FPGA Telephone

6.111 Final Report

Fall 2015

Rumen Hristov & Alan Medina

Contents

1 Abstract

We built a two-way telephone system. We used several FPGAs, which are connected with a single wire to transmit data and simulate a phone call. One person can dial another FPGA and the other person can pick up and start a conversation. If the call does not succeed then the caller hears the pre-recorded greeting message of the person who he is calling. After that he has the option to leave a message. Later the other person can listen to the saved messages.

2 Overview

2.1 Project Overview

The final state of our the labkit of our project is shown on Figure 1. We programmed three labkits with an identical Verilog. Each labkit has an identifier that is programmed with the switches. We display the identifier on the small dot display. Each labkit can call the other telephone systems that are connected to it by pressing a button. We can specify which identifier we want to call with the help of the switches. With another button we can hang up or take the call.

We also have a voice managing system. With a the help of the buttons we can record a pre-recorded message. If we make the call and the person doesn't pick up then we will hear his pre-recorded greeting. After that we will have the opportunity to leave a voice message that he will later be able to listen by pressing button 0.

For all voice communication we use the microphone and headphones adapters of the labkit and the AC97 drivers. The communication between two labkits is achieved by using an output and input user pin. We use a small circuit to solve transmission conflicts and a single wire (plus ground wire) to connect all of the telephones.

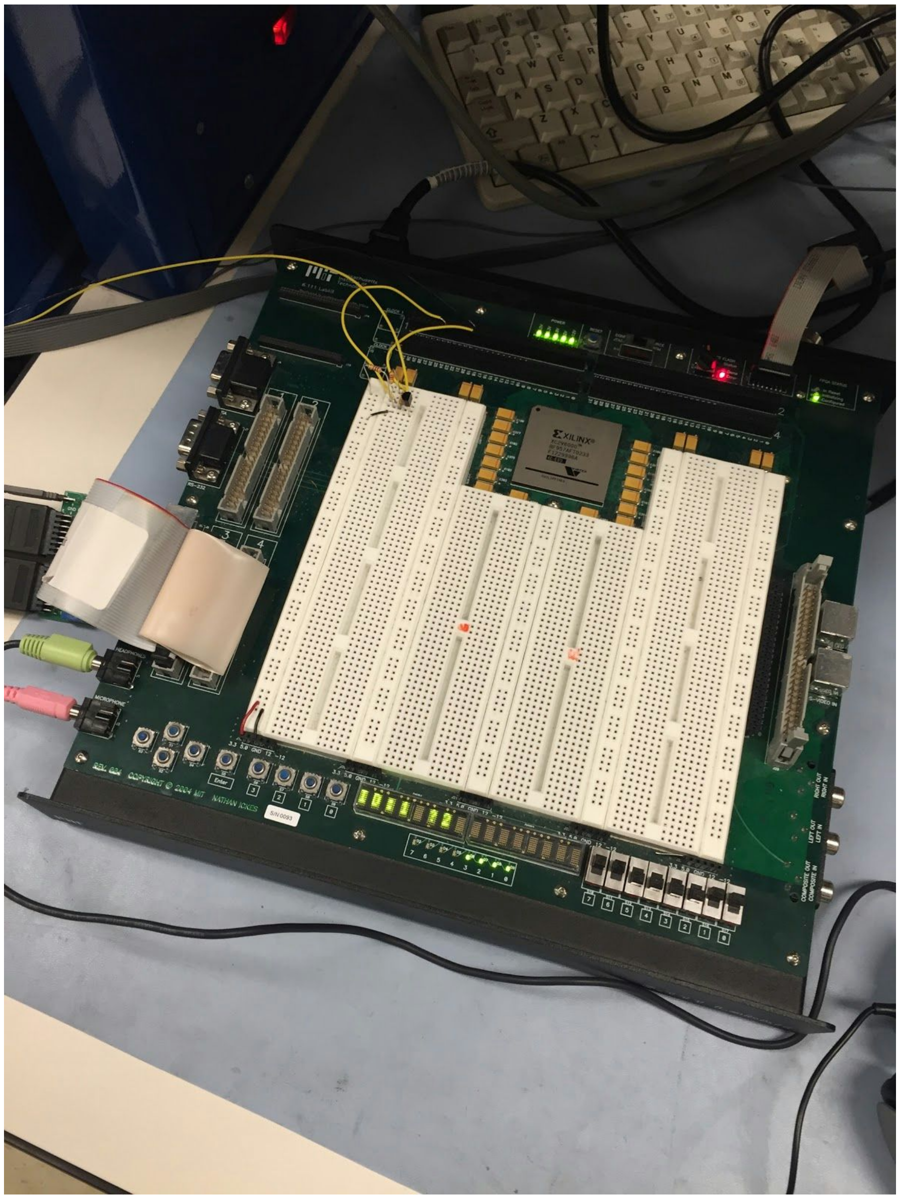

Figure 1. The labkit of a working telephone system.

2.2 Design Choices

2.2.1 High Level Design

At a high level, the system is a finite state machine that transitions into different states to perform different functions. In Figure 2, you can see the different possible states and which states can transition to the other states. More detail can be found in section 3.3, but the basic idea is simple. All of the other modules interact with the state machine. It then reacts to different inputs and the outputs will tell the other modules what functions need to be performed.

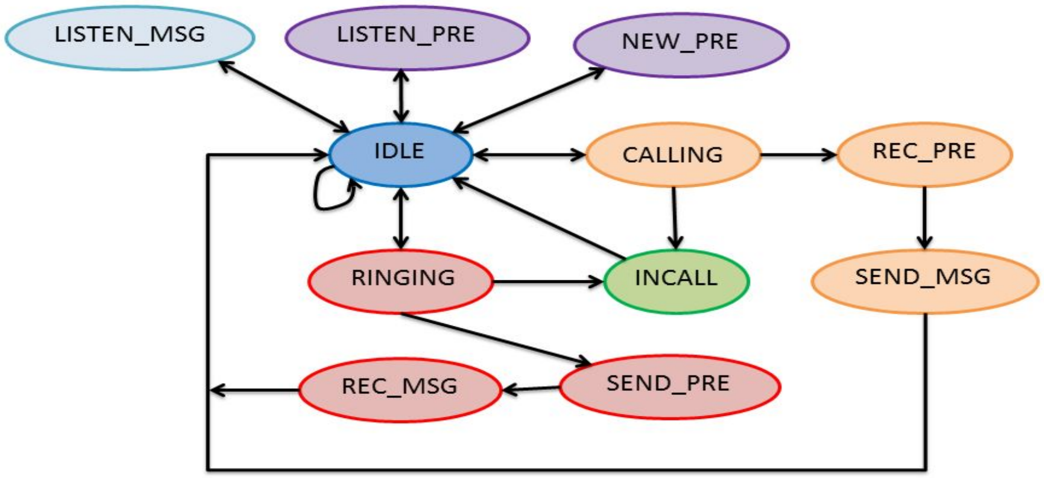

Figure 2: Finite State Machine for Phone System

Some important points to note from Figure 2 are MSG and PRE. MSG is the voice message that can be sent or received in the case that the phone is not answered. PRE is a pre-recorded greeting that will come from the memory. This is the message played to prompt a voice message (e.g. “Please leave a message after the beep”). The states that end in PRE or MSG are interacting with these.

2.2.2 Block Diagram

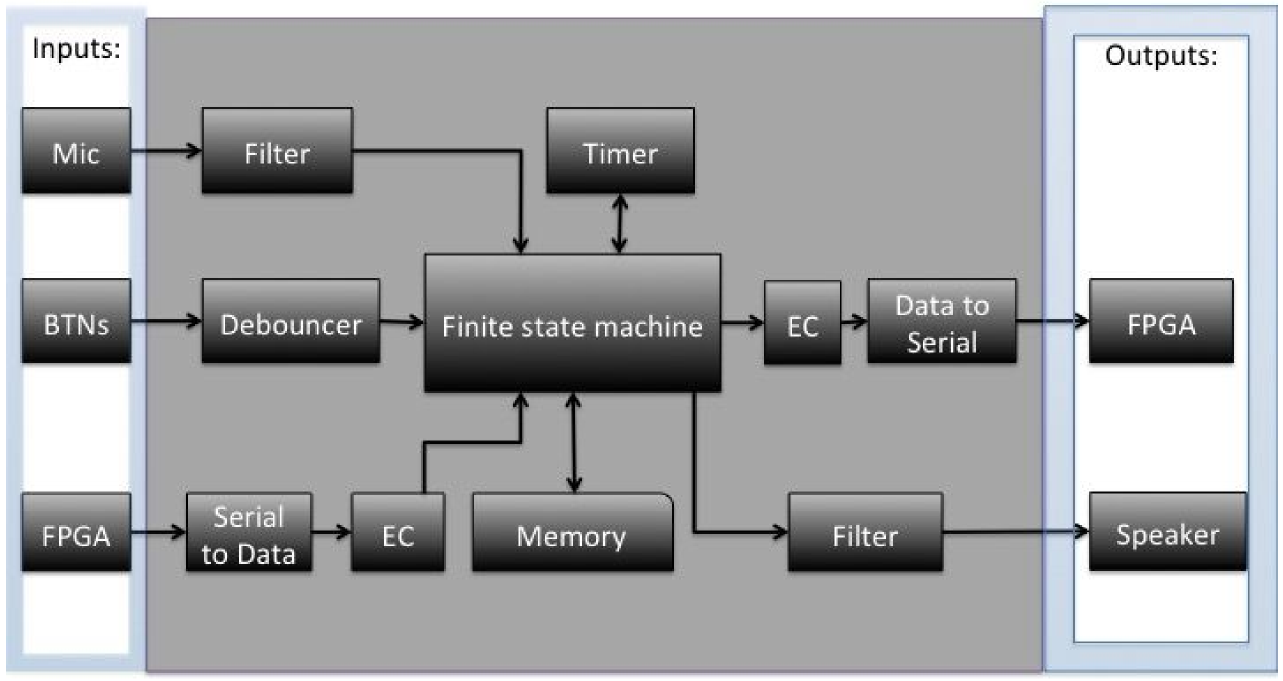

Figure 3. Block Diagram of the Telephone System.

Now that we have a sense of how the system acts, we can look at how all the different modules connect and interact with each other. In Figure 3, we can see that the finite state machine is at the heart of the system.

On the left side of Figure 3, we can see the inputs to the system. The microphone, buttons, and switches are very simple to work with since we used a lot of these in previous labs. The microphone uses the ac97 to give use digital audio. The buttons and switches are simple, but noisy, so those need to be debounced.

On the right side we can see the outputs. We are also sending data back to the ac97 so we can hear sounds from the headphones. Also not shown on the diagram is the display. The dot display on the labkit is used to show useful information such as your current ID and caller ID.

The FPGA component corresponds to the line that connects all the FPGAs. Sections 3.1.2 and 3.2 go into more detail on how this can be both an input and output. This is one of the more challenging parts of the project since we need to get a lot of accurate information to and from other FPGAs. We used to following packets scheme, containing 3 pieces of information:

- **address** [1:0] : Each FPGA has an unique address. If the packet address matches that FPGA's address, then it acts on that packet. Otherwise the packet is ignored.

- **header** [2:0] : This tells the receiver what kind of data is being sent. For example, we need to be able to distinguish between audio data for a conversation vs. the data for a message.
- **data** [7:0] : This is 8 bit audio data we get from the ac97.

3 Module Descriptions

3.1 Communication (Rumen)

In order to transfer data between the different labkits, we had to connect them with a wire. Because our goal was to have more than two telephones, we didn't want to use two directional wires. This would mean that every two labkits would need to be connected with a wire or have a chain of wires. We managed to do transfer information only with single wire between all of the labkits. First we started with an implementation with two labkits and two wires - one for each direction.

3.1.1 Two wires

In the two wire approach only labkit is writing to the wire at any given moment and only one is reading. A challenge is that our data is not a single bit and we had to serialize it and deserialize it. Also because the wire and the timing between the two FPGAs is not perfect we had to hold the signal for longer than a single cycle to make sure we read the correct information.

In the two wire approach we used two simple modules: send_data and get_data. Their main responsibility was serializing and deserializing a 13 bit packet. Each packet usually describes an audio sample from the ac97 and has specific header, described later.

The output module is taking a packet on a ready signal and starting the sending process. In order to avoid simple transmission mistakes, we first send a pre-specified 4 bit header, which is just the sequence 1011. This way on the receiving side we are sure that we are actually receiving data from the other FPGA and not some noise. When transmitting a bit, we held the signal for 8 cycles so the receiving FPGA can sample 8 times. In order to decide on the final output, we used a majority algorithm[1]. It is a very simple way to determine the number that has occurred at least half the times from n samples.

3.1.2 Single wire

In the single wire approach we had some complications. In this case we can have two labkits sending data in the same time. If one of them tries to send a low signal, while the other one a high signal we will get undefined behavior and current flowing from one labkit to another.

In order to resolve the problem, we built a small circuit, which we will explain in more detail in Section 3.2. The circuit ensures, that the input wire will be high, if any of the outputs from the labkits is high. If all of the output are low, then the input wire will also be low.

This small circuit allows us to have less interference between the two devices. For every labkit we have a global state, which is indicating whether we are in IDLE mode, SENDING_DATA or RECEIVING_DATA. We maintain an invariant, that if one labkit is sending data than all of the others will be receiving it at the same time. Again, we had two modules responsible for transferring the data. Because of the propagation delay of the circuit we had to hold the signal for more than 8 cycles (look at Section 6. for more details).

The global state does not allows us to send while receiving. If the state machine sends a message to the sending module, while we are receiving data, we can't start sending it right away. Even worse - the state machine can send several messages before the sending module has had a chance to act on any of them. In order to cope with this problem we implemented input and output buffers. These buffers can contain up to eight messages. When we want to send a message, we just add it to the buffer. If we are not receiving a message and we have a message in the output buffer, the sending module takes it and starts transmitting it.

The buffer hugely simplified our design, because the state machine did not have to worry about timing when sending the message. The extra slack helped us simplify the state machine for a little complication in the sending and receiving modules.

3.2 Hardware (Rumen)

In order to fix the outputting conflict with the wire, we had to create a small circuit. We needed a couple of a resistor and a BJT to resolve the conflicts in transmission. Figure 4. is a photo of the circuit in final state of the telephone. We just connect the output to an input of the BJT and the output of the BJT is connected to 3.3V with a pull up resistor and to the input of our phone.

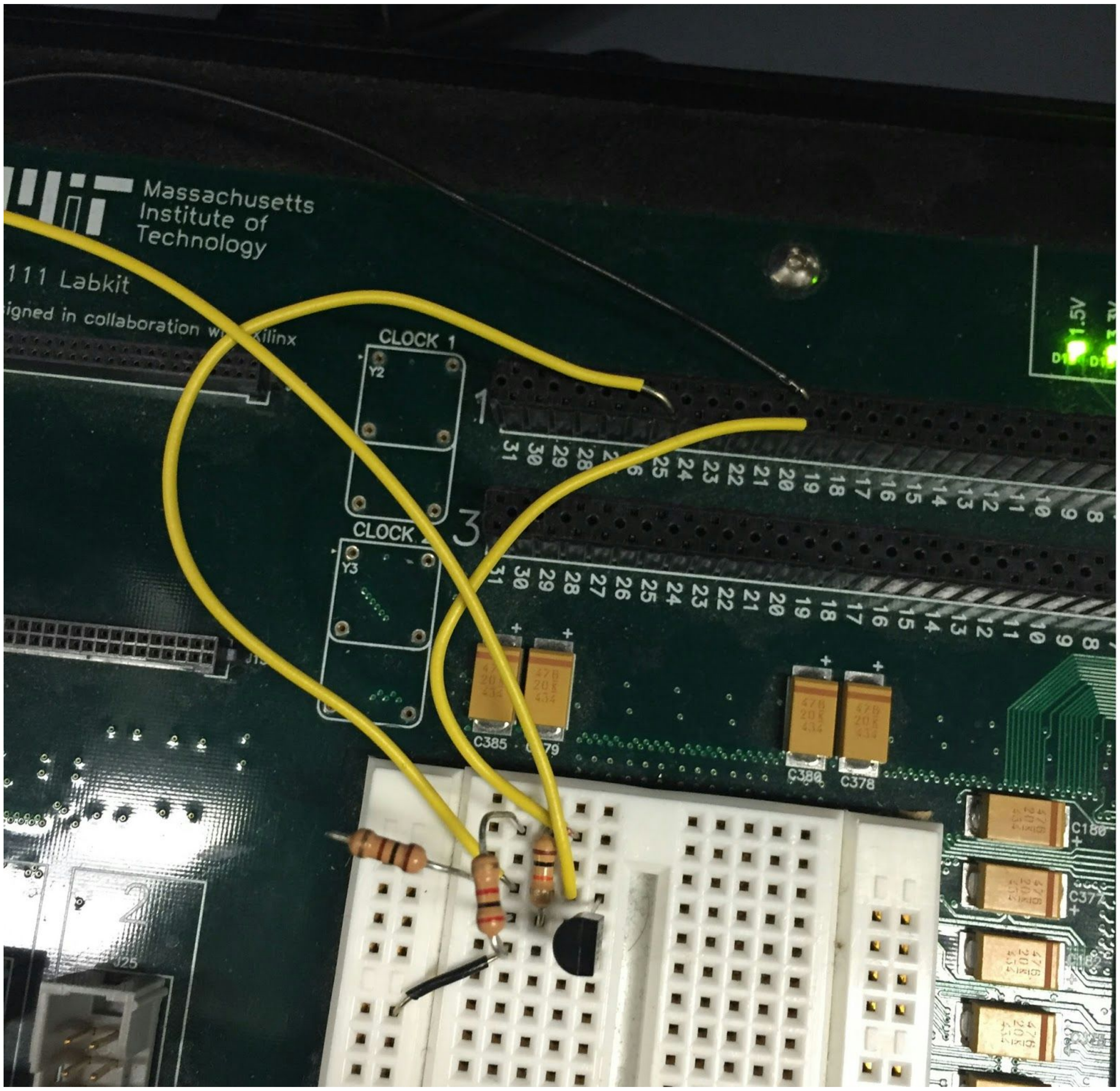

Figure 4. The conflict resolving circuit.

3.3 Finite State Machine (Alan)

The state machine is the central control of the telephone system. The function performed by the phone depends on the current state of this finite state machine. The default state is IDLE, and you can reach different states through different buttons the user can use or through packets the phone receives from other FPGAs.

To initiate a call, the user can select the intended caller using the switches and press the call button. This action changes to CALLING state and creates a packet with a specific header that the receiving FPGA will interpret as an incoming call. The receiving FPGA will enter the

RINGING state. While in RINGING, the user can push a button to pick up the phone and both FPGAs enter IN_CALL state. In this state, audio from the ac97 is downsampled to 6 kHz, put into packets, and given to other modules to be sent on the line. The received audio is upsampled back to 48 kHz and sent to the ac97 to play. When either FPGA hangs up, a specific packet is sent to let the other FPGA know and both will go back to IDLE.

The other possible state begin to get interesting, as they begin to use memory and more complex state changes. The simplest part of working with memory is setting a PRE, the pre-recorded greeting that gets sent if no one picks up. To set this message, the user presses a button to get to NEW_PRE. This state is a simple recorder that stores audio from your own ac97. When leaving this state, you enter NEW_PRE_END, where a half second beep is added to the end of your greeting, and then you go back to IDLE. You can then press a different button to enter LISTEN_PRE, so you can make sure it is correct. These state were also incredible useful for debugging(more on that in section 4).

Another situation we can get ourselves into is when a call is initiated, but no one answers. A separate timing module lets us know when too much time has passed in RINGING, so we enter SEND_PRE, and the other FPGA needs to enter REC_PRE. SEND_PRE looks into memory and sends your pre-recorded greeting while the other side simply plays back the sound it receives. From SEND_PRE, you will enter REC_MSG, and the other side will enter SEND_MSG. SEND_MSG sends audio from the ac97, while REC_MSG saves the received data into the next memory slot. Both FPGAs can then go back to IDLE.

To listen to a message that was left while you were away, you can select a slot using the switches and press a button to enter LISTEN_MSG. Depending on the switches, a different address in memory will be reached and played back.

3.4 Packet Reader (Alan)

This is a very simple module just to simplify some of the work of the state machine. This has the full received packet as an input, along with a ready signal. This module checks the address in the packet. If it matches the address of the FPGA the header and data are passed on to the finite state machine. If it the address does not match, it does not pass anything to the state machine.

3.5 Sounds (Alan)

The sound effects that are used by the phone are created here. This uses two other modules, a 750 Hz tone that I took directly from lab 5, and a modified version of this I used to create a 400 Hz tone(really 375, but it is close enough to 400). The IDLE state uses the 750 Hz tone, which is admittedly a bit high and loud. The CALLING tone alternates between 750 Hz tone and silence. The RINGING tone alternates the 750 Hz and 400 Hz tones.

3.6 Timer (Alan)

When a start signal is sent to this module, it starts counting seconds. The number of seconds counted depends on the state. It is mostly used by the RINGING state so that it doesn't ring indefinitely and is able to send a pre-recorded greeting.

4 Testing and Integration

4.1 State Machine(Alan & Rumen)

The state machine is very large and has a lot of different functions, so a pretty cautious approach was taken when building up this module. I started with the basic four state need to make a phone call; IDLE, RINGING, CALLING, and IN_CALL. These states were tested extensively in ModelSim to make sure they were performing proper functions and they were transitioning states properly. I didn't begin any other states until I felt comfortable with the results from simulation.

After this part, the rest of the states were finished up and there was more simulations in ModelSim. By the time the simulations were working well, we were beginning to get some progress with sending information between FPGAs. Some of the testing for the finite state machine was done alongside the testing for the communications.

We got some progress in debugging these together but we sometimes had trouble knowing if problems were caused by communication or by the state machine. To solve this issue, we used eight one-directional parallel wires connected the FPGAs. There wires communicated the headers and ready signal for packets. Having this setup ensure that the correct headers were getting across and the state machine bugs became more apparent. We went back to one bi-directional wire after this part of the testing.

4.2 Memory(Alan)

Getting the memory to actually work was a bit difficult at times, but testing it was fairly simple. The states NEW_PRE and LISTEN_PRE in the state machine act as a simple recorder, much like the one in lab 5. Once I got these two states to properly record and playback audio, I modeled all the other states that perform similar functions after these two.

When we started getting more consistent communication, I was able to test the messaging system. There were several problems that led to the final design of the project to only hold two messages, but given more time, that could easily have been improved since there was still a large amount of space left in the ZBT. Having more messages would require more addresses to be tracked. You would need this to know where in the memory each message is stored.

5 Modifications from original design

5.1 State Machine

The overall structure of the state machine stayed very close to the original design, with the exception of the NEW_PRE_END state. This state adds a half second beep to the end of your pre-recorded greeting. This could have been part of the NEW_PRE state, but it was just easier to make a separate state for this. It is a nice little feature to have and the user does not have to interact with this state at all.

5.2 Memory

The memory we used and the way we used it changed quite a lot as we found problems(more details in section 6). The original plan was to use flash memory, but there was a changes to BRAM and then to ZBT for the final version of the project.

The original design would have allowed us to store over 40 messages, and since it was in flash, they would be non-volatile. The final design allowed for two messages in the ZBT. The messages in the ZBT are also now volatile, so they are lost if you turn off the FPGA.

6 What went wrong

6.1 Memory

The first issue encountered with memory was writing to flash. Flash memory is great since it is non-volatile and large, but it's not RAM, so it is very particular in how you can write to it. When working with flash, ideally you will write to it once, or at least not very often, and then you are able to read from any part of it. It is not ideal for situations where you may be writing and rewriting to it often. When writing to it, it will only write sequentially, so you can't partition the memory like originally planned. Also, you can't simply re-write over other parts of the memory. It needs to be erased, but problems arise again since you can only erase blocks at a time. This led to the decision to drop the flash memory.

After switching to BRAM, the memory worked very well because of the simplicity of BRAM. We then wanted to switch to ZBT to allow for more messages if we got the chance, but the ZBT also brought along its fair share of problems. The biggest problem with this was timing. Since the ZBT is not inside the FPGA, but further away in the labkit, there is a clock skew that can cause timing problems. The sample code did a good job of showing how to make the timing work when working with video. I needed to take this code, and make it work properly with our clock. Luckily, the provided ramclock module worked well, so it was just a matter of understanding how it worked. The ramclock used a reference clock, in our case 27 MHz, and gave a new clock that would fix the clock skew when working with the ZBT. This new output clock, which we called clk, was then used as the main clock for our project.

It is important to understand all the pros and cons of each memory type before choosing one. We chose flash without fully understanding its limitations. This led to too much time being spent trying to get it work, when ZBT was a much better choice for our use. If we not spent so much time on flash, the messaging system on ZBT would probably be more complete and capable of storing many more messages.

6.2 One wire propagation issue

When we start using the one wire, we were experiencing very strange issues. There were a lot of errors in transmission and thus the packet header was not matching. We found the problem was that because we were sending each bit for 8 cycles on the line, the measurement on the receiving side was not accurate enough. We put an oscilloscope at the wire and we looked at the time needed to go from a transition in the output signal of the FPGA to transition in the input signal of the FPGA and we got the results from Figure 5. The blue signal is the signal after the BJT and the yellow one is the signal that the FPGA is sending.

We can see that it takes around 700ns to transition the receiving signal after we change the sending signal. We were holding the signal for only 8 cycles on a 27MHz clock. This is equivalent to only 296ns. This meant we were holding the signal for too short and the receiving end read the wrong bit. We had to increase the number of cycles in which we hold the signal to 32 to ensure that we receive the correct information. Also we ignored the first ~500ns (13 cycles) of the samples that we collect on the receiving end because we know they will be impacted by the propagation delay of our circuit.

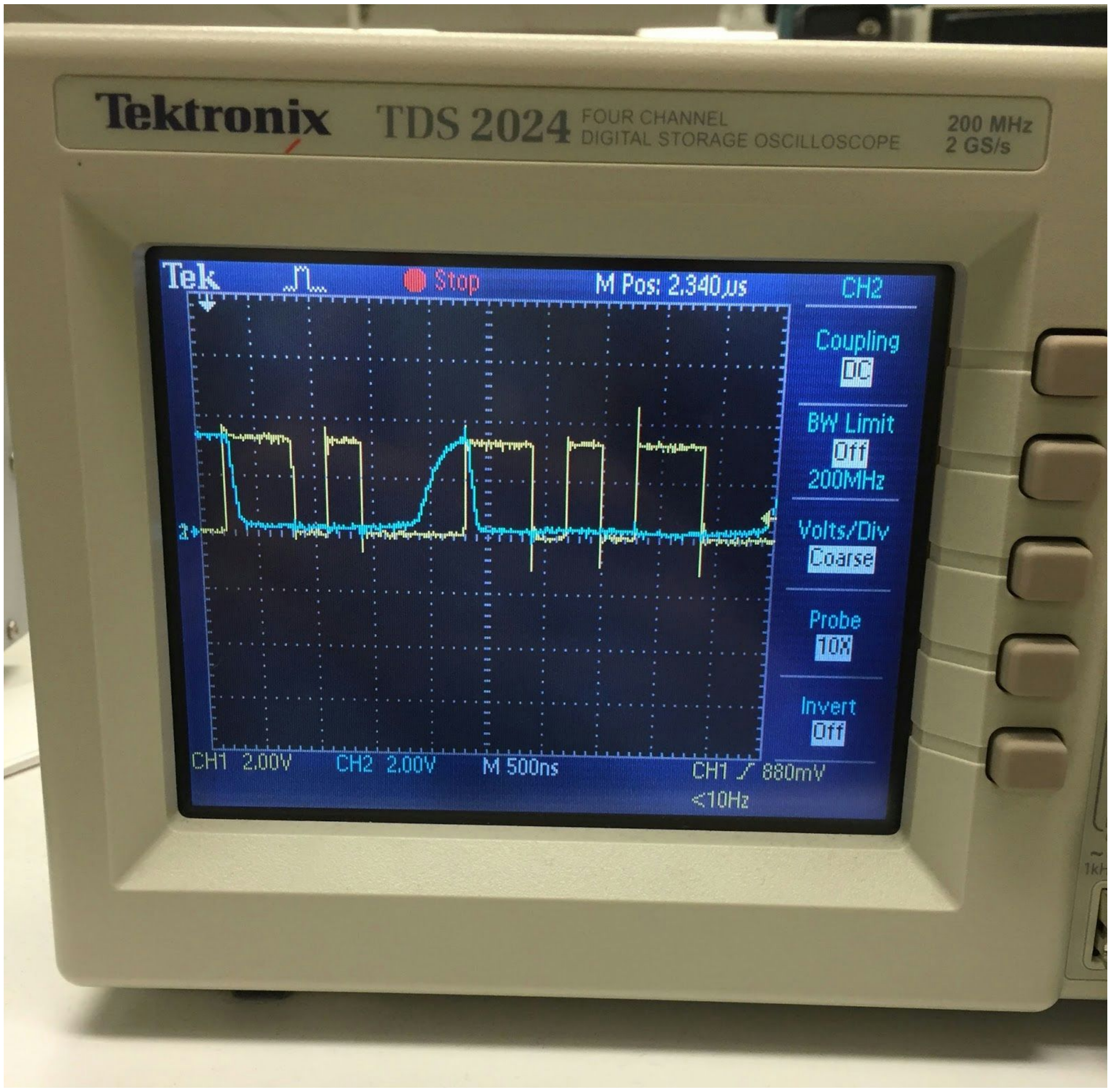

Figure 5. The oscilloscope reading of the wire between two FPGAs.

7 Conclusion

Overall, we are very pleased with the outcome of our project. Although there were a few problems along the way, there were parts of the project that went very well, or at least as expected. For example, the serial communication over a single wire was not easy, but once the bugs were removed, it worked surprisingly well and had very few errors. The success of this part of the project made it much easier to connect a third FPGA to the group. Correctly sent packets allowed good communication between two without the third interfering.

The finite state machine was also a very successful part of the project. This part was tough to debug, but we expected a debugging process like this. There wasn't any huge surprises and our final version was very close to what we planned in the original state machine diagram in Figure 2. The only state we that was different was a state to add in a tone that the user never has to interact with.

The memory was a surprisingly big issue. It didn't go as planned, but at least we did learn from it. We learned the advantages and disadvantages of the three memory types available in the labkit. Despite our problems we managed to save multiple messages(two) into the ZBT.

The display we used on the project was simple, but we thought it was great and made the project much better. On the labkit's dot display, we displayed your ID, the ID of the person you are trying to call, and even caller ID when receiving a call.

8 References

[1] http://www.cs.utexas.edu/~moore/best-ideas/mjrty/