Shared medium phone

Karim Liman-Tinguiri, Isabel Pesce Mattos, and Alexandre Oliveira.

Table of Contents

Overview 2
Sync Adder and Sync Remover (Karim) 3
The Sync Adder (Karim) 3
The physical layer – the MAX 485 transceivers (Karim) 4
The Sync Remover (Karim) 5
The Phone FSM (Karim) 7
Mic Wrapper (Alex) 8
Voice Buffer (Alex) 10
TCU (Transmission Control Unit) (Alex) 11
Parity “Stuffer” (Isabel) 13
Overview 13
Detailed implementation 13
Parity “Destuffer” (Isabel) 14
Overview 14
Detailed implementation 14
IDLE: 15
SAMPLING: 15
SAMPED_WAITING_FOR_TCU: 15
SENDING _TO_TCU: 15
How correction works? 15
Packet Analyzer (Isabel) 16
Overview 16
Detailed implementation 17
IDLE: 17
S_SAMPLING_PTYPE: 17
S_WAITING_FOR_TYPE0_TO_END: 17

S_TRANS_TYPE1_1 and S_TRANS_TYPE1_2: 17
Siren Generator and "I'm Calling Sound" (Isabel) 18
Summary 18
Table of Figures 19

Overview

Our project is a phone network that allows two-way phone calls as well as conference calls. Each phone FPGA has a unique 4-bit phone number. It is built on a shared medium – a pair of wires that every FPGA taps into. As a result every data packet is seen by every FPGA. We implement a mini network stack that consists of the following modules in the data path:

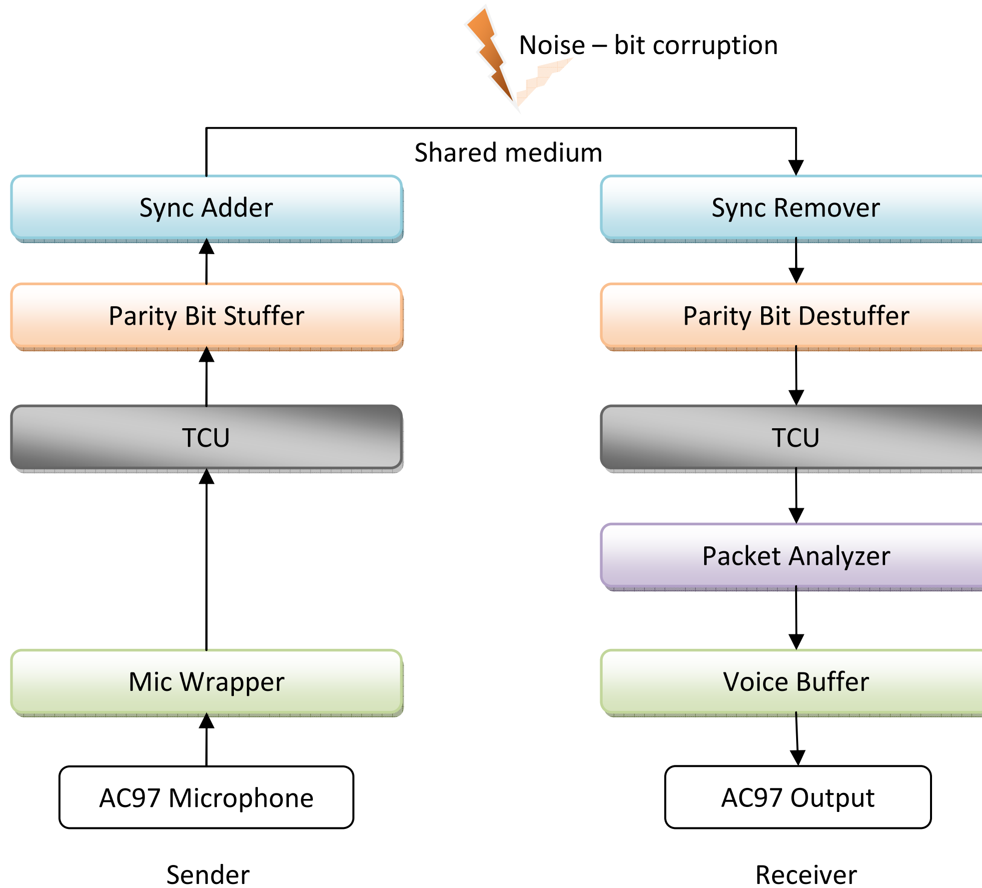


Figure 1: The half-duplex path of voice through our phone system

The above figure shows the modules in the data path between two FPGAs. Important module such as the phone FSM do not appear as they are not on the data path. We distributed tasks symmetrically; such that the person writing the sync adder also wrote the sync remover, the person writing the parity bit

stuffer also wrote the destuffer and the person writing the mic wrapper also wrote the voice buffer. This reduces the risk of errors due to different interpretations of the encoding.

Please keep in mind that since phone calls are full duplex, every sender is also a receiver and vice versa.

Sync Adder and Sync Remover (Karim)

Those two modules are separated by the shared medium. Every FPGA has a sync adder to write data to the common wire, and a sync remover to obtain data written to the wire. Because the cable is a serial link, the sync adder must serialize the data it receives, and likewise the sync remover must deserialize it.

Data is written on the wire in frames. Every frame contains a fixed 32-bit sync sequence that allows the receiver to detect the beginning of a frame and determine when best to sample. The next 11 bits after the sync sequence are the length field. They tell the receiver how long the frame is. Below are the inputs and outputs of the sync adder and the sync remover.

The Sync Adder (Karim)

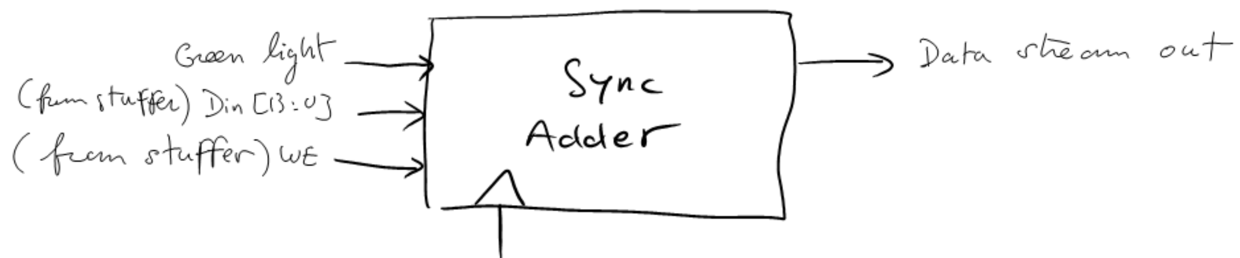


Figure 2: Inputs and outputs for the sync adder

Data can be written to the sync adder at any time. So we chose to add a “write enable” signal to it that the rest of the system can raise at any time when it want to write data to the wire. It must then provide a new 14-bit block of data[1] to the sync adder at every clock cycle until it decides to lower “write enable”. At this point, the sync adder’s internal buffer is filled with data but it does not immediately start writing to the shared medium (this would cause collisions). Instead, it waits for a green light input signal to go up telling it that it is ok to write. There are many ways to generate this green light signal. The simplest way is simply to negate the “busy” output from the sync remover (see the section “Sync Remover” below for more details). Lastly, the sync adder produces a write enable output that goes to 1 to tell our transceiver (the Max485) to start driving the shared medium. Below is a timing diagram showing how data can be written to the wire using the sync adder.

[1] We chose a block size of 14 bits because our samples are 8 bits and we add 6 parity bits in parity bit stuffer. Naturally, the block size a parameter and thus can be set to an arbitrary value.

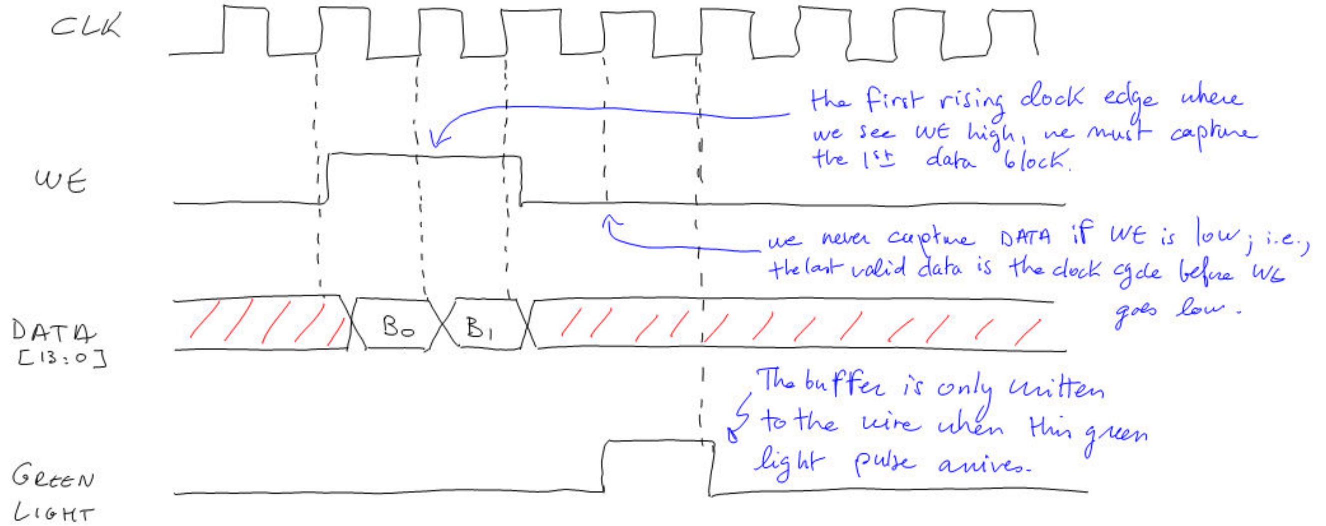


Figure 3: Timing diagram for a write event to the sync adder

The physical layer – the MAX 485 transceivers (Karim)

The MAX 485 transceivers are chips designed to operate on a shared medium using differential drive. Any chip can drive the medium at any time by raising the DE signal and everyone hears everything that goes on the wire. The same chip reads and writes, and unless specific measures are taken, every device hears what it is currently writing on the wire. If two devices try to drive the wire at the same time, collision happens. The topology that we used is given below, reproduced from the MAX485 data sheet.

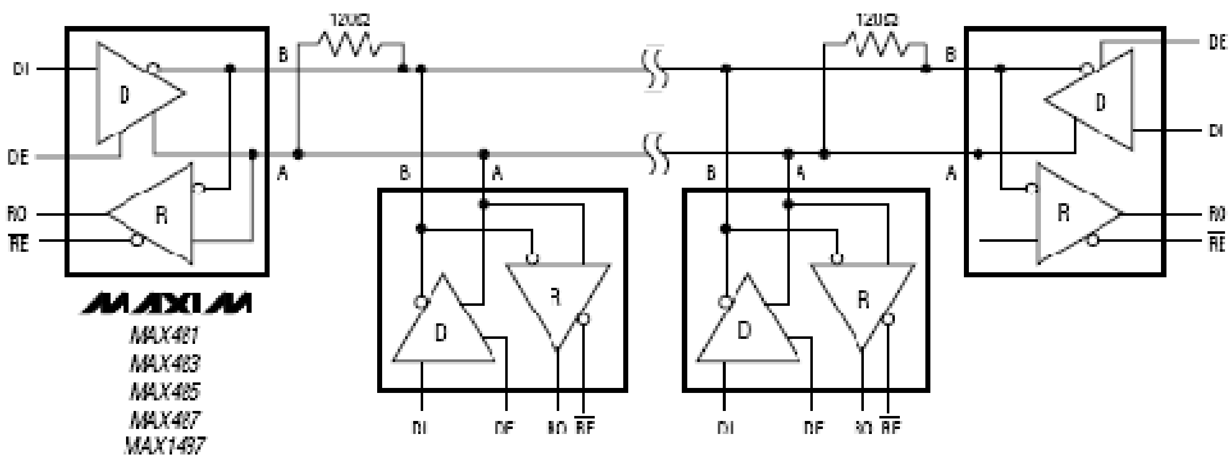


Figure 4: Network topology

The DI input is the one bit serialized data from the sync adder. The sync adder also controls DE, the pin that tells the MAX485 to start driving the wire. The RE bar signal is connected to ground, thus we are always listening to the wire (we discard our own echo in the FPGA) in case a sync sequence appears. The RO pin is connected to the sync remover.

The MAX485 devices are asynchronous and have a theoretical maximum data rate of 2.5 mbps which is $1/11^{th}$ of our clock rate. In practice, we were able to oversample by a factor as small as 8 (thus

transmitting at 3.4 mbps). Since the device is asynchronous, we were presented with multiple options to transmit clock information. We chose not to go with Manchester encoding but instead to use the sync sequence to pick the best moment to sample at (see the sync remover section below for more details on this) and keep the packets short enough that the clocks don't drift significantly. We had a few issues with clock drift early on when we were producing long packets. Shortening the packet sizes fixed the problem. The trade-off was that the shorter the packet size, the smaller the ratio of useful data to packet size (which includes the fixed-length sync sequence).

The Sync Remover (Karim)

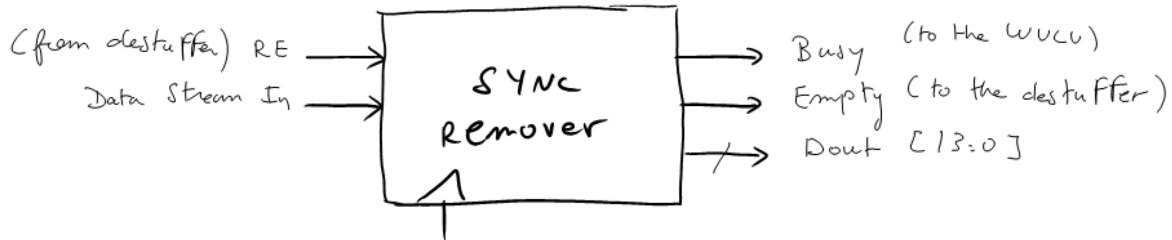


Figure 5: Inputs and Outputs for the sync remover

The sync remover listens to the wire for packets. Because packets can arrive at any time (and the system might not be ready to handle them), we use a passive signaling scheme. That is, when the sync remover has sampled a packet from the wire in its internal buffer, it sets its empty output to low and waits for the next module (the parity bit remover/destuffer) to raise its "read enable" before it outputs data (akin to a FIFO on the output end). It keeps writing the deserialized data one block at a time (a block is 14 bits) until it is done at which point it raises its empty output and goes back to listening to the wire as shown in the timing diagram below.

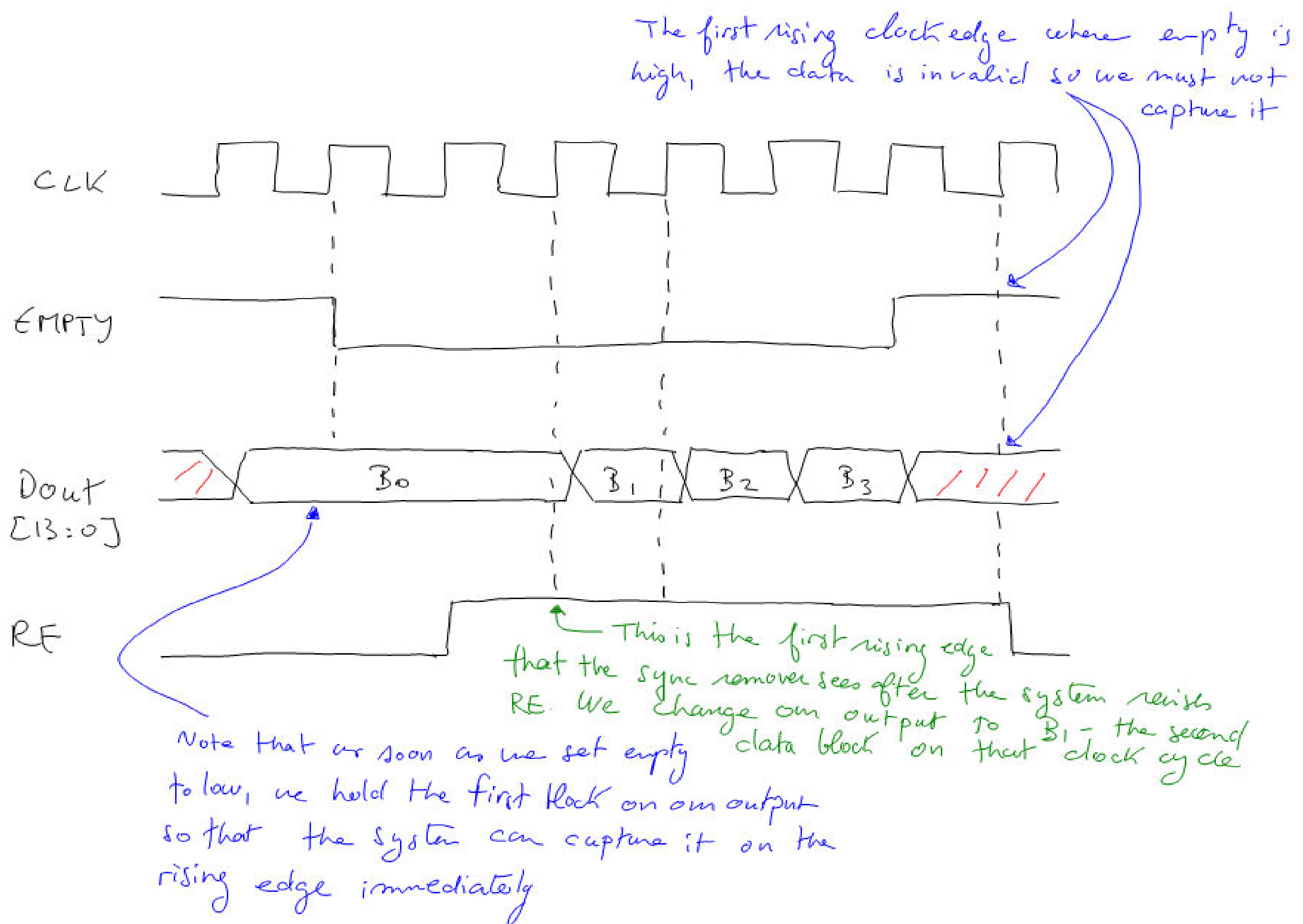


Figure 6: Timing diagram for the sync remover

The sync remover is substantially more complex than the sync adder. It continuously listens to the wire for a sync sequence. As soon as it hears the sync sequence, it samples the length, given by the next 11 bits. Note that the sync remover and the sync adder must use the same oversampling factor (the parameter in both modules needs to be set to the same value). Thus the largest possible frame size is $2^{11} = 2048$ bits. I implement the sync sequence detection using shift registers. Assuming the oversampling factor is 8, that means that every 8 clock cycles I capture one bit from the wire and put it in the shift register moving the previous 32 samples (32 because the sync sequence is 32 bits long) to the left. I then compare the content of the shift register with the magic sync sequence. If they match, I transition into the sampling length field state. During that state I sample the length field and then transition to the sampling data state where I sample the data for the length specified in the length field. I hand over the data to the rest of the system as per the timing diagram in Figure 6.

We cannot make any assumptions about the synchronization of the clocks of the FPGA when the sync sequence is sampled. Thus it is very possible that we sample every time the wire at an edge of the sync sequence. If this were to happen, we would not detect the sync sequence. To fix this problem, I actually sample the wire at two locations that are 45° out of phase by using two sets of shift registers to detect

the sync. That means that if the first set of locations was on an edge, then the second set of locations will not be on an edge and will show a valid sync. I then remember which phase to sample the data (in a register called "phase" in my code) on and assume that the clock drift is small over the packet's length.

The Phone FSM (Karim)

The phone FSM maintains the state of the system. It takes in all the physical inputs (current FPGA number, phone number to call, and the call/hang-up button) as well as signals about packets in transit from the packet analyzer. Based on that information and on some internal counters and timeouts, it produces outputs telling the mic wrapper (which is the only producer of packets) what packets to produce (e.g., ringing packets or calling packets), determines whether the siren is ringing, and whether the AC97 output should come from the voice buffer, or the "I'm calling sound generator." Below is a state transition diagram for the phone FSM.

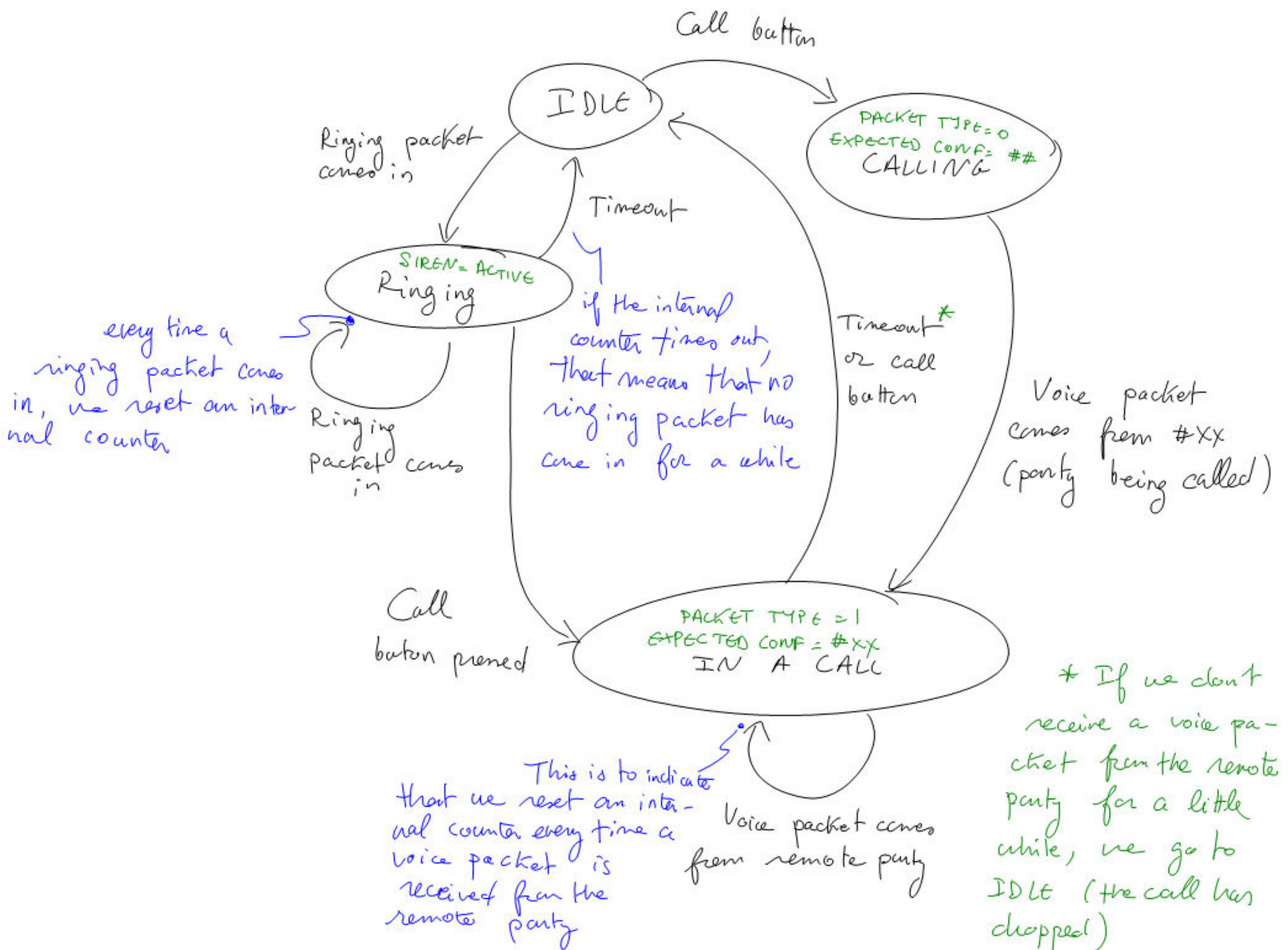


Figure 7: State transition diagram for the Phone FSM

Note that packets of type 0 are ringing packets (i.e., they tell their recipient to start ringing because it is being called) and type 1 packets contain voice data. Setting the packet type tells the mic wrapper what types of packet it should produce. The expected conference number tells the mic wrapper who to address the voice packets to and it also tells the packet analyzer which packets to discard (because they don't belong to the expected conference) and which ones to keep. The timeouts help ensure that calls are dropped when no packets are received for a little while.

Mic Wrapper (Alex)

The function of the Mic Wrapper is to generate packets. All the packets that flow through the network are generated by this module. The Mic Wrapper can generate two different types of packets: voice packets and calling packets. Voice packets contain a header and voice data coming from the microphone through the AC97. Calling packets don't contain any voice data, they just contain a header.

Here's a diagram showing the structure of a voice packet. The 'packet type' of these packets is always 0000,0001.

From (4 bits)	To (4 bits)	Packet Type (8 bits)	Voice Data (40 bits)

And here's a diagram showing the structure of a calling packet. The 'packet type' of these packets is always 0000,0000.

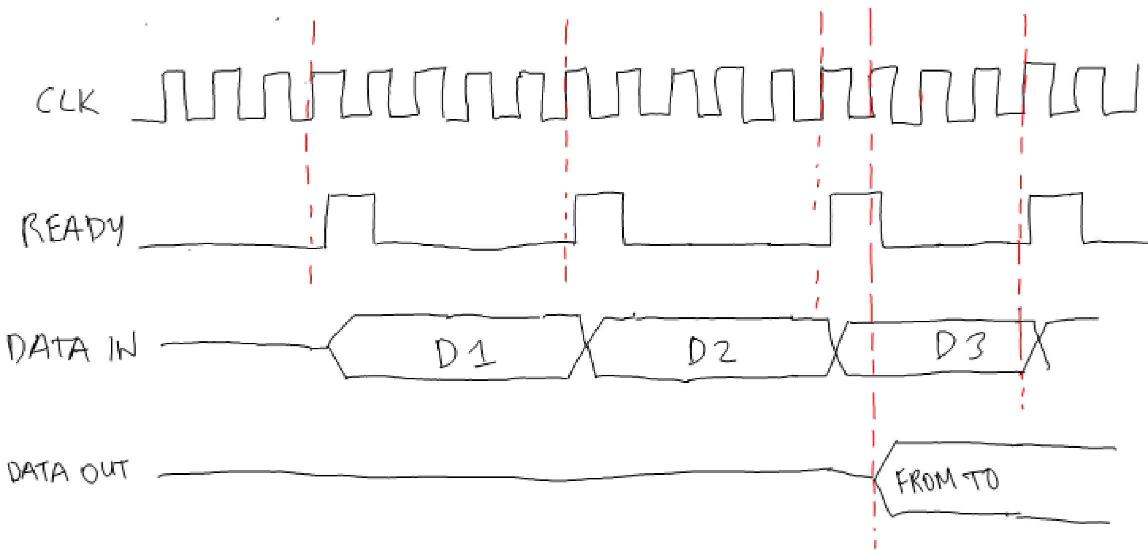
From (4 bits)	To (4 bits)	Packet Type (8 bits)

The instructions on what kind of packet to generate come from Phone FSM, the module that stores the state of the phone. The Phone FSM tells the Mic Wrapper the values to put in for 'for', 'from' and 'packet type' fields. Depending on the value of packet type received, the Mic Wrapper will decide whether or not to include voice data.

Here's a diagram showing the inputs and outputs of the Mic Wrapper.

Figure 8 - Block Diagram of the Mic Wrapper showing the inputs and outputs

And here is a waveform illustrating how the inputs and outputs are interrelated.



Continued waveform:

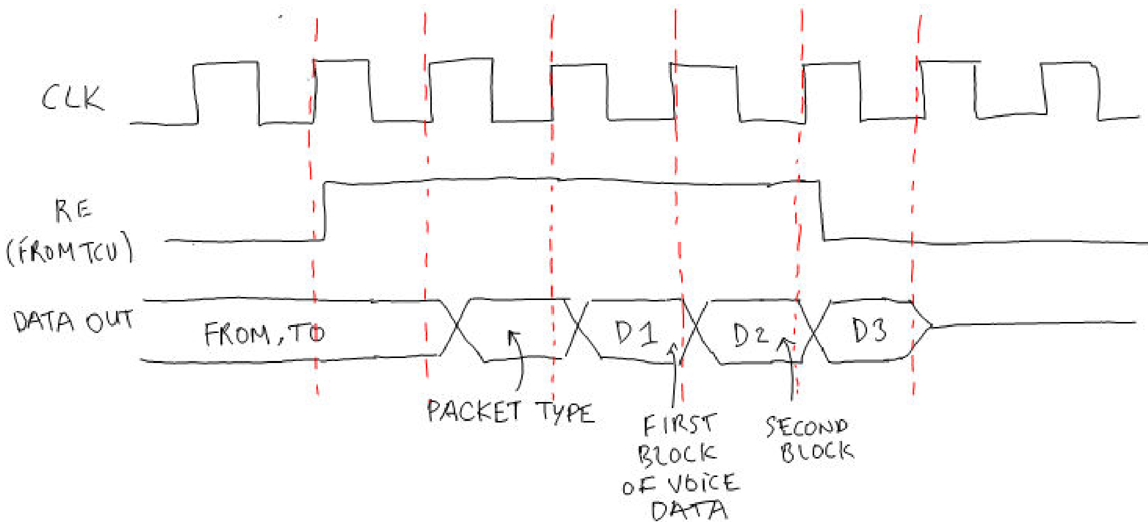


Figure 9 - Waveform showing how inputs and outputs are related

On the waveform above you can see that as soon as the Mic Wrapper has a frame of data ready, it holds the first 8 bits of the packet (the 'from' and the 'to' fields) ready for the TCU to read. This is so that the TCU can raise RE and sample on the following clock cycle.

The data out doesn't change until the Mic Wrapper sees the RE of the TCU go high. At this point, it changes to 'packet type' followed by all the voice data blocks it has. The clock cycle after the RE from the TCU goes low, the Mic Wrapper stops sending the data.

Voice Buffer (Alex)

The Voice Buffer's job is to get voice frames, extract the voice data and give them to the AC97 so that audio is heard on the headphones. It does this so that the burst of data coming from the TCU (a packet) can be played back smoothly at the rate that the AC97 asks for it. The AC97 signals that it is ready to receive a new sample of voice data by giving the Voice Buffer a 'ready' pulse.

Since our project allows for conference calling, the Voice Buffer needs to be able to get multiple frames and merge them. This is done by averaging the frames. If, for example, between 'ready' pulses we receive 4 frames, the Voice Buffer adds them together and divides by 4 when outputting.[2]

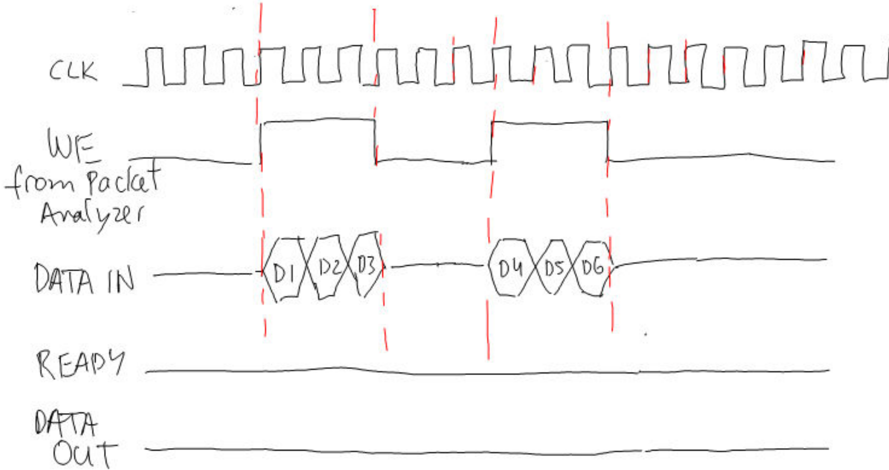
Figure 10 - Block diagram of the Voice Buffer illustrating inputs and outputs

The Voice Buffer has two storage places. The first is where it stores the running sum of the frames coming in. As frames come in, it adds them together and places them here. The second, is a buffer with a frame that is always ready to be outputted. If at any time the AC97 is ready, the Voice Buffer will have this frame ready to give it. Every time a ready frame is sent to the AC97, the data from the running sum is transferred to the ready frame so there's a new frame ready.

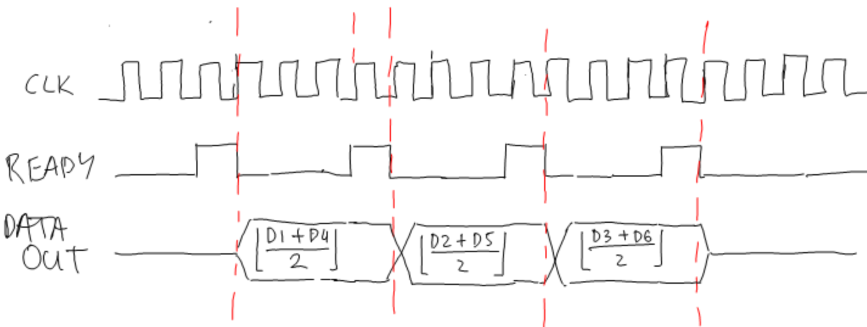
The packet analyzer is the module that gives the voice buffer the packets. Only voice packets come through so that the voice buffer doesn't need to worry about checking for the packet type.

Here's a waveform making how the signals interact more explicit:

[2] Since integer division is difficult to implement in hardware, we simple shift over data. This means we can only do division by powers of 2. We output approximate results by dividing by the closest power of two. If for example averaging required a division by 3, we would divide by 4 instead.



And here is the continued waveform



TCU (Transmission Control Unit) (Alex)

The role of this module is to relay packets across modules and to keep track of the next available conference number. The TCU always initiates the transmission of signals, meaning it tells other modules to start writing by raising the RE or starts writing itself to other modules by raising WE.

The TCU also keeps track of the 'next conference number'. If the phone wants to start a new conference, 'new conference number' will be the number of this new conference. TCU keeps track of this number by always reading the header of the packets it's transmitting. Since the input to our phone is a shared medium, the TCU can read every single packet being sent in our network. We let the 'next conference number' simply be the maximum conference number seen in the shared medium plus 1.

Here are the inputs and outputs of the TCU:

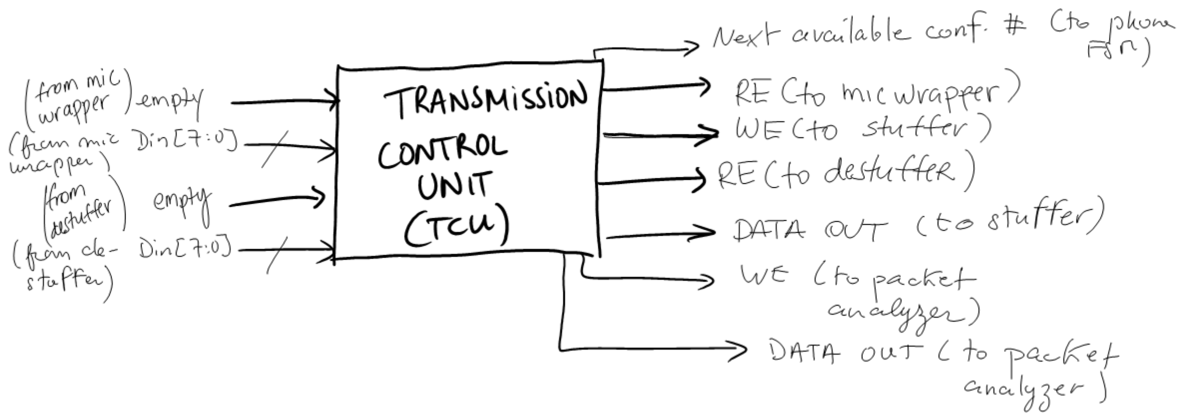


Figure 11 - Block diagram showing the inputs and outputs of the TCU

In the diagram above we see that the TCU gets data from the Mic Wrapper. The TCU waits until the empty signal from the Mic Wrapper is low, raises RE on the following clock cycle and starts transmitting data. The data from the Mic Wrapper is transmitted to the Parity Bit Stuffer. Once Mic Wrapper's empty goes low, the TCU lowers its RE on the following clock cycle and stops transmitting.

A very similar process is done for the data coming from the de-stuffer. The only difference is that the data is instead routed to the Packet Analyzer.

Here are timing diagrams illustrating this transfer of information:

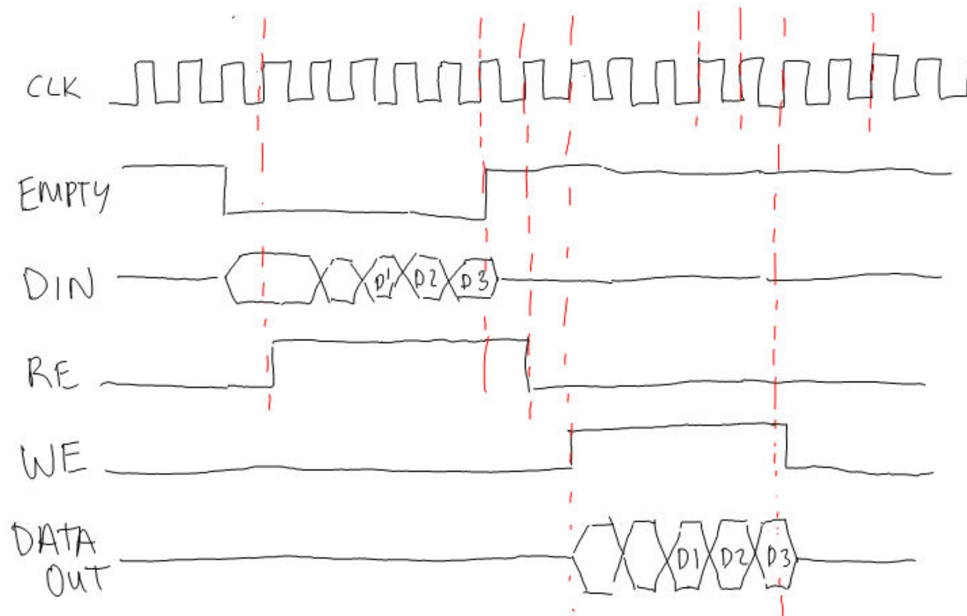


Figure 12 - Timing diagram for the TCU. The din stands for the data in from either the Mic Wrapper or the Parity De-Stuffer. The output is respectively either the Parity Stuffer or the Packet Analyzer.

Parity "Stuffer" (Isabel)

Overview

In order to prevent data corruption, this module adds parity bits to the data coming out of the TCU. It takes blocks of 8 bits of data, and adds 6 parity bits to it. It uses an even parity scheme: all bits in each row or column have to add up to an even number. For example, for a block of data 00110110 (in white), it adds the following parity bits (in blue):

0	1	1	0	0
1	1	0	0	0
1	0	1	0	

Table 1: Parity bits layout

Detailed implementation

Figure 13: Inputs and Outputs for the Parity Bit Stuffer

The data in comes from the TCU, in buses of width 8. The data out leaves in buses of 14 bits. The lower 8 bits of Dout is defined by the original data. In order to define the 6 higher bits, the stuffer adds parity bits to the data in, by "xor"ing all the bits in a column or row. This way, the parity bit will be 1 if all bits in a certain column add up to an odd number; the parity bit will be 0 if all bits in a certain column add up to an even number. With this strategy, we will always achieve an even number of 1's when adding up the "1"s of a column or row that includes the parity bits. The parity bits are added every positive clock edge.

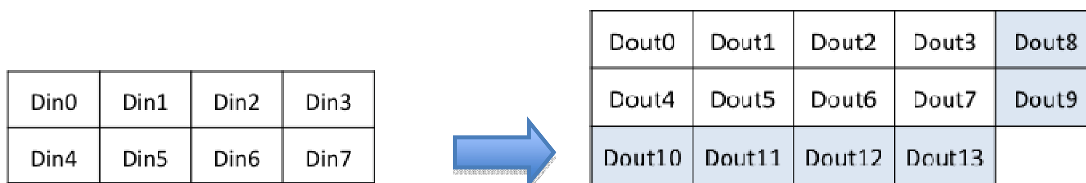


Figure 14: Parity bit augmentation process

One detail the stuffer also takes care of is delaying the WE from TCU for one clock cycle. This is important to maintain the modularity of our system. Since all the parity bit "stuffing " happens in one clock cycle, by delaying the TCU's signal by one clock delay, we are effectively simulating what the TCU->SYNC ADDER connection would be one clock cycle later, but still ensuring that the sync adder receives correct signals.

Parity "Destuffer" (Isabel)

Overview

This module buffers in an entire frame of data coming from the Sync Remover, attempting to correct errors on-the-fly. Should the Parity "Destuffer" encounter an uncorrectable number of errors, it drops the entire frame.

Detailed implementation

Figure 15: Inputs and Outputs for the Parity Bit Destuffer

The module uses a state machine. These are the possible states:

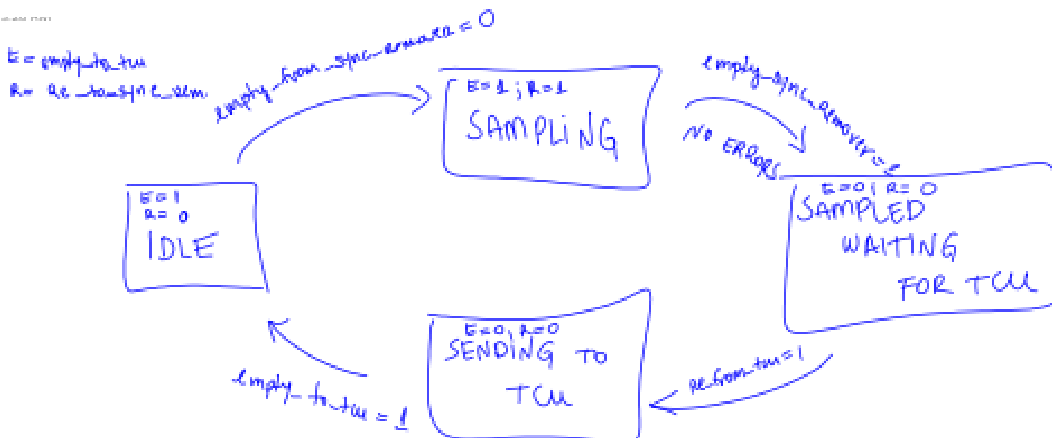


Figure 16: State transition diagram for the packet analyzer

IDLE:

The module is in IDLE state whenever there is no data to be sampled from the Sync Remover. That is, the empty signal from sync remover is high.

SAMPLING:

The module is in SAMPLING state if the Sync Remover has data ready (empty is low). When in this state, data is being read from the Sync Remover (re_to_sync is high), but there is still no data ready for the TCU (empty_to_tcu is low). In this state, there is a check for the number of errors a packet has. If it has more than a critical number (1 in this case) in a 8-bit packet, then the whole frame is dropped. If there was 1 error, the module corrects it. After correction of a whole frame, the system transitions to "SAMPLED_WAITING_FOR_TCU." During SAMPLING, the data is stored to a bram.

SAMPED_WAITING_FOR_TCU:

After sampling data, we want to make sure that all data sampled is written to the TCU. Therefore, before we start sending the sampled data to the TCU, we need to make sure that the TCU is reading from us. This state exists while the TCU has not raised its RE. During SAMPED_WAITING_FOR_TCU, there is no extra data being sampled from the Sync Remover (re_to_sync is low). The TCU know the Parity Destuffer is ready because its empty is low.

SENDING _TO_TCU:

Whenever the TCU is reading from the Parity "Destuffer" (re_from_tcu is high), the sampled data stored in the bram is written to the TCU. When "Destuffer" is done writing to the TCU, it sets its empty to high, and transitions back to IDLE.

How correction works?

The even parity is checked by "xor"ing bits of a certain row or column. For example, if it gives you a "1" in a column, it means that the bits add up to an odd number. This way, we know that at least one of the bits in that column have an error. By "xor"ing the bits for all rows and columns, we can have the combinations that would lead to an error in certain bits.

For example, if there was an error in the first column, and also an error in the first row (assume it's a single-bit error), then we know that the corrupted data must be D1, and we can just flip its value:

D1	D1	D2	D3	D8
D4	D5	D6	D7	D9
D10	D11	D12	D13	

Table 2: Layout of the data with the parity bits

Packet Analyzer (Isabel)

Overview

This module receives packets from the TCU and determines the packet type, from and to fields, and feeds this information to the Phone FSM. There are only two types of packets: callings packets and voice packets. All packets are at least two frames long: the first frame contains information on the "from" and "to", and the second holds information about the package type. Therefore, when sampling data, it only takes two clock cycles to have information about type package available to the Phone FSM.

Figure 17: Inputs and Outputs for the Packet Analyzer

It's important to note that, no matter the type of package, the Packet Analyzer always send a one clock-cycle-long pulse to the Phone FSM whenever a frame comes in.

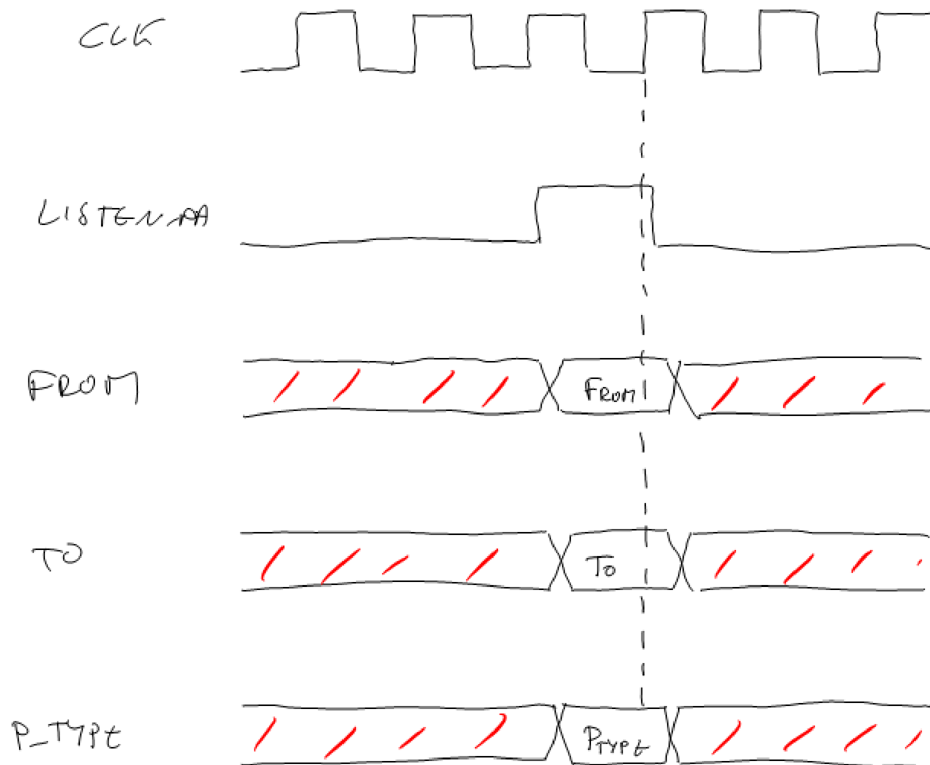


Figure 18: Timing diagram for the packet analyzer

Detailed implementation

The module uses a state machine. These are the possible states:

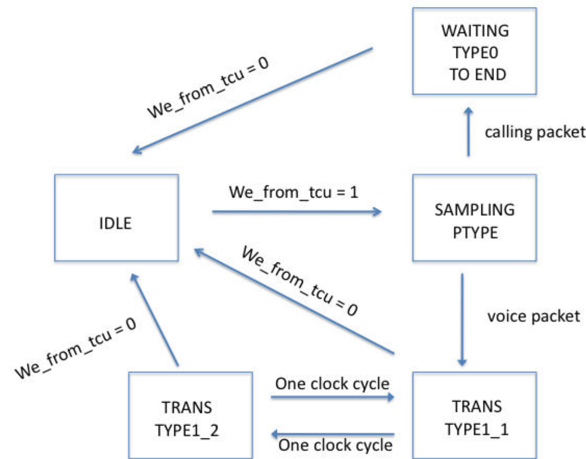


Figure 19: State machine for the packet analyzer

IDLE:

Extracts the information of the first frame: “from” and “to” (which is the same as the conference). Then, it transitions to S_SAMPLING_TYPE. Note that we decided to make the “to” always a conference number (Phone FSM also takes care of the details of that).

S_SAMPLING_PTYPE:

Extracts information on the packet type. If it’s a calling packet, transitions to S_WAITING_FOR_TYPE0_TO_END, which is just a one clock cycle state to take care of finalizing sampling calling package before going back to IDLE. If it’s a voice packet, then there are two cases, depending on whom the packets are to. If the “to conference” field matches up the expected conference number, then data start being stored in a bram, and we to the voice buffer is raised to notify the start of data transmission (transitions to S_TRANS_TYPE1_1). If they don’t match, then there is no need to store data in the bram, since the voice package is not for that “to”. For this case, we just have to finish sampling, so we transition to S_WAITING_FOR_TYPE0_TO_END for a clock cycle, and then go back to IDLE.

S_WAITING_FOR_TYPE0_TO_END:

Exists in order to make sure that last 8-bit block to be analyzed in a frame is taken care of. Effectively, it delays going back to IDLE by one clock cycle, because this way we can get correct information on the sender and conference when in IDLE.

S_TRANS_TYPE1_1 and S_TRANS_TYPE1_2:

These 2 states are very similar. In both of them, voice packets are being transmitted to the Voice Buffer. The reason why we need both states is because I use a bram with only 2 entries, and, upon storing Dout, I keep alternating the data between bram0 and bram1. Therefore, Packet Analyzer is either writing to

bram0 and reading out bram1 (S_TRANS_TYPE1_2), or writing to bram1 and reading out bram0 (S_TRANS_TYPE1_1). Whenever the TCU finishes writing to the packet analyzer, it goes back to IDLE.

Siren Generator and "I'm Calling Sound" (Isabel)

For generating the siren, we fast alternate between 400Hz and 700Hz. For generating the I'm calling sound, we follow the North American specifications found at http://en.wikipedia.org/wiki/Ringback_tone. It sums a 440 Hz tone with a 480 Hz tone (what can be easily accomplished by "or"ing the tones) and applies these following a 2 second on and 4 second off cadence.

Summary

Overall the project was a great learning experience. Despite the large effort we put into planning, we used up every hour of the buffer time we had allotted. A disproportionate portion of our time went into debugging because some of the initial test benches were not thorough enough, and it seems like there were some communication issues on what the inter-module communication specs are (especially about the first clock cycle where data is valid and the last clock cycle on which it should be sampled).

We made extensive use of both Modelsim and the logic analyzer. One of the issues we had is that we underestimated the time it took from a working simulation (two phones talking to each other in Modelsim) to the same scenario implemented on the FPGA with actual voice, real data corruption etc. A number of settings that are very convenient for simulation (for example, putting very few samples per packets and very high data rates) were not practical for the actual phone. Also, despite being extremely careful not to perform multiple read/write accesses on our register arrays, it seems like the tool did not realize that those register arrays could be mapped to a Bram. This is probably due to it not realizing that the states of our system are mutually exclusive. This caused very long compilation times as it tried to route thousands of bits worth of registers across the chip. Some unexplained bugs were solved by turning the FPGAs off and on after hours of investigation. Lastly, it did not draw our attention that divisions (required to average samples for conference calls) could not be done in one clock cycle until we were trying to run our project on the actual chip (Modelsim does not complain about that). The error messages that ISE 8.x (the version installed on the lab computer) were sometimes very cryptic. Recompiling the project under ISE 10.x gave us much more meaningful error messages. We debugged the last few issues related to the actual hardware by routing signals we wanted to look at to the logic analyzer.

Other than that, our phone performed surprisingly well, exceeding our expectations in terms data link reliability. We also learned a great deal about team dynamics and the need to write detailed specifications with carefully sketched timing diagrams to minimize misunderstandings.

Table of Figures

Figure 1: The half-duplex path of voice through our phone system 2
Figure 2: Inputs and outputs for the sync adder 3
Figure 3: Timing diagram for a write event to the sync adder 4
Figure 4: Network topology 4
Figure 5: Inputs and Outputs for the sync remover........ 5
Figure 6: Timing diagram for the sync remover........ 6
Figure 7: State transition diagram for the Phone FSM 7
Figure 8 - Block Diagram of the Mic Wrapper showing the inputs and outputs 9
Figure 9 - Waveform showing how inputs and outputs are related........ 9
Figure 10 - Block diagram of the Voice Buffer illustrating inputs and outputs........ 10
Figure 11 - Block diagram showing the inputs and outputs of the TCU 12
Figure 12 - Timing diagram for the TCU. The din stands for the data in from either the Mic Wrapper or the Parity De-Stuffer. The output is respectively either the Parity Stuffer or the Packet Analyzer. 12
Figure 13: Inputs and Outputs for the Parity Bit Stuffer........ 13
Figure 14: Parity bit augmentation process........ 13
Figure 15: Inputs and Outputs for the Parity Bit Destuffer 14
Figure 16: State transition diagram for the packet analyzer 15
Figure 17: Inputs and Outputs for the Packet Analyzer........ 16
Figure 18: Timing diagram for the packet analyzer 16
Figure 19: State machine for the packet analyzer 17