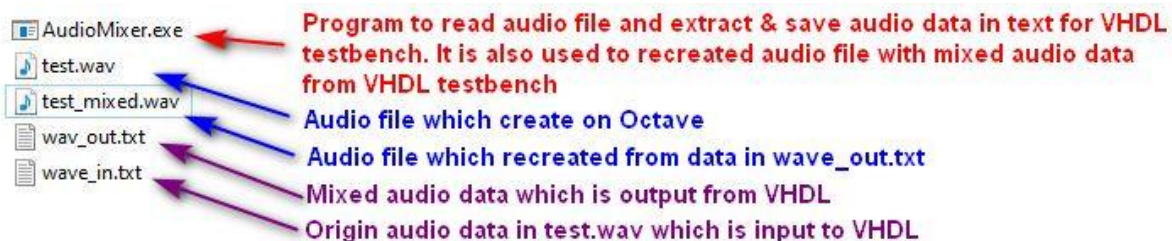


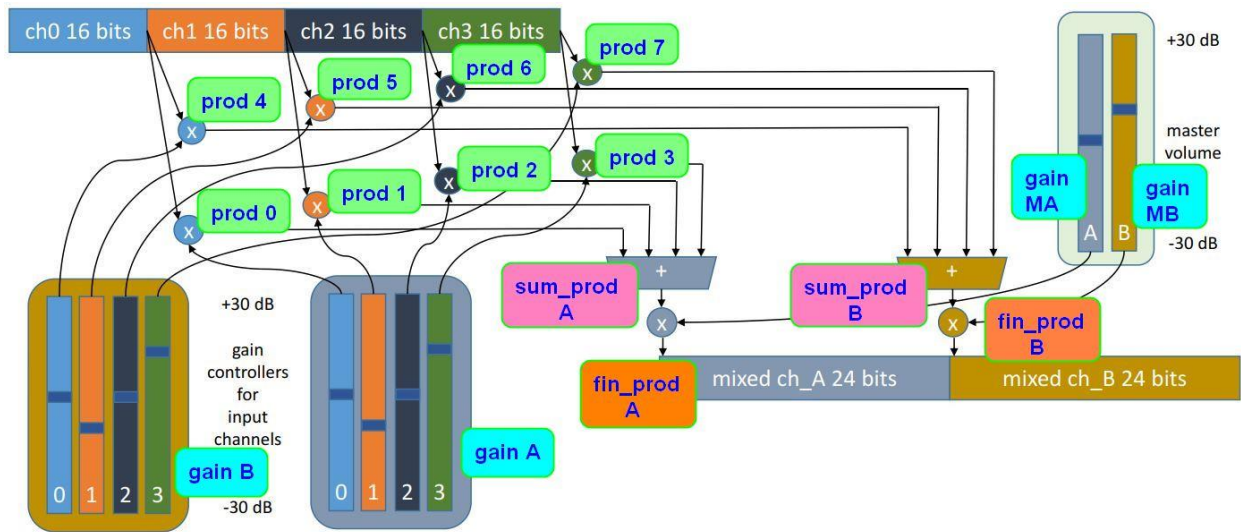
Figure 3: Audio test and result

2.2.1 System behavior

Implemented in *mixer.vhd*.

In order to have two mixed output channels, the calculation on input signals is performed with the following steps in order. The results for every step are shown in Picture 1.

1. Multiply input signal with gain A and then gain B. The results for this step are $prod_0 = ch0 * gain_A0$, $product_4 = ch0 * gain_B0$, $prod_1 = ch1 * gain_A1$, $prod_5 = ch1 * gain_B1$ and so on.
2. Calculate sum of products. The results for this step are $sum_prod_A = prod_0 + prod_1 + prod_2 + prod_3$ and $sum_prod_B = prod_4 + prod_5 + prod_6 + prod_7$.
3. Multiply the result at step 2 with gain MA and MB. The final results are $fin_prod_A = sum_prod_A * gain_MA$ and $fin_prod_B = sum_prod_B * gain_MB$. These are also the output.



Picture 1: Results after every calculation

After doing the analysis on the calculation to have the results, the next step is the system behavior implementation. To achieve this, a state machine is defined which helps the writing VHDL code. The behavior of MIXER will follow the state machine. Every time the input signal is available, the system will perform the calculation on that signal together with previous results and output the result. The whole behavior is display in Figure 3.

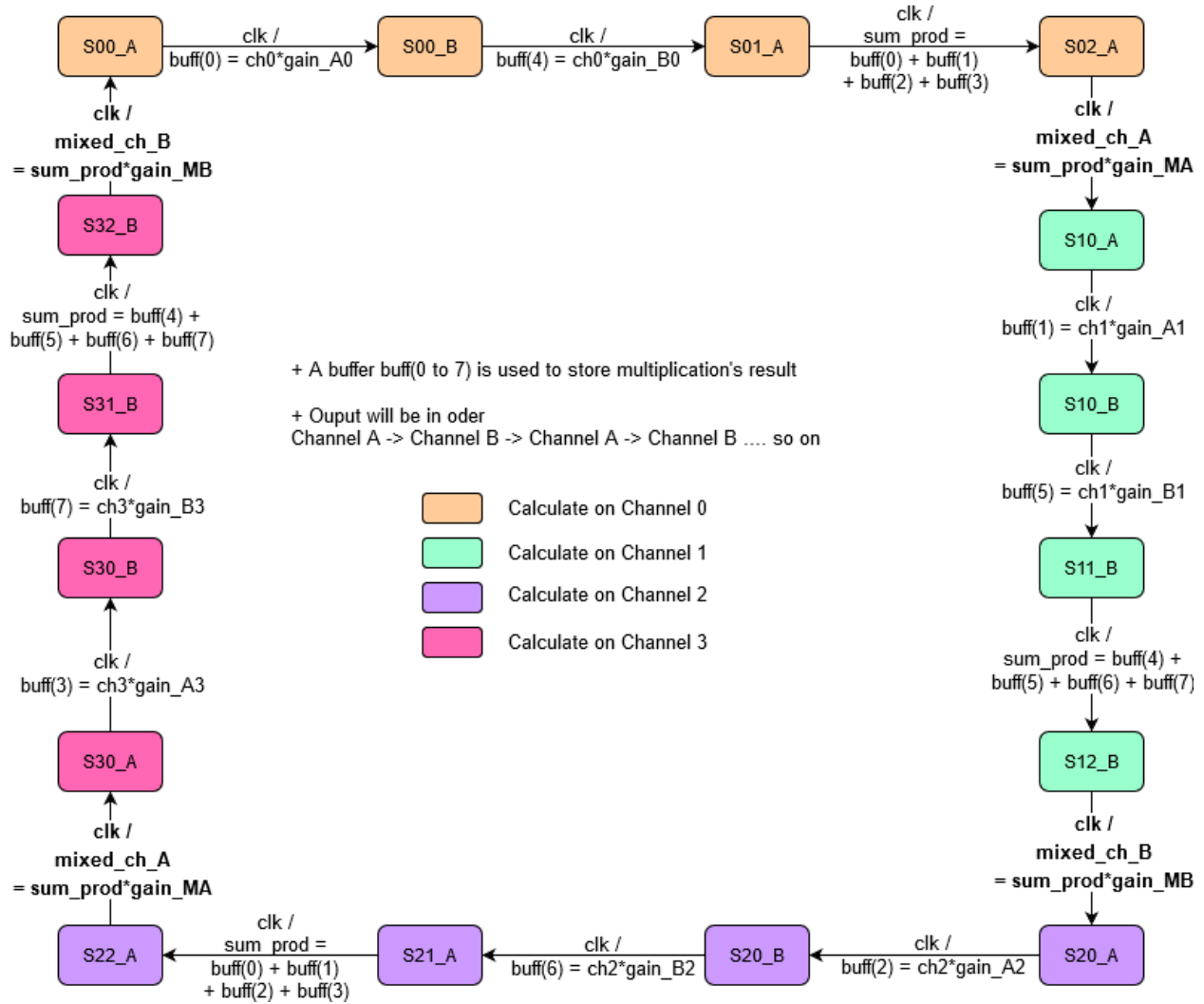
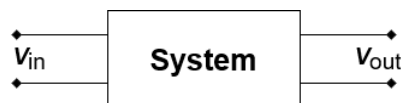


Figure 3: MIXER state machine

2.2.2 Gain - loss calculation

Implemented in *gain_calculate_pkg.vhd* with function *gainCal*.

The calculation for Amplification (Gain) and Damping (Loss)^[1]



$V_{out} > V_{in}$ means **gain**. The dB value is positive (+).

$V_{out} < V_{in}$ means **loss**. The dB value is negative (-).

$$L = 20 \cdot \log(V_2 / V_1) \text{ in dB.}$$

When

$$L = 30\text{dB} \Leftrightarrow 20 \cdot \log(V_2 / V_1) = 30 \Leftrightarrow V_2 \approx 31.62 V_1$$

$$L = -30\text{dB} \Leftrightarrow 20 \cdot \log(V_2 / V_1) = -30 \Leftrightarrow V_2 \approx (1/31.62) V_1$$

Base on above calculation the conversion for gain and loss will be calculated as in Table 1.

Gain: 5.5 fixed point, unsigned (-30 dB ... +30 dB)

VHDL: 10 bits, value (1/31 ... 31)

	Bits	Level	Calculation	Example				
Gain	9	+16	Data_in * Level	Input:	b'10110 00000			
	8	+8		Level:	1*16 + 0*8 + 1*4 + 1*2 + 0*1			= 22
	7	+4		Data_in:				= 2
	6	+2		Result:	2* 22			= 44
	5	+1						
Loss	4	+1	Data_in / Level	Input:	b'00000 10011			
	3	+2		Level:	1*1 + 0*2 + 0*4 + 1*8 + 1*16			= 25
	2	+4		Data_in:				= 2405
	1	+8		Result:	2405 / 25			= 96
	0	+16						
Maximum Gain: 16 + 8 + 4 + 2 + 1 = 31 (≈ 31.62 ⇔ 30 dB)								
Maximum Loss: 1 / (16 + 8 + 4 + 2 + 1) = 1/31 (≈ 1/31.62 ⇔ -30 dB)								

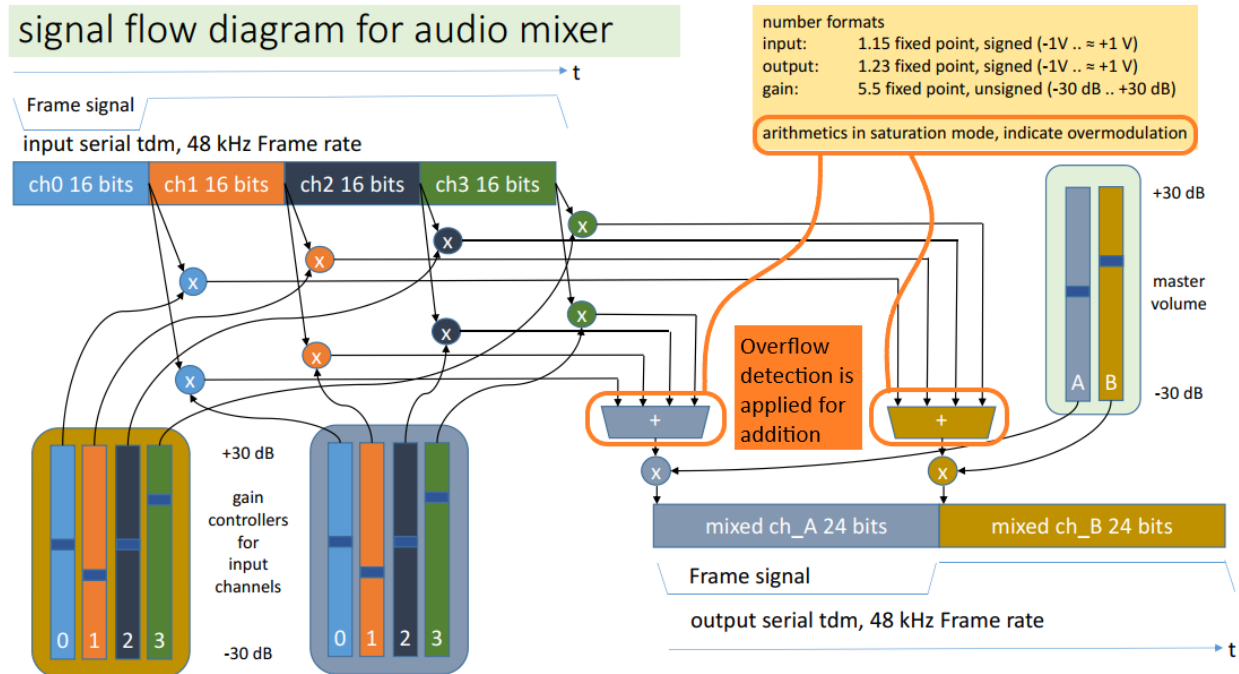
Table 1: Gain and Loss conversion

There are different ways to convert the gain from input signal. Above is only one way to calculation gain from raw data. This can be changed in the function *gainCal* without effect to other functions.

2.2.3 Saturation algorithm

Implemented in *gain_calculate_pkg.vhd* with function *overflowCal*.

The saturation algorithm is developed to detect overmodulation when the gain is adjusted at high level and the value of modulated signals are beyond the limit value (-1V ... 1V). This is one of the most complicated part in VHDL code for this assignment. The algorithm is applied for addition four signals after the multiplications for both output channels A and B as in the picture below.



Picture 2: Overflow detection

Before developing an algorithm to detect overflow for adding four signed number, the overflow rule for adding two N^1 -bit signed numbers is consider with Table 2.

Num 1 sign	Num 2 sign	Result sign	Judgment	Result
+	+	-	Positive overflow	'01' & "N-1 bit 1"
-	-	+	Negative overflow	'10' & "N-1 bit 0"
Other cases			No overflow	'00' & Num 1 + Num 2

Table 2: Overflow detection for two signed N-bit numbers

It is notable that two numbers with different signs will never create an overflow. Base on this rule, when adding four numbers, each two numbers with different signs will be added first, after that the sign of result will be checked and perform adding with next number. If there is no number with different signs, the Table 2 will be used for judgment.

The possibility for the sign of each number is positive (+) and negative (-). Therefore, the total cases need to be considered is $2^4 = 16$ for four numbers.

The pseudo code for detecting overflow is described as below.

¹ N = 16 is the current setting

Algorithm overflowDetection (num1 <N bit>, num2 <N bit>,
num3 <N bit>, num4 <N bit>)

Description: Detect overflow in result when adding 4 N-bit signed numbers

Pre 4 signed N-bit Numbers

Post add 4 signed N-bit Numbers with overflow detection

Return (N+2)-bit Number with information as below

bit[N+1 and N]: Overflow indication
00: No overflow
01: Positive overflow
10: Negative overflow

bit[N-1 downto 0]: Adding result

//Adding 2 number the result is extended 1 bit. Therefore, adding 4 number the result will be extended 2 bits

```

1 temp = ("00" & num1) + ("00" & num2) + ("00" & num3) + ("00" & num4);
2 if (num1 >= 0 and num2 >= 0 and num3 >= 0 and num4 >= 0) then          //+ + + +
    1 if temp(N+1 downto N-1) not equal "000" then result = "010" & "N-1 bit 1"
    2 else result = "00" & temp(N-1 downto 0)
3 else if (num1 < 0 and num2 < 0 and num3 < 0 and num4 < 0) then        //- - - -
    1 if temp(N+1 downto N-1) not equal "111" then result = "101" & "N-1 bit 0"
    2 else result = "10" & temp(N-1 downto 0)
4 else
    //store the sign of numbers into variable numPos with following rule
    //      num4  num3  num2  num1  num4  num3  num2  num1
    //bit    7      6      5      4      3      2      1      0
    //sign   -      -      -      -      +      +      +      +
    1 If (num1 >= 0) then numPos(0) = '1' else numPos(4) = '1' ... If (num4 >= 0) then numPos(3) = '1' else
    numPos(7) = '1'
    2 switch numPos
        1 case numPos = "1000 0111"                                     //+ + + -
            1 temp1 = num1+num4                                          //+ -
                1 if temp1 >= 0 then temp2 = temp1 + num2                //+ +
                    1 if temp2 < 0 then result = "010" & "N-1 bit 1"
                    2 else temp3 = temp2 + num3                          //+ +
                        1 if temp3 < 0 then result = "010" & "N-1 bit 1"
                        2 else result = "00" & temp(N-1 downto 0)
                2 else temp2 = temp1 + num2                              // - +
                    1 if temp2 >= 0 then temp3 = temp2 + num3            //+ +
                        1 if temp3 < 0 then result = "010" & temp(N-1 downto 0)
                        2 else result = "00" & temp(N-1 downto 0)
                    2 else result = "00" & temp(N-1 downto 0)
        2 other cases : Apply the same method
5 return result

```

Figure 4: Pseudo code for overflow detection algorithm

2.2.4 Clock calculation and auxiliary functions

Implemented in *mixer.vhd* with process named *clk_divider* and *gain_calculate_pkg.vhd* under the functions *limitResult* and *limitFinalResult*.

In order to process the 48KHz serial input signals properly, the clock calculation is necessary. Suppose the main clock on the FPGA is 96MHz, this is much faster than the speed of input signal. If the calculation is performed on every main clock rising edge is unnecessary and impossible for validation. Hence, a clock divider is a solution to overcome this challenge. It will create a new clock with speed much slower than main clock.

Base on the figure 3, the result is output for every four clocks and considering the speed of the input signal is 48KHz the clock calculation will be $4 \times 48\text{KHz} = 192\text{KHz}$ to make sure all the operations on the input have to finish before the next input comes. It is also notice that the output has the same speed as input. With the flow in the state machine the output is produced after every four calculation clocks and just right before the next input come, it means that the **output speed is also 48KHz**.

To generate the 192KHz clock speed from 96MHz, a counter is used. The value of this counter is calculated based on the formula.

$$\text{Counter_value} = 96\text{MHz}/192\text{KHz} = 500$$

It means that on every rising edge of 96MHz clock the counter value will be increased by '1' and the period for new clock is the time that counter increase from 0 to 499.

Along with clock calculation, the limit functions are implemented to limit the result of multiplications which are shown in Picture 1 in case the result are greater than value of N-bits signed number (16 bits and 24 bits as default value).

3 VERIFICATION

In HDL, writing testbench and running simulation are important as same as design and implementation. The test phase consume time even much higher than implementation phase. It is not like creating software where the bug in the code can be fixed and then software will be installed or flashed again on the same hardware. While HDL is used to create hardware such as FPGA and ASICs, if any bug in or after the production (hardware is in production) the lost will be huge, chips cannot be used and the whole process is restarted in most of cases even for a small mistake.

Design needs creativity - verification needs patience.

3.1 Unit test - Test for Gain - loss calculation and limit functions

This test is to validate the functions written in *gain_calculate_pkg.vhd* including *gainCal*, *limitResult* and *limitFinalResult*. This test can be considered as unit test. The testbench can be found in *calculate_tb.vhd* and the test cases is presented in *mixer_test.xlsm* under the sheet named *TestCase_Manual* with IDs: *GAI_01*, *LOS_01*, *MUL_01*, *MUL_02*, *LIM_01*, *LIM_02*. The result including simulation waves are also available for each test case in that sheet.

Note: for convenience while analyzing wave forms, some test cases are commented, just uncomment and run simulation again to see the waves.

3.2 Unit test - Test for overflow detection

Testbench is also written in *calculate_tb.vhd* which validate the function *overFlowCal*. Accordingly, the test cases are mentioned with IDs *OVF_01*, *OVF_02*, *OVF_03*, *OVF_04* and *OVF_05*. The result can also be found in *mixer_test.xlsm* under the sheet named *TestCase_Manual*.

3.3 System test – Test circuit behavior

This test is focus on the behavior of the circuit which is implemented in file *mixer.vhd*. To ensure the behavior of system is proper as well as the test is reliable, the large amount of data is generated by MS Excel randomly. After that, the test data is converted into 16-bit signed number with HEX format. A macro is developed in Visual Basic to perform the calculation on this data automatically.

gain_ctrA0	gain_ctrA1	gain_ctrA2	gain_ctrA3	gain_ctrB0	gain_ctrB1	gain_ctrB2	gain_ctrB3	gain_ctrMA	gain_ctrMB	data_in
0	0	0	0	0	0	0	0	0	0	25930
32	0	32	0	32	32	32	32	32	0	16524
20	29	23	30	31	21	24	23	30	20	-6554
864	800	352	704	640	256	256	384	192	352	9251

Table 3: Generated data sample

Process for system test is illustrated in the following diagram

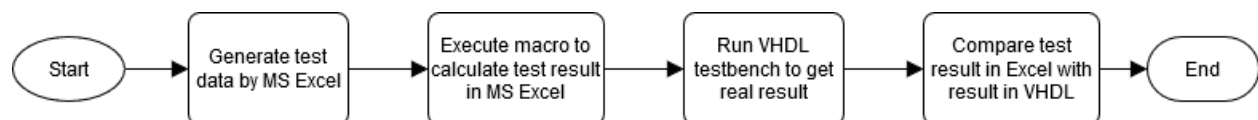


Figure 5: System test process

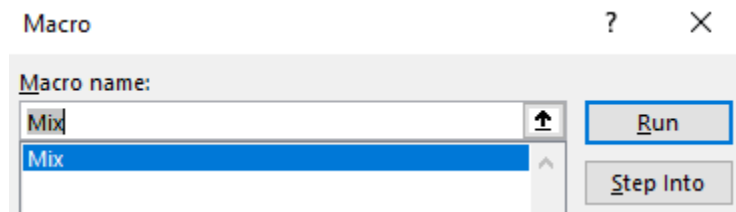
The tested is executed with the following steps:

1. Open *mixer_test.xlsm* → *Generation* tab → copy any row with existing data to a new row, the new data will be generated.
2. Copy the generated data with only the area with column name in Table 3 → override the content in *Calculation* sheet with the same column (Paste value only). The purpose of this step is to avoid the data keep changing automatically on the *Generation* sheet whenever any change on the whole workbook and to generate HEX data in the *TestDataInHex* sheet.
3. After having the new data in *TestDataInHex* sheet, save this sheet in txt format with name *mixer_data_in.txt* and eliminate unnecessary “F” characters. The final content for the file should look like below.

000	000	000	000	000	000	000	000	000	000	3E1A
000	020	020	020	000	020	020	020	020	000	A1DD
015	01F	01C	017	015	01A	01A	015	01D	01A	A669
0E0	1C0	360	3A0	220	100	380	1C0	1A0	2C0	EAAF
012	004	006	015	002	017	01F	018	004	01E	9D4A

Picture 3: Content of test data

4. Once everything is ready, running the macro Mix in *mixer_test.xlsm* on sheet to get the result with Excel. After that, run the simulation for *mixer_tb.vhd*. Compare the result in *mixer_data_out.txt* with column *final_out* in *mixer_test.xlsm* and the content will be the same.



Picture 4: Run macro Mix to generate result automatically on MS Excel

3.4 Acceptance test - Test with audio data

To be able to test with audio data, a wave file^[2] is created with Octave. This wave file has data rate 48KHz and the duration is 1 second with four channels^[3]. The command to create audio file is below:

```
>> freq = 1000;
>> fs = 48000;
>> tone=[sin(2*pi*freq*(1:fs)/fs);sin(pi*freq*(1:fs)/fs);sin(0.3*pi*freq*(1:fs)/fs);sin(1.3*pi*freq*(1:fs)/fs)];
>> tone = tone';
>> audiowrite('test.wav',tone,fs);
```

Figure 6: Generate audio file by Octave command

The data for file test.wav is shown in Picture 3 as below. The duration is 1 second with 4 channels and data rate is 48KHz. Hence, total data sample in this file is $48000 \times 4 = 192000$ samples and stored in array 48000×4 .

Name	Class	Dimension	Value
freq	double	1x1	1000
fs	double	1x1	48000
tone	double	48000x4	[0.13053, 0.065403, 0.019634, 0.084982; 0.25882, 0.13053, 0.0...

Figure 7: Audio data structur in Octave

Once wave file is created, a C++ program^[4] read this file, extract the data of samples in double numbers, convert to 16-bit signed hex number and write into a text file. After that, a VHDL testbench will read this text file line by line to get the audio data, running the simulation for this testbench will create data for new wave file with duration also the same 1 second, 2 channels 24bits with sample rate is double compare to original file (the channel is reduced the sample rate is increased). C++ program read this new data and create a new audio file with new format

The whole process for testing is followed the below chart. The testbench can be found in *test_wav_tb.vhd*.

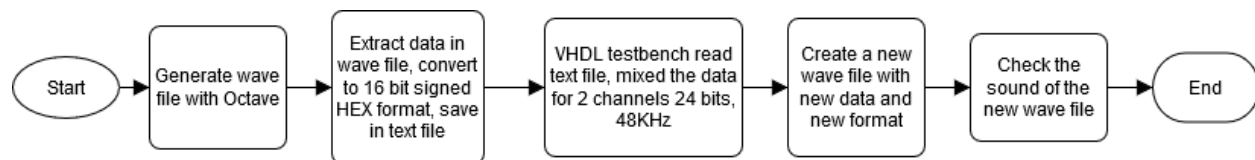


Figure 8: Audio test process

To perform the test, follow the steps:

1. Open Octave and execute commands as mentioned above to create *test.wav* file
2. Copy *test.wav* into *AudioMixer\bin\Release*. Double click on *AudioMixer.exe*. It will generate a new file name *wave_in.txt*.
3. Copy *wave_in.txt* into VHDL source code folder, then run simulation for *mixer_tb.vhd*. It will generate a new file name *wave_out.txt*.
4. Copy *wave_out.txt* into *AudioMixer\bin\Release*. Double click on *AudioMixer.exe* again. A new file is generated under the name *test_mixed.wav*. This is the audio file after mixing.

Compare the sound from *test_mixed.wav* and *test.wav*.

4 SYNTHESIS

Synthesis is the process of converting the code in Hardware Description Language (HDL) to gate level netlist considering the constraints and optimizations. It also involves translating the HDL code to technology specific gates. Synthesis has 3 steps as listed below:

1. Translation: RTL code is translated to technology independent representation. The logic obtained would be in Boolean form.
2. Optimization: it includes optimization using SoP (Sum of Products) and PoS (Product of Sums) optimization methods.
3. Technology mapping: The technology independent code is converted to technology specific logic gates and thereby supports in optimization

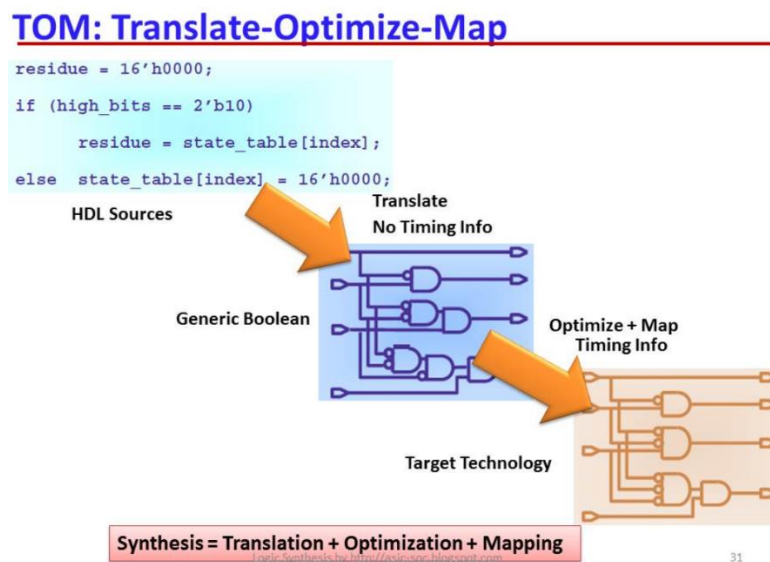


Figure 9: Synthesis steps ^[5]

At first, setting project with proper FPGA device in Lattice, the result is shown as below:

mixer project summary			
Module Name:	mixer	Synthesis:	SynplifyPro
Implementation Name:	mixer	Strategy Name:	Strategy1
Last Process:		State:	
Target Device:	LAXP2-17E-5QN208E	Device Family:	LatticeXP2
Device Type:	LAXP2-17E	Package Type:	PQFP208
Performance grade:	5	Operating conditions:	AUTO
Logic preference file:	mixer.lpf		
Physical Preference file:	mixer/mixer_mixer.prf		
Product Version:	3.11.3.469	Patch Version:	
Updated:	2020/09/30 13:41:17		
Implementation Location:	E:/ESM_Master/Semester_2/HW_SW_Co_Dsgn/Lattice/mixer		
Project File:	E:/ESM_Master/Semester_2/HW_SW_Co_Dsgn/Lattice/mixer.ldf		

Figure 10: Project setting in Lattice

In this section, the main synthesis steps will be mentioned for this assignment. These steps have modifications to have a better result, they are still belonging to standard synthesis process in Lattice Software. The successful steps for synthesis VHDL code are shown below.

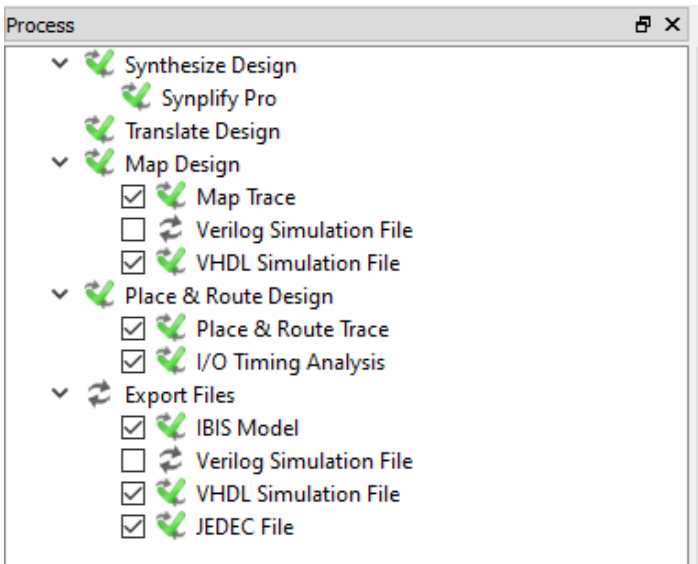


Figure 11: Synthesis process in Lattice

4.1 Verify Functionality with Simulation

Before running the synthesis, a final verification step to ensure the circuit has the expected behavior. It is done via Menu Tools → Simulation Wizard. This will trigger Active-HDL software to simulate the behavior of the circuit. The result looks like below picture. In the wave form, the timing will be measured to ensure that the speed of *data_in* is 48KHz 16 bits, the speed of *data_out* is 48KHz 24bits, the main *clk* is 96MHz, the *clkCal* is 192KHz and so on.

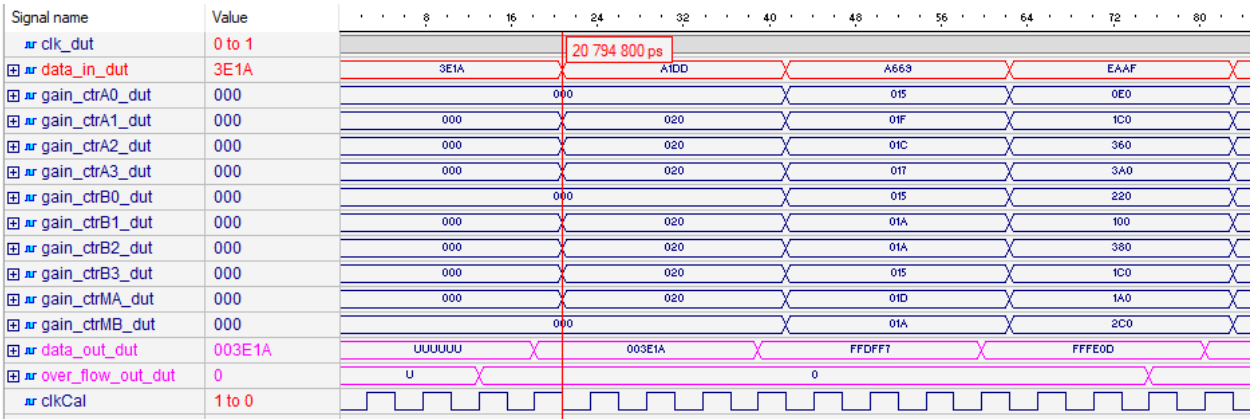


Figure 12: Simulation result

4.2 Location pin setting

The pin location for input and output signals can be configured in *mixer.lpf*. In this assignment, only Pin location for data input, clock and data output are set. The other input such as gain levels are also configured in the same way. The basic configuration for some pins as below:

```
LOCATE COMP "clk" SITE "30";  
LOCATE COMP "data_out[0]" SITE "64";  
LOCATE COMP "data_in[0]" SITE "197";  
INPUT_SETUP ALLPORTS 10.417000 ns CLKPORT "clk";
```

Figure 13: Location pin setting

4.3 Examine result and enhancement

Once the setting is completed, the synthesis can be processed. For each step in Figure 10, a log file is generated. Some warning and errors might appear. Errors must have fixed them and start process from the beginning.

For example, after finishing the Map Design phase, a list of instances is generated. Then check whether the number of instances is meet with requirement. In this case, the multiplier is limited to two. If requirement is not meet, VHDL code must be optimized and do steps again.


Hierarchy---Post Map Resources						
Unit	File	LUT4	PFU Registers	MULT	Carry Cells	SLICE
 mixer_datapath	...mixer/source/mixer.vhd	966(966)	15(15)	2(2)	334(334)	954(954)

Figure 14: Generated instances in Lattice

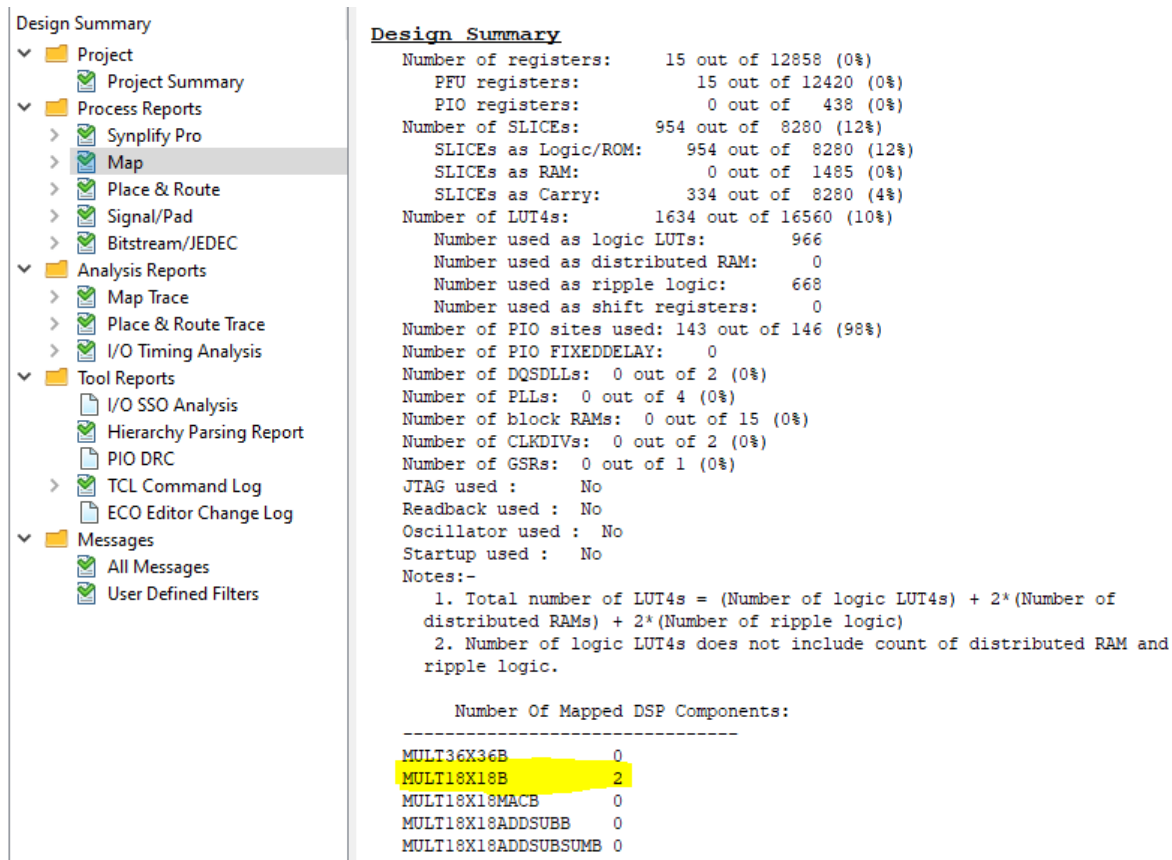


Figure 15: Design summary for MIXER

4.4 Floor plan and physical view

An overview, for the instances and ports on FPGA is shown in floor plan view as below. The physical view is on the right with connection between instances. The result can be different on different machine.

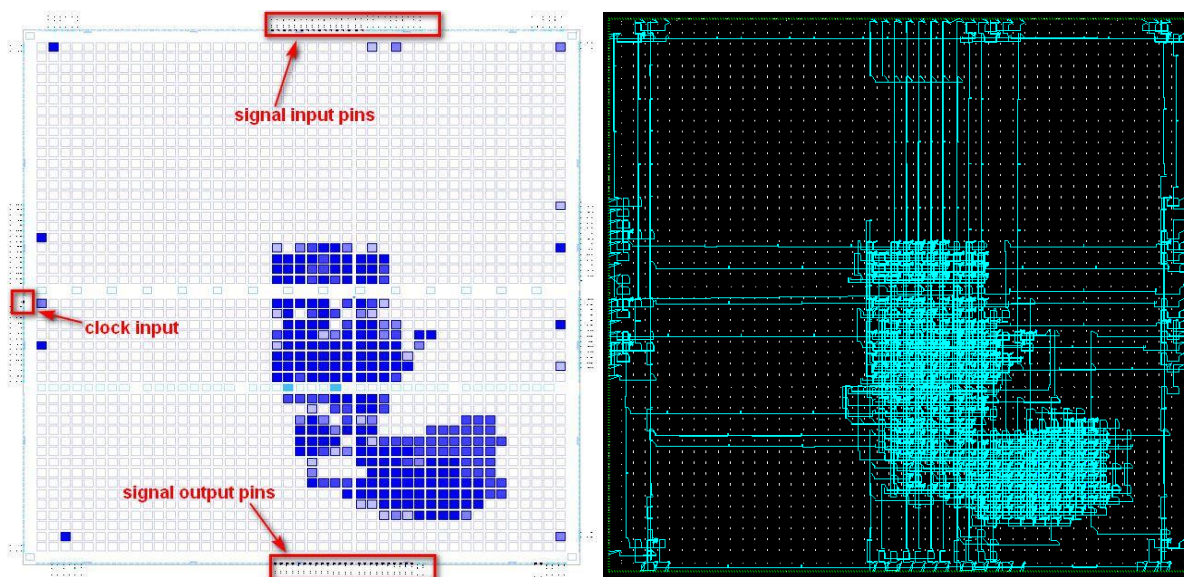


Figure 16: Floor plan and physical View after synthesis

5 CONCLUSION

5.1 Gained knowledge

By doing this assignment, we had the chance to practice design a simple FPGA chip, code, simulation and synthesis with VHDL. We learned how to use ModelSim for coding, simulating VHDL. We got to know the basic steps for a synthesis an FPGA chip as well as had a view of circuit after synthesis with Lattice.

There were many issues during the implementation, they were really challenging for us. However, we gained many coding and simulating VHDL techniques via these issues. We had a chance to learn how to develop basic process for development to ensure the quality of the outcome and meet the deadline.

This assignment trained us about the carefulness and patience not only through every line of code but also research for new topics.

5.2 Open points and improvement

As beginners, we still lack domain knowledge and document. Therefore, the result certainly needs to be improved. Especially, the VHDL code optimization was considered but some warning during the synthesis are still existing.

Due to the non-availability of FPGA board, the outcome is available at computer simulation only. Setting clock with real hardware, checking the output and many more other settings which can be done with only real hardware are pending.

The reset feature should be added.

5.3 Extension

Providing a preset base on the type of sound is also a feature can be added in the future.

6 REFERENCES

[¹] Calculation: Amplification (gain) and damping (loss) as factor (ratio) to the level in decibels (dB) [Online]. Available at: <http://www.sengpielaudio.com/calculator-amplification.htm>

[²] Microsoft WAV sound file format [Online]. Available at: <http://soundfile.sapp.org/doc/WaveFormat/>

[³] Getting started with Octave and making a sine oscillator [Online]. Available at: https://www.youtube.com/watch?v=tx_cjBjZ2zM

[⁴] How to read WAV format file in C++ [Online]. Available at: <https://rogerchansdigitalworld.blogspot.com/2010/05/how-to-read-wav-format-file-in-c.html>

[⁵] ASIC Synthesis: Synthesis definition, goals [Online]. Available at: <https://asic-soc.blogspot.com/2013/07/asic-synthesis-synthesis-definition.html>

Tutorials for beginners and advanced in VHDL [Online]. Available at: <https://www.nandland.com/vhdl/tutorials/index.html>