PROJECT4 REPORT

1. **TITLE**

Digital Modulation for Sound Signals

GAROUFALI MARIA NEFELI, AEM: 3129, 20/1/23

1. **SUMMARY**

The 4th lab project of the course is presented in this report. This project concerns the construction of a communication system recording guide and sound playback using Digital Modulation. Its implementation took place in stages, which will be detailed. Each stage is described by Implementation, Verification, and Experiment.

1. **INTRODUCTION**

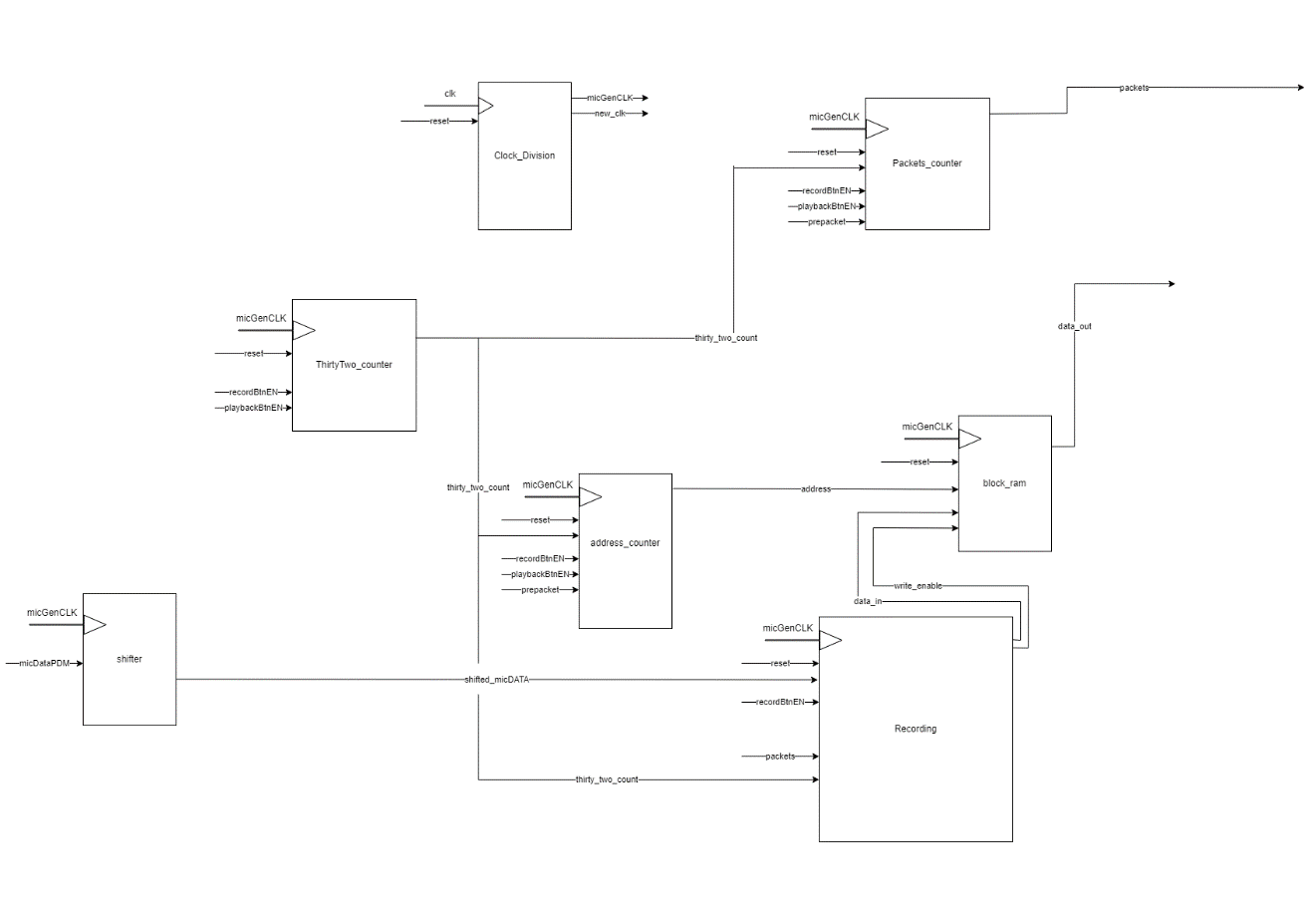
The objective of the 4th lab work is to implement a recording guide and sound playback using Digital Modulation. Practically the guide must convert an analog signal from the continuous time domain into a digital signal of discrete values for the audio recording process. For the reproduction (playback) sound must be followed the reverse procedure (from discrete field to continuous).

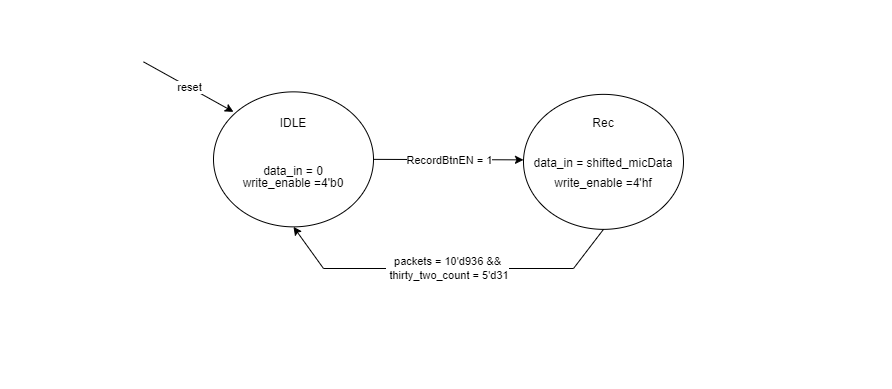
The conversion from analog to digital signal and from digital to analog is performed internally by the peripherals to be used, that is, from the ADMP421 microphone and the Sallen-Key Butterworth low-pass filter which drives the jack audio output port. So, we can deal with its processes’ conversion to and from the continuous field of time as a "black box". Thus, in the framework of the work we manage only digital signals. More specifically with the pressing of a button, the recording of the message should start. After the recording is done, another button is used to indicate the playback functionality. When this button is pressed the message should be played in the Mono Audio output, in other words, the speakers.

1. **PART A – Microphone Driver Implementation and Data Deserialization**

* **IMPLEMENTATION**

In this part, we should implement the Microphone Driver. More specifically we should take the data from the microphone deserialize them and then pass them to the block ram. To do that we should, first of all, make an MMCM to divide the clock of the board and make a clock of 500ns or 2MHz. This is necessary as the microphone produces data at this frequency. Reading the manual for the MMCM we conclude that such a big frequency should be done by cascading the MMCM. The project also requires a clock of 200MHz. For this reason, we use the MMCM again to produce this clock too. Having produced all the necessary clocks, we should read again the manual to initialize the block ram. We read in the manual that the block ram we want to use has 2 ports for input and 2 ports for output, and, that it can be used in cascade for more data. We don’t want the cascade mode and neither the 2 ports. So, we maintain port A for all our purposes. I selected to pass the data to the memory in “packets” 32 bits. For this width, we need to use the [14:5] bits of the address and increase them by 1 every time we want to change the address in the memory, while the 15th bit is always set to 1. This happens because when we use the memory in blocks, we need fewer bits to refer to them. However, instead of using the [14:5] bits and increasing them by 1, I used the [14:0] bits and increase them by 32 each time I wanted to change the address. This leads to exactly the same result as if we look closer, by adding 32, we use in fact the bits [14:5] the addition happens to the 5th bit. Now, we needed to make a deserializer to take the data from the microphone bit by bit and convert them to data with a 32-bit width for the memory. This module is nothing but a shifter working by the clock. In every posedge clock, it shifts the data coming from the microphone to a 32-bit variable. As a sequential system, it is constructed with non-blocking assignments. The output variable of this module is going as input to the FSM of this Part. Before analyzing the memory let’s talk about 2 other modules necessary for the project. These are a 32-bit counter and a counter of packets. As we previously mentioned we send the data in packets of 32 bits. For this reason, we need the 32-bit counter to count each bit of the packet. The second counter is obviously to count the number of packets we send to the memory. Both these modules are constructed with non-blocking assignments as they are sequential. The memory we use is 36Kbits. Rounding this we could say that we want to use 30Kbits. 30,000 bits divided by 32 bits per packet results in 937.5 packets. However, to make our life easier I selected to send 937 packets instead of 937.5. That is why we need this counter. Another input to the FSM is the variable for the recording button. So, let’s analyze the FSM. The FSM to summarize takes as inputs the clock, the reset, the rebutton, the shifted data from the deserializer, and the output of the counters, the thirty\_two\_count, and the packets. As the data coming from the microphone and the deserializer is coming by the clock of 2MHz I used the same clock for the FSM and every other module (counters). I could use the smaller clock of 5ns (200MHz) but except that it is impractical it has a phase difference with the clock of 500ns (2MHz). This happens perhaps due to the cascading needed to produce the 500ns clock. So, I preferred the accuracy of the 500ns clock. The FSM is MOORE and has 2 states the IDLE and the Rec. The outputs of the FSM are the write enable and the data\_in. Both the outputs will go to the Bram as inputs. In the IDLE state both the outputs are set to 0. As the rec\_button is pressed we move to the next state (Rec). There the write enable is set to 4’b1111 and the data\_in takes the value of the deserializer in other words, the shifted data. In this way when we are in the Rec state by pressing the button the ram will take the data from the microphone and as the write enable is set to 1 they will be written to the memory. The FSM returns to the IDLE state as we have written all the packets in the memory (or with reset). In other words, if packets are 936 and we are in the final bit of that packet i.e. thirty\_two\_count is 31 we return to the IDLE state as the recording has come to an end. Finally, let’s analyze another very important unit for the project and the Part. The address\_counter. As we can assume from the title this module is a counter and its purpose is to count and change the address each time we want to move to the memory. As a counter, it is constructed with non-blocking assignments. The counter is set to 0 once the reset button is pressed. It is also set to 0 once the rec and play buttons are pressed. This is done to count properly the addresses from the start of the recording/playback and doesn’t lose any of them. The counter is also set to 0 once it has reached the last address written/used from the memory and simultaneously we are at the last bit of that address. The counter changes its value increasing by 32 each time we are at the last bit of an address. This is quite logical as at the time one block/packet of 32 bits of the memory ends we need to move to the next one. In any other case, the counter holds its previous value. Not to forget that in all this process the 15th bit is always set to 1 as the manual says and the implementation requires. Let’s also notice again that in violation of the manual, I use all the [14:0] remaining bits of the address variable instead of the [14:5]. That’s not a problem though since I increase the value by 32 and not by 1. So, the result is the same. Let’s now focus on the implementation of the counters. Starting with the 32 counter is set to 0 whenever the reset button is pressed. It is also set to 0 when the rec or play buttons are pressed. That happens to avoid count losing by the start of the recording or the playback. In other words, we do this to ensure that the counter will start counting from 0, by the time we press the corresponding buttons to record or play our message. In any other case, we increase its value by 1, at every posedge clock. If we had the smaller clock (5ns) we could not do that directly. We should have created an enable pulse at every 100 posedges of the 5ns clock and increased the counter at every posedge clock && enable = 1. Let’s now deepen into the implementation of the second counter, the packets\_counter. This is also a counter and so a sequential system. Thus, it is constructed with non-blocking assignments. Now as the reset or the rec or the play buttons are pressed the counter is set to 0. The reason for that is the same as it was explained above for the other counter. This counter is also set to 0 as it reaches its final value. This is when the packets are 936 && we are at the last bit of the last packet i.e., thirty\_two\_count is equal to 31. The counter increases its value by 1, as we are in the last bit of any packet than the last. This is quite logical, as when a packet ends another must start. In any other case, the counter reserves its previous value. The inputs and outputs of each unit as well as the interconnection between them are shown in more detail in the dataflow below. The FSM is shown also below.





* **SIMULATION**

This Part was tested with the simulation method. For this reason, a testbench unit was made. The goal of this unit was to check at the first point if the data are built properly with the deserializer, if the counters are working as expected and if the data were actually written in the memory. We will see the results of the simulation in the 2nd Part. That is because to see what has been written to the memory we need to implement the reading mode of the memory. So, let’s move to the analysis of Part B.

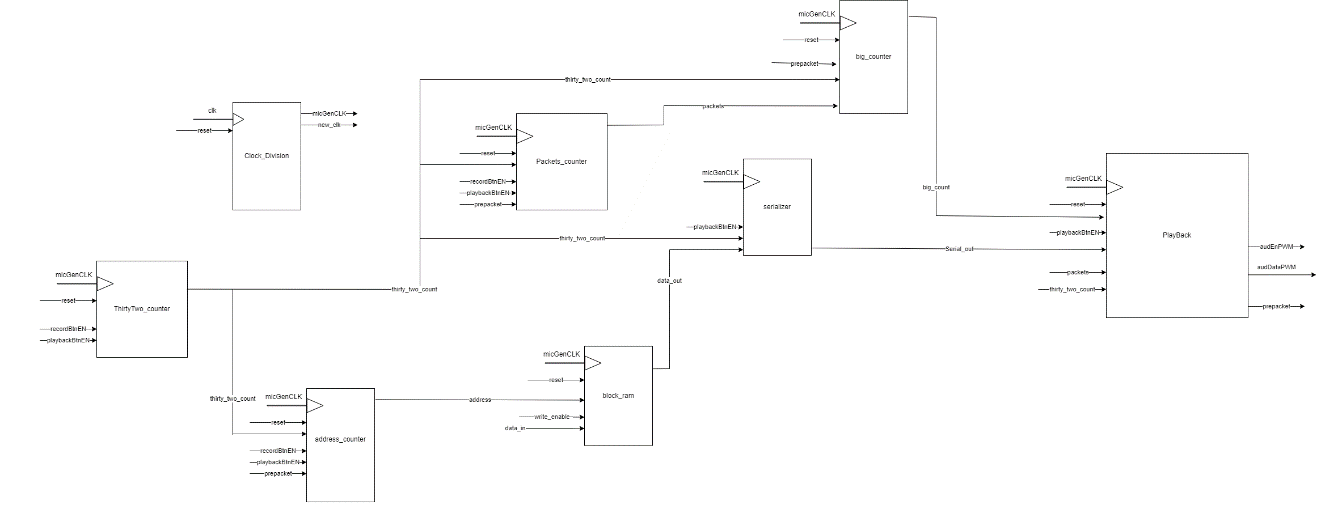
* **EXPERIMENT**

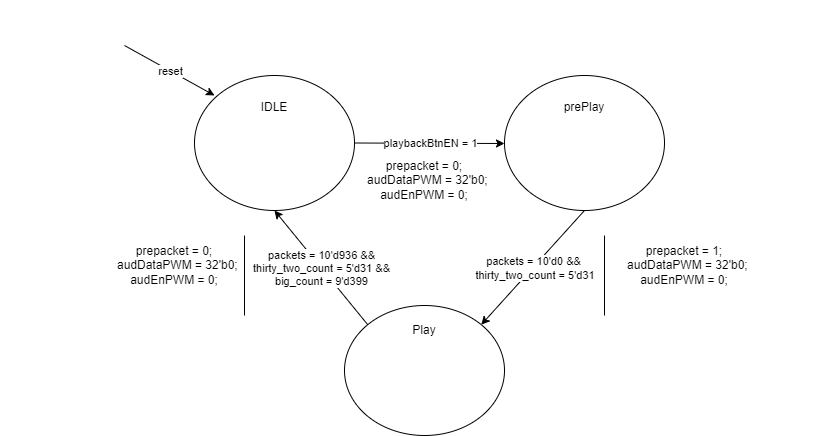
This Part has been checked through the experimental method. However, we can’t check this alone. We will see the experiment complete with the 2nd Part.

**PART B - Audio Player Driver and Data Serialization**

* **IMPLEMENTATION**

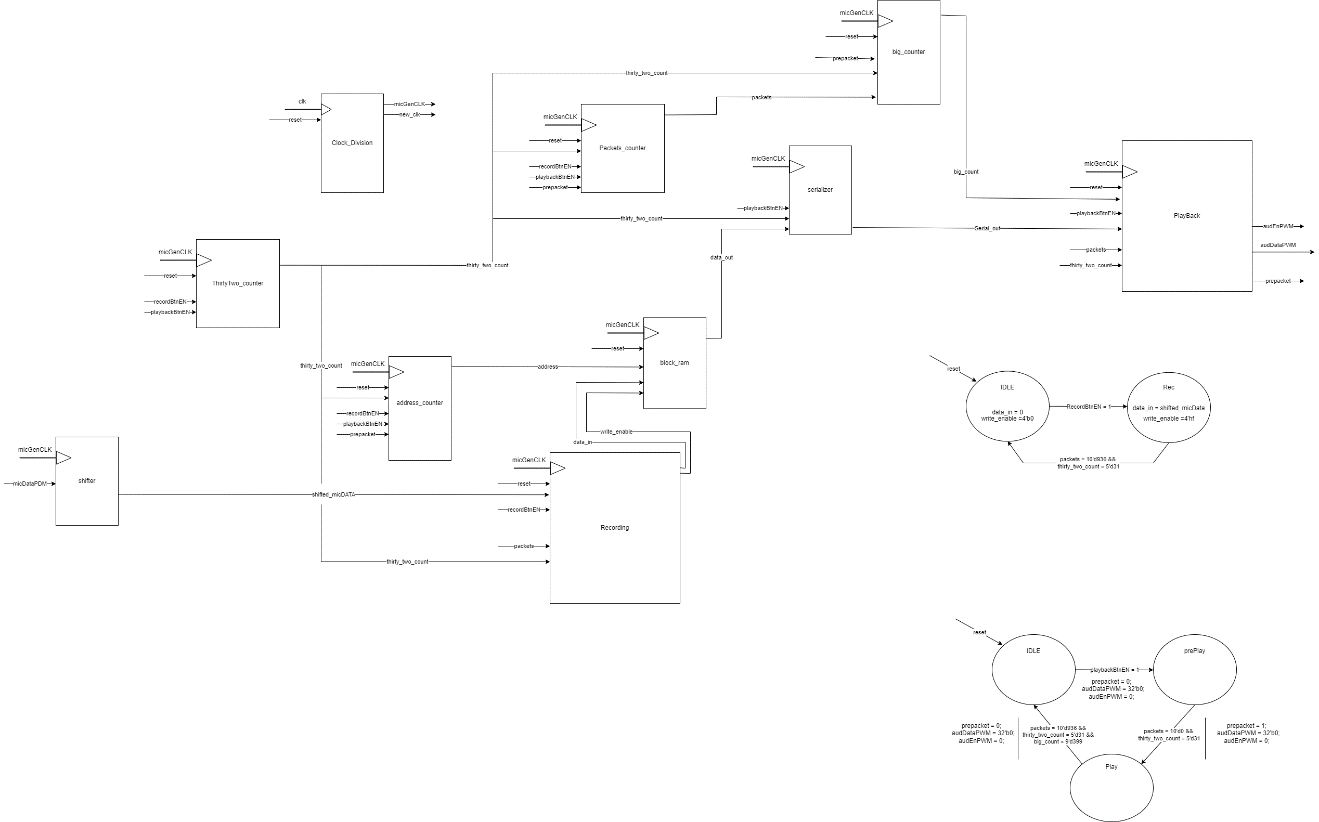
For that part, we wanted to implement the Audio Player Driver where pressing a new button will read the data from the BRAM to send it to the corresponding output driving the Sallen-Key Butterworth Low Pass Filter for a duration of ∼5sec. However, since the saved message is much shorter, it is recommended the saved message be repeated as many times as needed. Retrying the message requires proper record address management. Finally, serializing communication data from memory to filter is necessary. So, starting the analysis by pressing the Play button we must take all the data from the memory. These data will have 32-bit width. To pass them to the speaker/output we must serialize them first. This will be done with the module of the serializer. This is a sequential unit, so it is constructed with non-blocking assignments. This unit takes data from the memory according to the address. This “parallel data” are given to a variable in the module of the serializer, whenever we press the play button or each time we move to the next packet. From this variable, we pass the data to the serial out in every posedge clock. Now let’s examine the address module. The first data from the memory will be given to us after a clock pulse. This would affect the output as the first bit from the first packet will be given to us, not at the time we will press the play button, but after a clock. For this reason, I created a variable to called “prepacket”. So, let’s examine the FSM for Part B where this pulse is created. This FSM is a MEALY FSM, meaning that the output for every state depends on the input. This FSM has 3 states. The first state is of course the IDLE state. The next state is the preplay state. We enter that state as we press the play button. This state lasts for only 1 packet. During this packet, we take the first data from the memory. At the end of the state i.e., when we are in the 0th packet and in the last bit of it we create the prepacket pulse. At this time, we move to the next state too. The next state is the Play state. In this state, we set the audEnPWM variable which is an enabling variable for the audio to 1. We also give the output audDataPWM to the data from the serializer. This variable is an input to other modules. First of all, the packets\_counter. Whenever the prepacket is 1 we set the packets to 0. In this way, having the data from the memory we start from packet 0 to count the packets we will “transmit” to the output/speaker. The address however doesn’t reset its value. From the first address that we are at when we press the play button, we move to the next address at the end of each packet as we did in the recording process. The thirty\_two\_counter is the same as we analyzed it previously. The other thing we wanted was to produce the message in enough time. This is done with an extra counter I created to play the message multiple times. The message lasts currently 0.015s. With the counter, the message will last 400 times more, i.e., 6secs. As a sequential system, this unit is made with non-blocking assignments. This counter counts from 0 to 399 and increases its value by 1, whenever all the packets have been transmitted. This counter resets its value with the reset or with the prepacket. The data flow for both the FSM and that part is shown below.



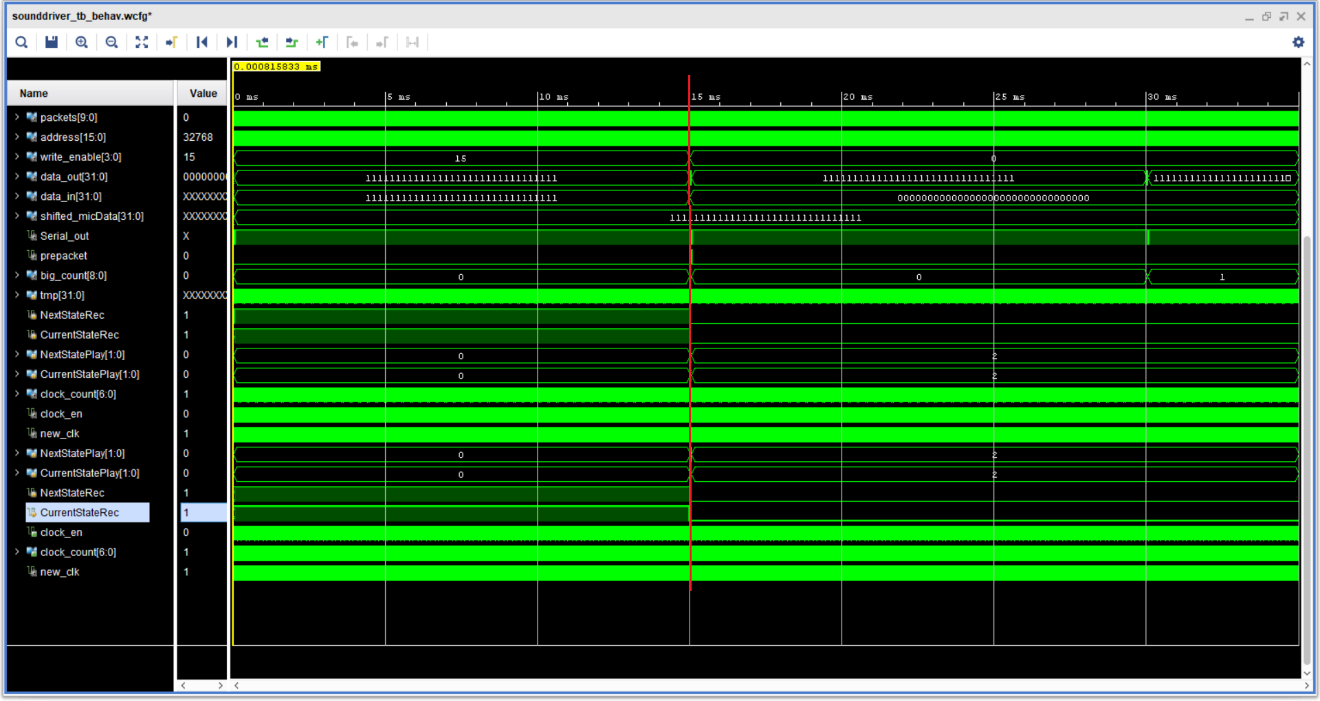


* **SIMULATION**

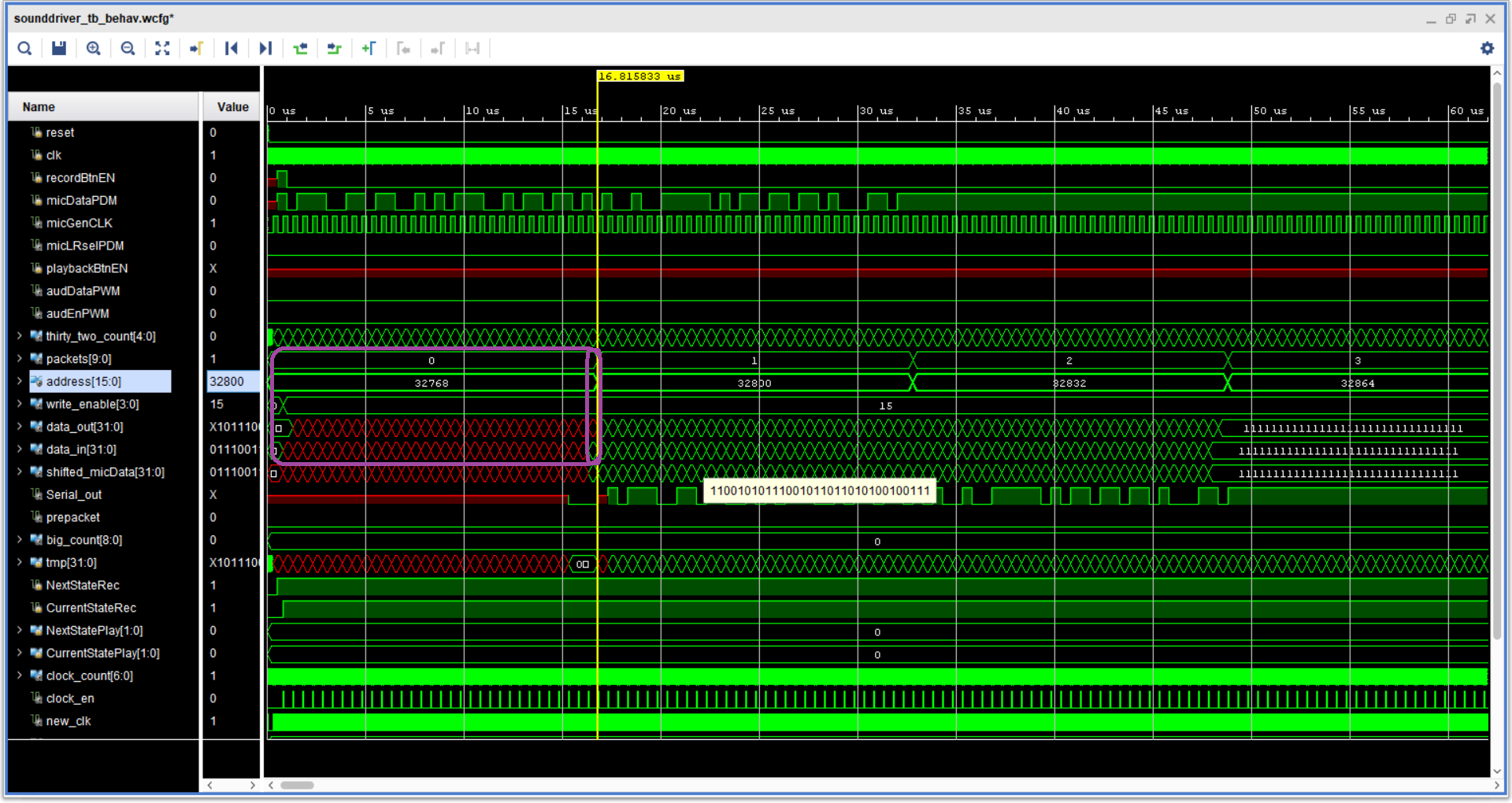
This part was checked with the simulation method. For this reason, we made a testbench. With this, we wanted to see if the counters work right if the FSM moves to the states as expected and if at the end the serial out takes the right data from the memory. We can see the screenshots from the simulation for both parts right below. We also see the data flow of the whole project.



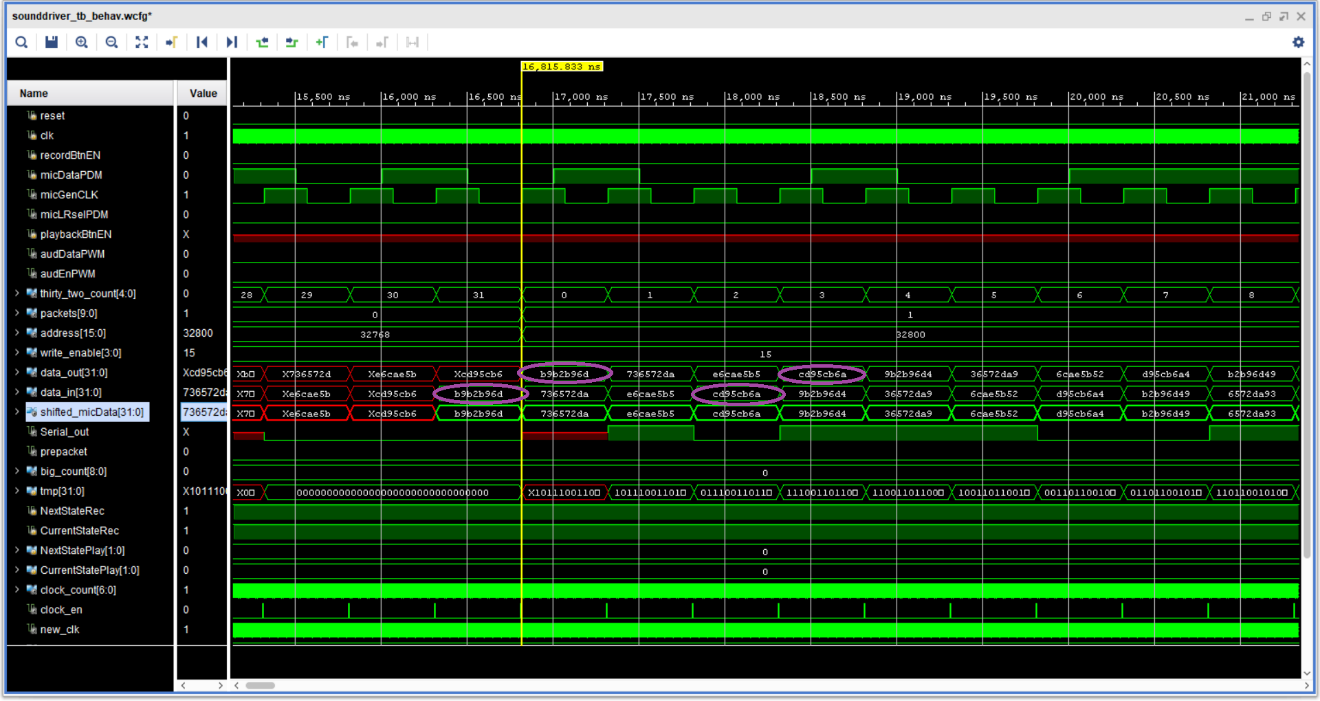
So, let’s examine the results of the simulation in detail. In the first screenshot, we see the results of the whole simulation for both parts. More specifically, before the red line we are in the recording process, and after the red line in the playback process. We also see that the recording lasts 0.015s as expected. We have adjusted the simulation time to assure that the message will continue after the first time it was played. Of course, we couldn’t simulate 6s, because that would take a lot of time. At first glance, we also see the states of the rec FSM. Before the red line, we are at the rec state while after that we are at the IDLE state. The opposite happens with the playback. More specifically we notice that before the red line we are in the IDLE state for the playback while after that, for the most part, we are in the play state. We see that we do not enter this state directly with the push of the play button. We expect that due to the preplay state.



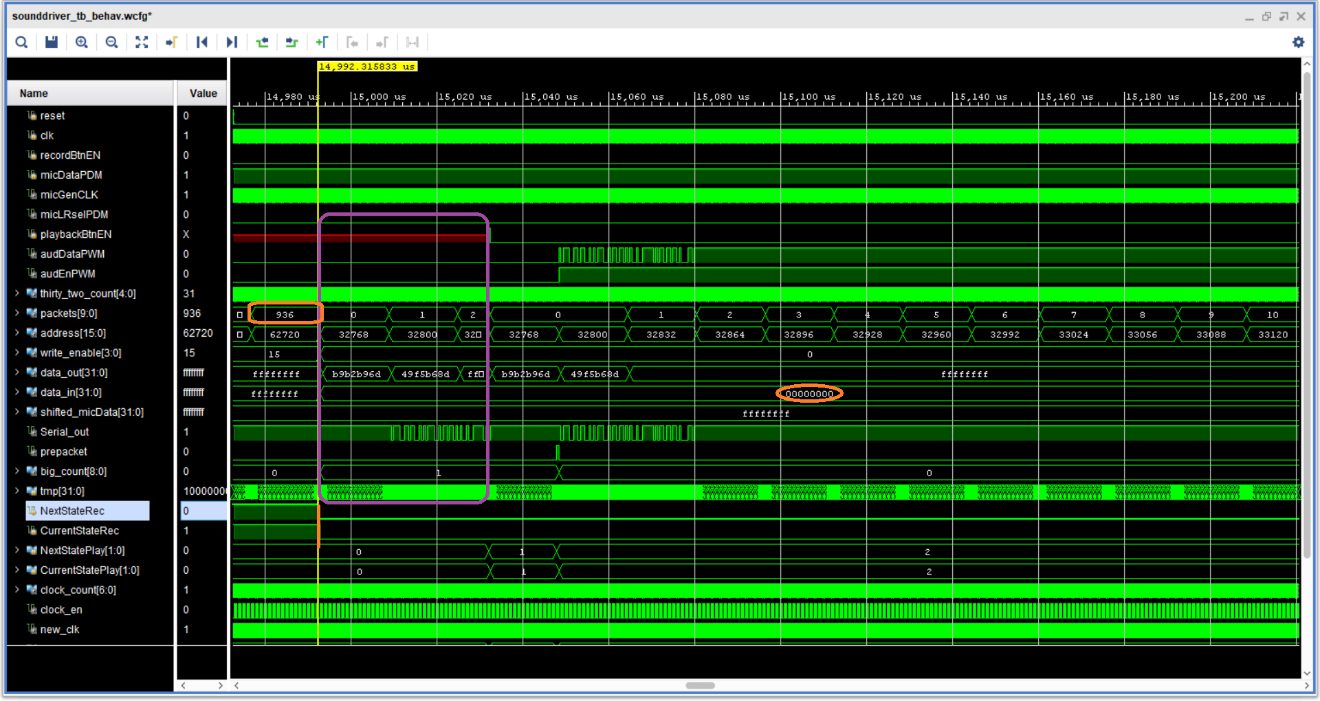
Let’s now take a closer look at the recording. We see in purple that the data from the microphone are built gradually through the deserializer so that at the final bit of the packet, we have the message ready for memory. That is the reason why the corresponding variables are red at the beginning because the message has not been built yet. We see that the 3rd packet has the value 32’b1111…1111. We also see the packets that start from 0 and after 32 bits, they increase by 1. The variable data\_in are the data going to the memory and taking its value from the variable shifted\_micData. The variable data\_out is the data read from the memory. Although we do not want to read the memory at this moment, due to the read enable that is always set to 1. However, we can ignore them since we don’t need them.



Now let’s zoom more into this screenshot.



With a closer look, we see that the message is completed at the last bit of the first packet, as well as that it is gradually building in the previous bits. We also see that the data going to the memory are the same as the data we gave in the testbench. We can also notice that the data\_out which is the data read from the memory takes the value of the previous cycle. That is also expected because the data need a cycle to be written to the memory, and a cycle to read them. So, if we ask to write a message to a particular address at the moment t0 and at the same time we want to read that address, this is what will happen. The message will be written at the moment t0 to the address and with the next clock we will read the data from this address that we just wrote. Now, let’s see the end of the recording.

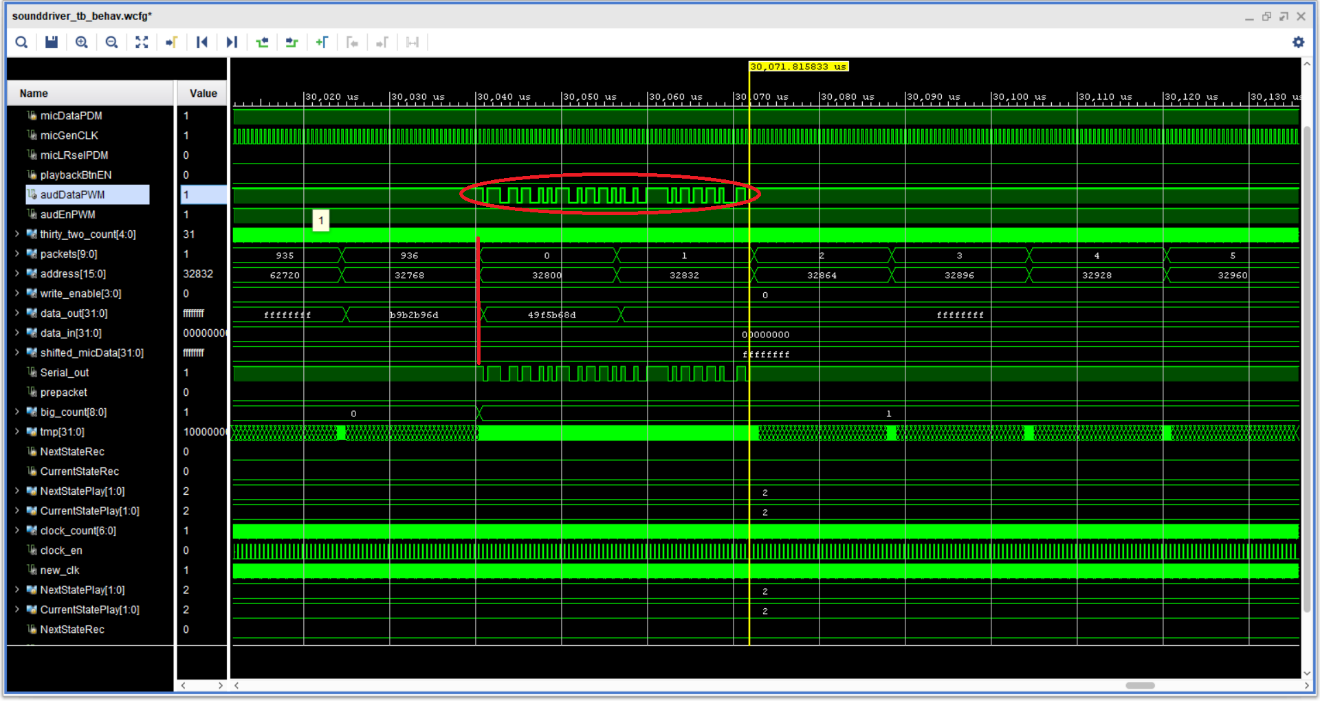


We see in orange that the recording ends indeed at the 937(0-->936) packet. We also see in orange that the data\_in takes the value 0, and not the shifted\_micData from the microphone. We also notice that write enable becomes 0. Finally, the value of the states of recording FSM also changes and goes to the IDLE state. In purple, are shown the other variables, i.e., the counters, the address, and so on. These variables may take various values, but that is not a problem, since we do not want to use them at that moment. Whenever we want to use them, we remember to reset them. Now let’s see what happens when we push the button to play the message.

Εικόνα που περιέχει κείμενο, ηλεκτρονικές συσκευές

Περιγραφή που δημιουργήθηκε αυτόματα

We see in orange all the states of the playback. More specifically we can see that before the yellow line, we were at the state 0 or the IDLE state and that with the pushing of the play button, we got to the preplay state. We can see that this state lasts only for 1 packet as we wanted as well as that the output during this state is 0. That is because we haven’t got yet the first data from the memory. We can also notice the pulse prepacket. This pulse is generated at the end of this state and affects the packets counter. More specifically, it resets the packet’s value to 0, so that we don’t lose any packets on the transmission. Moving on we change state and go to the play process. There, having the data from the memory we start serializing them and transmitting them to the output. We can see that the circled signal is the output that goes to the speaker, and we can ensure that these data are the same as we put in the memory through the testbench. Let’s now see what happens when we have transmitted all the packets from the memory.



We can see that after packet 936, i.e., the last packet, the playback doesn’t stop. That is true to happen because we wanted to reproduce the message for enough time. So that happens because we start again playing the message due to the big\_counter. Although it was impossible to simulate for 6 seconds that the playback lasts, I experimentally checked that the playback lasts indeed 6 seconds, and it does not run forever. Finally, let’s see in more detail what happens with the clock of 200MHz we should use, and I didn’t.

Εικόνα που περιέχει κείμενο, ηλεκτρονικές συσκευές, υπολογιστής

Περιγραφή που δημιουργήθηκε αυτόματα

Although I didn’t use that smaller clock, I made all the necessary changes to use it if I wanted. More specifically, I created a module to produce an “enable” pulse after 500ns/5ns = 100, 5ns pulses. At this moment would be the posedge 500ns clock that I want in the project. The only change I should do to use that smaller clock was to write instead of posedge clock, “posedge clock && enable = 1” and put the smaller clock. However, that is impractical as the only clock we work with precision/accuracy of the 500ns clock in the whole project. Except that as we see in the simulation, the posedge of the 500ns and 5ns clock has a slight difference. We see that because the red line and the yellow line do not match, but the one is a few ns before the other. The reason why that happens is probably that the 500ns clock needs the cascading mode of the MMCM to be produced, while the other clock does not. These reasons lead me not to use the small clock, although I made everything necessary to use it.

* **EXPERIMENT**

This part was tested experimentally. The synthesis and implementation didn’t show any errors or latches. Also, the bitstream was written successfully. First, I connected the board with the speakers, then I pressed the rec button and make some noise near the microphone. Then I pressed the play button to play the message. The result was white noise. That was the expected result and it lasted for 6 seconds as expected. Then I pushed the button again and then I pressed the reset button. This time I wanted to reset the system and stop playing the message. I also tested the system by recording different noises to see that each time the output was slightly different, according to the input.

1. **Conclusion**

The project was about the implementation of a recording guide and sound playback using Digital Modulation. Along with that, it was necessary to instantiate properly and use a BRAM to save the message coming from the microphone. The challenges of this project were mainly to understand how the peripherical devices work. More specifically, reading the manual for the BRAM and understanding the way it works was a challenge. I needed to make some simulations by myself after reading the manual to understand how memory actually works. The same happened with the cascading mode of the MMCM. Although we have used this again, it was something different to make a clock with the cascading mode of it. As for the implementation, good practice as always was to reset the counters whenever I wanted to use them. Finally, the project had initially a Part C. A technical problem occurred and prevented me from implementing it. As for the new optional part C, due to lack of time, I didn’t manage to work it, although I wanted to.