FPGA Telephone Exchange

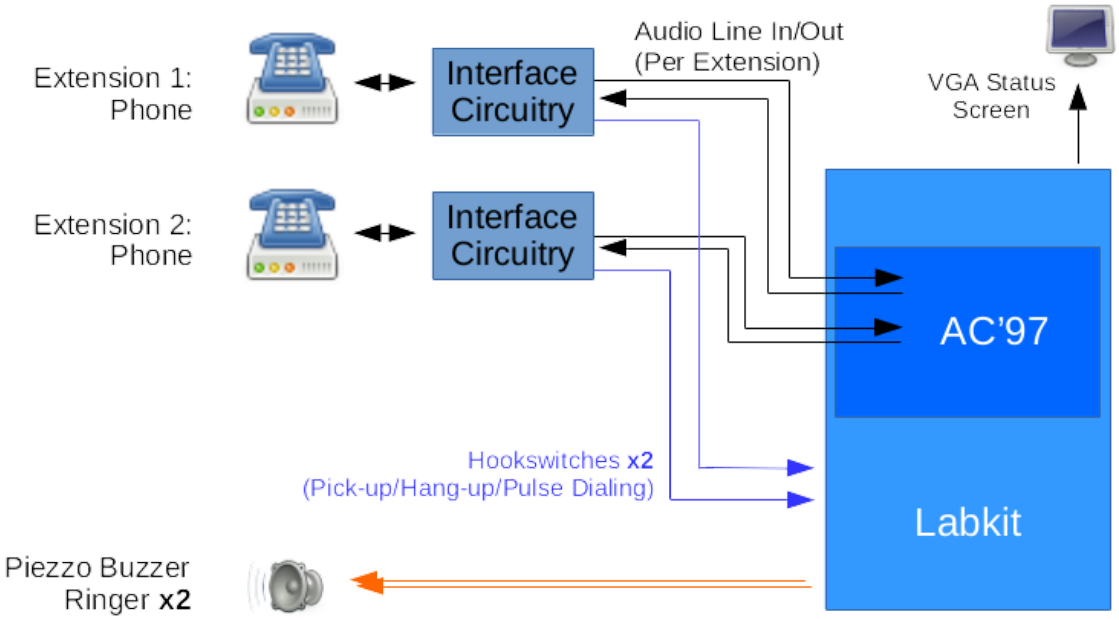
6.111 Introductory Digital Systems Laboratory
Final Project Report
Fall 2016

Tristan Honscheid

Introduction

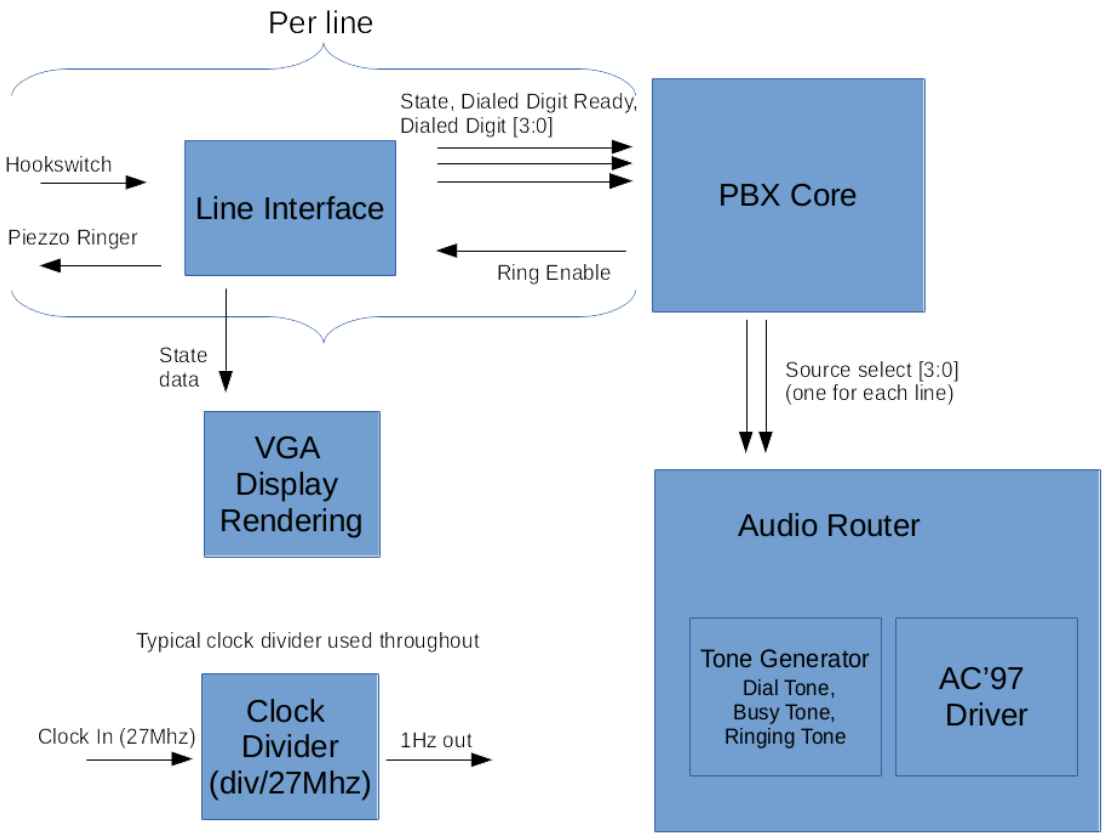
The FPGA telephone exchange projects implement an automated, dial phone system using commodity landline telephones. Users can pick up the phones, hear real dial and busy tones, dial other extensions, and receive calls, just like the the real phone network.

In my project, two landline telephones, combined with custom interfacing hardware, will be connected to the labkit to act as two extensions. The FPGA operates a state machine capable of decoding dialed numbers, playing dial and busy tones, and connecting calls between the attached lines. Each line has one input and one output channel on the labkit's AC'97 audio codec acting as a 'virtual switchboard' so that two phones in a call can be digitally connected by passing audio samples accordingly. Finally, a VGA display shows status information about the system, such as which phones are in use.



Landline telephone technology has seen little change in the past one hundred years. This makes a typical phone difficult to incorporate into modern digital systems. Therefore, each extension has a custom interface circuit that adapts the high-voltage, analog phone circuitry into FPGA- and labkit-friendly signals. Specifically, this external circuitry powers the phone, detects when users pick up and pulse-dial the phone, and provides a four-wire audio input/output signal pair for sending and receiving audio signals. Additionally, external piezo buzzers act as ringers, one for each extension.

Inside the FPGA, numerous modules handle audio processing, VGA output, and various state machines for tracking phones and calls.



Each extension/line has a *line_interface* module that performs the low-level phone interfacing. By monitoring the hookswitch input from the interface circuitry (see next section), this module's state machine outputs decoded dialed digits and a "phone active" signal when picked up. It also includes a siren tone generator for driving a piezo ringer at the standard North American ring cadence of 2 seconds on, 4 seconds off.

The audio router is the FPGA's virtual switchboard. It brings in four audio streams from the AC'97 codec (Extension 1 in, Extension 1 out, Extension 2 in, Extension 2 out) and maps them according to source select inputs. It also contains tone generators that play call progress tones, like dial tone, from BRAM memory. The audio router can accomplish tasks like "send dial tone to Extension 1, and silence to Extension 2" or "connect Extension 1 to Extension 2 and 2 to 1 in order to connect a call." Whatever comes out is what the user hears in his or her phone.

The *pbx* module is the heart of the phone exchange. It tracks each phone's status and dialed numbers using a state machine per line. It commands the audio router to connect two extensions when a call goes through and play then appropriate tones to each phone.

The VGA module shows status information about each phone. It accesses a character ROM in BRAM to create a text display based on information received from the line interfaces and PBX module.

Finally, a series of utility modules like clock dividers and debouncers are used throughout.

Why?

I chose this project because it fulfills a long-lasting fascination with phones and communications systems. I built many intercom systems as a child, and at one point convinced my neighbor, who worked for AT&T, to collect some retired equipment and help me install an old business phone system in my house. (I could call my sister's room when dinner was ready, or, better yet, make an announcement over loudspeakers I stashed in my home's air vents) Later, in high school, I wanted to build my own system from scratch. I got pretty far but never finished it. Although this system has virtually no aspects of its implementation in common (the original used microcontrollers), the knowledge gained was valuable.

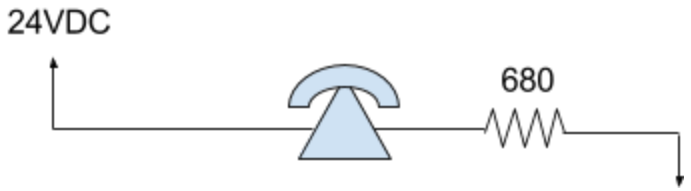
Hardware

As previously alluded, landline telephones are not easily interfaced and require special circuitry. This section goes into depth on the development of this hardware.

Brief Background on Telephone Electronics

Telephones support many features, including two-way audio transmission, dialing, ringing, and receiving power. All of this is accomplished through an ingenious, albeit old-fashioned, two wire interface.

In order to first begin using a phone, it must be powered. Real telephone lines use -48VDC, however, much lower voltages are sufficient. I use 24VDC. A series resistor sets a modest current, about 25mA. This circuit will power a phone and allow a user to operate it.



Any phone system must detect when the phone is picked up. Picking up creates a DC current path through the phone by closing the *hookswitch,* the switch that the phone's receiver depresses when hung up. By placing an opto-isolator in series, we can get an indication that current is flowing. This is called the "off-hook" state.

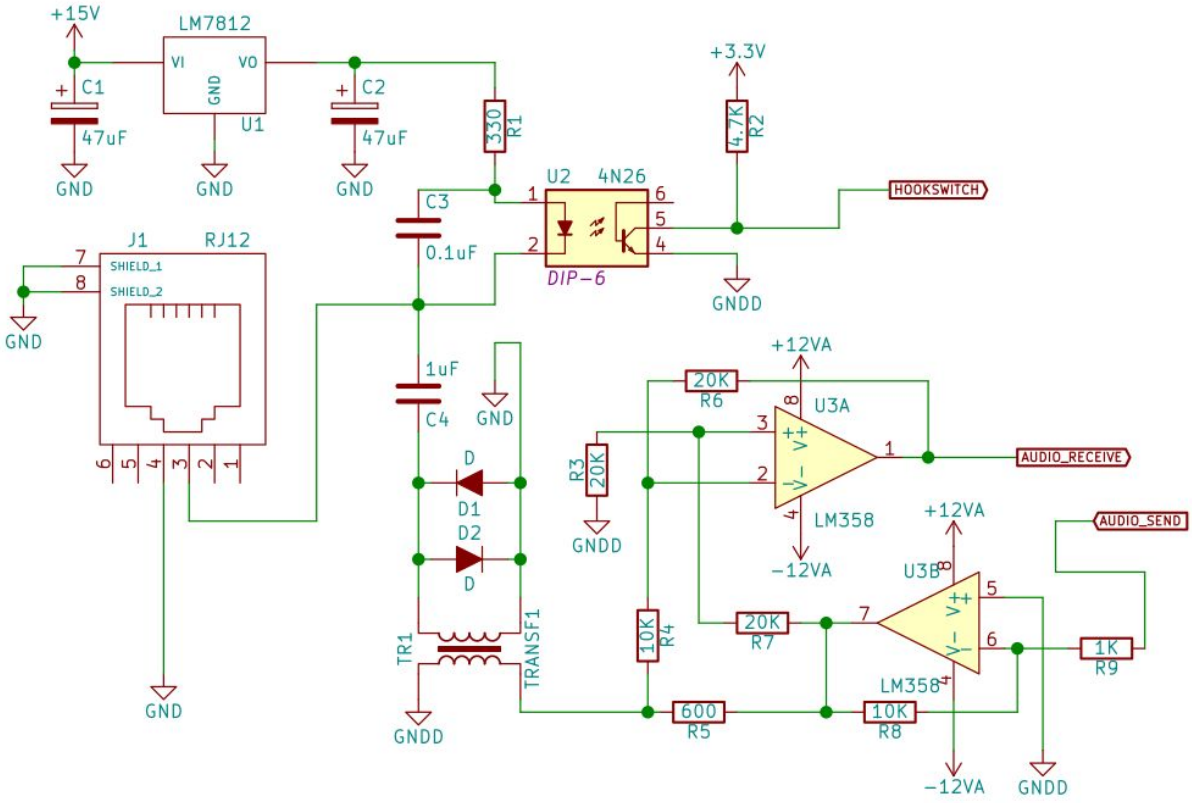
This also give us support for pulse-dialing. Pulse dialing is an older dialing mechanism where the hookswitch is pulsed to signal digits. Dialing a "1" briefly pulses the hookswitch once. "2" twice. "3" three times, and so on. By monitoring and timing the pulses coming out of the opto-isolator, a phone system can count the pulses and determined what got dialed.

Aside: modern phone networks now use DTMF (dual-tone multi-frequency) dialing, where the phone generates and sends an audio waveform corresponding to a digit. For example, a "5" is dialed by sending a 770Hz+1336Hz tone. Many phones and networks still permit pulse-dialing for backwards compatibility, but this is fading.

The Interface Circuitry

The project's telephone circuitry builds off of this to also send and receive audio. In the above circuit, audio is riding on top of the 24VDC power supply voltage (the phone modulates the sound onto the line with a transformer or some other solid-state circuit). This large DC offset must be eliminated using a capacitor. The audio is then passed through an isolation transformer to keep the high-voltage phone circuitry separate from the low-voltage digital electronics.

Finally, the audio needs to be split from a two-wire circuit to a four-wire circuit. Right now, all audio is mixed on one bi-directional signal, making it difficult to send and receive simultaneously. A circuit called a telephone hybrid is used to remedy this. The basic theory of operation is to use one op-amp to drive the transmitted signal into the line, and use a second op-amp as a differential amplifier to amplify the line minus the transmitted signal to extract pure received audio. From here, the audio connections go straight into the labkit's external audio ports.



The circuit is built on an external breadboard and is powered by the labkit's +/- 12VDC supply.

Implementation

The op-amp circuit requires careful tuning of the resistors, particularly R7 and R3 to effectively remove all transmit audio from the receive output. A loading resistor of 680 ohm might also be necessary parallel to the phone-side of the transformer. Careful tuning is critical. Even small traces of the transmit audio being allowed into the receive signal path can cause terrible noise and feedback when two sets of this circuit are connected back-to-back, as they would be during a call. Fortunately, I was able to mitigate this with a trim potentiometer for careful adjustment. Many hours of debugging, possibly more than any single Verilog module, were needed here.

Overall, this circuit provides a digital 3.3V hookswitch signal to the FPGA and two line-level audio connections terminated by RCA jacks.

A true telephone system would also support ringing the bells within the phones as opposed to using external piezo buzzers. However, ringing is complex and requires a 50-90Vrms 20Hz power source applied to the phone line in series with the DC source. (The odd frequency is due

to the resonance of the LC circuit formed by the mechanical bell ringer and a certain AC-coupling capacitor in the phone; ordinary 60Hz will not make a sound). Not only is this clumbersome to generate, it also requires additional circuitry called a *ring trip*. This circuit stops the AC ringing current once the user picks up the phone. This is critical, because 90VAC applied to an off-hook phone will blow out the earpiece (empirically verified). I successfully built such a circuit in high school, but it was not easy. For the sake of simplicity (and safety), ringing was left out of this project.

FPGA Logic

Line Interface

Port Type	Name	Description
input	`clock`	27 Mhz system clock
input	`reset`	Reset signal
input	`hookswitch`	Raw hookswitch signal from interface circuit. Passed through `user3` header.
input	`ring_enable`	Signal from rest of system that this phone should ring. Held high for duration of ringing.
input	`enable_1sec`	High for one clock cycle once a second.
output	`phone_active`	Goes high when the phone is picked up and low when put down. It filters out pulse-dial digits.
output	`dial_digit [3:0]`	A 4-bit BCD output indicating the most recently-dialed digit. 10 is sent for “0”.
output	`dial_digit_ready`	Goes high for one clock cycle to indicate that the user has completed dialing the digit. It is safe to latch `dial_digit`.
output	`ringer_driver`	Output to the piezo buzzer. It contains a sweeping tone with the proper 2-sec on, 4-sec off cadence when `ring_enable` is high.
output	`State_out [3:0]`	State machine debugging output

The line interface provides a layer of abstraction for each extension/line to simplify further processing and logic. In addition to clock and reset, the chief input is hookswitch, which comes

from the opto-isolator in the interface circuit via the user3 I/O header. It is inverted in the top-level module so that a logic high indicates off-hook. A logic low indicates hang-up or a pulse-dial digit.

Within the module is a state machine for monitoring the phone. Below is a description of each state.

1. **STATE_IDLE:** In this state, the phone is not in use. `phone_active` is set low to indicate to the rest of the system that this phone is available to receive calls but should otherwise be ignored.

   However, once hookswitch goes high (because the user lifted the phone), we transition to a different state. If we are not being called (`ring_enable` is low), we assume the user wants to initiate a call, so we transition to `STATE_DIALING`. Otherwise, we jump straight to `STATE_IN_CALL`.

2. **STATE_DIALING:** When the user wishes to place a call, they must dial an extension number. In this state, we wait for `hookswitch` to go low again and start a timer. If the `hookswitch` signal goes high again within a set timeout (approximately 500ms), it is assumed a pulse-dial digit was received and `dial_digit` is incremented. However, if `hookswitch` stays low for an extended period of time (1 second), it is assumed the user just hung up and we go back to `STATE_IDLE`. Once the first dialing pulse is detected, another timer starts to enforce a dialing timeout. After 2 seconds, dialing is deemed complete and `dial_digit_ready` is asserted for one cycle. We then go to `STATE_IN_CALL`.

3. **STATE_IN_CALL:** In this state, the user is connected to the phone network, either speaking with somebody or waiting to connect. Hanging up (`hookswitch` goes low) simply returns to `STATE_IDLE`.

This state machine captures dialed digits and also provides the useful `phone_active` signal. This is distinct from the raw `hookswitch` signal because it is held high throughout dialing. This is important, because `phone_active` is used by the remaining logic to determine if this extension is busy (i.e. unable to receive a call). Without it, if somebody else were to call this phone while this user was dialing, it is possible that the phone could be deemed not busy when the user was actually in the middle of pulse dialing. Thanks to this module, no pulse timing is needed anywhere else.

The line interface module also handles the simulated ringing, which is independent of the state machine. A submodule called siren (my car alarm from Lab 4) generates a sweeping tone suitable for driving a piezo element, outputted on `ringer_driver`. It is enabled by the logical AND of two signals: `ring_enable` (an input used to trigger ringing), and `ring_cadence`, an internally generated signal that is on for two seconds and off for four. This cadence matches the North American standard ring pattern and is derived from the `enable_1sec` input.

Audio Router

Port Type	Name	Description
input	clock_27mhz	Main system clock
input	reset	System reset signal
input	ext1_source [3:0]	Source select for extension 1
input	Ext2_source [3:0]	Source select for extension 2
output	audio_reset_b	AC'97 reset
output	ac97_sdata_out	Audio data to AC'97
input	ac97_sdata_in	Audio data from AC'97
output	ac97_synch	AC'97 synchronization signal
input	ac97_bit_clock	Clock for the sdata serial lines.

The audio router provides the virtual switching fabric for the phone exchange and a hardware interface to the AC'97 audio codec. It is responsibly for all audio processing and routing. When a phone call takes place, samples are record from one phone and played into the other, and vice-versa. Tones are also generated within this module.

This module contains many submodules will later come together to complete the router.

AC'97 interface

Through the use of two submodules, `ac97` and `ac97_commands`, adapted from Lab 5, audio frames are clocked in from the AC'97 and split into four 20-bit signals: `left_in_data, right_in_data, left_out_data, right_out_data`. These signals carry 20-bit signed audio samples to and from the AC'97 ADCs and DACs. 48Khz sampling is used, which is the

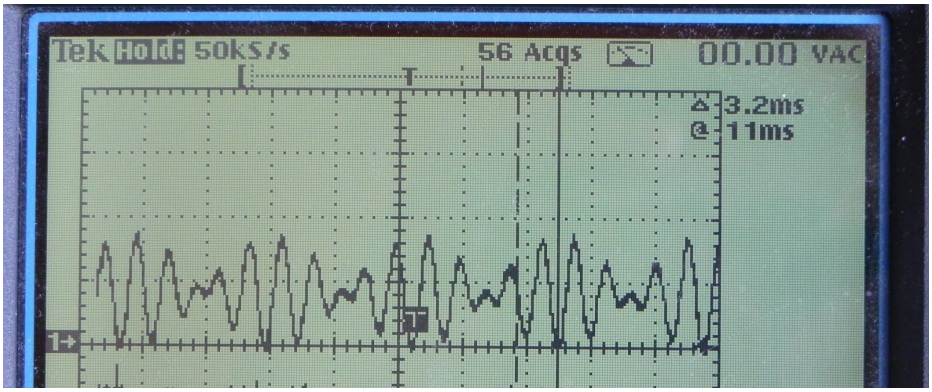
default but overkill for voice telephony. (The real phone network is 8-bit audio at 8KHz sample rate, although this may have improved in recent years)

The AC'97, through the `ac97_commands` is configured to record from the Line In input and output to the Line Out output. These ports are attached to RCA jacks on the labkit, making it easy to attach cables to. This is achieved by setting the master volume register (0x02) to approximately 10dB attenuation, the PCM volume (0x18) to half-full, the record selector register (0x1A) to input 4 (Line In), the recording gain (0x1C) to half its range, and finally cutting out mix1 (register 0x20).

The `ac97` module packs and unpacks the commands and samples into and out of serial frames. A `ready` signal is asserted whenever a new frame is about to be sent or received. This signal is used to coordinate much of the remaining audio router logic.

Tone Generator

In a real phone network, users hear several call progress tones throughout the call. These include dial tone when first lifting the phone, a busy tone if the called party is busy, and a ringing tone if the distant phone is ringing. My system replicates these tones.



Tone	**Frequency**	**Cadence**
Dial Tone	350Hz + 440Hz	Constant
Busy Tone	480Hz + 620Hz	500ms on, 500ms off
Ringing Tone	440Hz + 480Hz	2 sec on, 4 sec off

To generate these tones, a Python script was written to create the samples and save them to a COE file. This file was then used to initialize a block memory module in ISE.

```
import math

SAMPLE_RATE = 8000.0
FREQS = ((480.0, 620.0), (350.0, 440.0), (440.0, 480.0))
NUM_SAMPLES = 800

def twocomp(x):
        return (int(x) + 256) % 256

with open("tones.coe","w") as f:
    f.write("; 800 samples of busy tone, dial tone, and lastly, ringing tone\n")
    f.write("memory_initialization_radix=10;\n")
    f.write("memory_initialization_vector=\n")
    for waveform in FREQS:
        for x in range(NUM_SAMPLES):
            f.write("{0},\n".format(
                    twocomp(
                        64*math.cos(2.0*math.pi*waveform[0]*x/SAMPLE_RATE) + \
                        64*math.cos(2.0*math.pi*waveform[1]*x/SAMPLE_RATE)
                    )
                )
            )
    f.seek(-2, 1)
    f.write(";")
```

The resulting block memory module contains 2,400 8-bit samples (800 for each type of tone). The tone generator module only operates at 8KHz (which is more than adequate), but this allows the module to neatly pipeline its memory access. Accessing the memory requires two steps that must occur on different clock cycles: (1) set the requested address, and then (2) read the data. Whenever the ready signal is asserted 48,000 times per second, one step is executed at a time, in a loop

1. Request next busy tone sample, go to (2)
2. Output busy tone sample, go to (3)
3. Request next dial tone sample, go to (4)
4. Output dial tone sample, go to (5)
5. Request next ringing tone sample, go to (6)
6. Output ringing tone sample, and restart at (1).

This works nicely because three tones multiplied by two steps each is 6 clock cycles, which means 8000 samples are delivered per sample per waveform when clocked with the 48KHz `ready` signal from the `ac97` module.

The waveforms are sent back to the main audio router module. However, before that happens, a timer is used to apply the correct cadences to the streams. This timer runs by prescaling the 48KHz `ready` clock.

Some difficulties encountered while developing the tone generator include using an incorrect sample format. Originally, the Python script generated unsigned 8-bit samples that were concatenated with 12 zeros to produce 20-bit samples for the AC'97. However, this produced oddly inverted waveforms on the output. The samples needed to be converted into two's complement signed integers. The script was reworked to add 255 to each sample and taking the modulus 256.

Routing

Back in the top of the audio_router module, the AC'97 input and output audio streams and the tone streams are brought together for switching. A series of case statements selects the appropriate sources for each output (where each output is a user's phone's speaker) based on source select inputs `ext1_source` and `ext2_source`. This codes runs once every time `ready` is asserted.

```
case(ext1_source)
    default:  left_out_data <= 0;
    1:  left_out_data <= right_in_data;
    2:  left_out_data <= left_in_data;
    11: left_out_data <= {dial_tone_stream, 12'b000000000000};
    12: left_out_data <= {busy_tone_stream, 12'b000000000000};
    13: left_out_data <= {ringing_tone_stream, 12'b000000000000};
endcase
case(ext2_source)
    default:  right_out_data <= 0;
    1:  right_out_data <= right_in_data;
    2:  right_out_data <= left_in_data;
    11: right_out_data <= {dial_tone_stream, 12'b000000000000};
    12: right_out_data <= {busy_tone_stream, 12'b000000000000};
    13: right_out_data <= {ringing_tone_stream, 12'b000000000000};
endcase
```

Adding More Phones

Originally, this project called for having three phones. The AC'97 contains only two ADCs and two DACs (one for left and another for right). A possible workaround for this involved attempting to multiplex more than two input/output streams by rapidly switching the recording mux and active output between different samples (and thereby sacrificing overall sample rate).

Assuming the record select mux can keep up, this method should work fine for sampling three streams at 24KHz each (we can do the first two at once because of the stereo ADC). Outputting three concurrent streams would follow a similar theory. The samples would be lined-up round-robin and a different output's mute function would be undone when that output's sample arrived at the DACs (like a chip-select signal but analog). However, this does not yield three concurrent half-rate streams. Instead, it gives three half rate concurrent streams with zeroes every

other sample. It could be possible to low-pass filter the output or implement a sample-and-hold circuit to try and fill in the zero samples, but this was all deemed too complex to be worthwhile. Should the system be expanded, a true multi-channel audio solution should be used.

PBX Core

Port Type	Name	Description
input	`clock_27mhz`	Main system clock
input	`reset`	System reset signal
input	`Ext_dial [7:0]`	Concatenated dial_digit from each line interface
input	`Ext_dial_done [1:0]`	Concatenated dial_digit_ready from each LI
input	`Ext_active [1:0]`	Concatenated phone_active signal from each LI
output	`Ext_ring [1:0]`	Concatenated ring_enable signal from each LI
output	`State_out [7:0]`	Concatenated state output from each line's internal FSM

The PBX core (Private Branch Exchange, a technical name for a small private phone system like this) handles interactions between phones.

The module was originally designed to be high parameterizable but this was cut due to time constraints. However, some aspects of this design remained. One example is the aggregated signal ports to accommodate multiple connected line interfaces. For example, ext_dial[7:4] contains Extension 2's last dialed digit (from its line interface). This permits parameterized access to any given phone. For example, Extension *n* can be rung with `ext_ring[n-1] <= 1;` Ideally, the module itself would be parameterized by the number of lines and scale automatically. For example, inputs would be defined as `output reg [N:0] ext_ring` to accomodate N ringers. Right now, the bulk of the module is implemented with two repetitive and complementary state machines. Below is a description of one such state machine for phone *n*.

1. **STATE_IDLE**: In this state, the phone is doing nothing. This is the only state in which the phone can receive calls from others. Otherwise, it waits here until the phone becomes active (`ext_active[n]` goes high) and transitions to STATE_WAIT_DIGIT.
2. **STATE_WAIT_DIGIT:** At this point the phone has been picked up and the user is preparing to dial. The FSM commands the audio router to begin playing a dial tone into this phone though the appropriate source selector outputs

(`ext_audio_control[4*n+3: 4*n] <= AUDIO_ROUTER_DIALTONE;`) Once the dialed digit is registered (dial_digit_ready), the number is evaluated. If the number is a valid extension and that extension is in `STATE_IDLE`, the call can proceed and gets set up. This phone is placed in `STATE_DISTANT_RING`, and the called phone is placed in `STATE_DISTANT_CALL`. If the number is bad or busy, the phone is sent to `STATE_BAD_NUMBER`.

3. **STATE_BAD_NUMBER**: In this state, a busy tone is played until the user hangs up (which sends the phone back to `STATE_IDLE`).
4. **STATE_DISTANT_RING**: This state plays a ringing tone to the calling phone to indicate that the distant, called party is ringing. This state continues until the called phone answers. At that point, the phone transitions to `STATE_IN_CALL`.
5. **STATE_DISTANT_CALL**: This state is for phones receiving a call from afar. The PBX activates the ringer for that phone to signal the user to pick up. Once the user does, this phone also transitions to `STATE_IN_CALL`.
6. **STATE_IN_CALL:** Once both phones in the call are in this state, the actual call connection can begin. The ringing is stopped and the other phone's audio is played into the current phone. (Since each FSM does this, the call gets connected both ways, so there is a bidirectional link). At this point, either phone getting hung up will terminate the entire call by sending both phones into the STATE_TEARDOWN state.
7. **STATE_TEARDOWN:** Teardown refers to the steps taken to dismantle a phone connection. In the real phone network, this is quite a process in its own right if one imagines the large number of relays that once needed to be reset across the country for a long-distance call. However, in this system, it is as simple as instructing the audio router to send silence to both phones. There is also a requirement that both phones be hung up before transitioning back to STATE_IDLE. This ensures dial tone isn't sent immediately after the call ends.

Finally, there is a state output for debugging and connection to the LED displays and VGA screen. This is a concatenation of all FSMs' states. In this case, when `state_out` equals `0x55`, both phones are in `STATE_IN_CALL` and are shown connected on the VGA screen.

Overall, the PBX module is good at achieving its goal for a two-phone system, but very cumbersome should it need to be expanded. It was my initial goal to have a loop and array of states for each phone as opposed to a new hard-coded state machine for each one. However, I ran into issues getting the `for` loop to operate properly. This would certainly be the first area to improve given more time.

VGA Screen

Port Type	Name	Description
input	`vclock`	65Mhz clock for VGA
output	`Vsync, hsync, blank`	VGA timing signal
output	`pixel`	A one-bit monochromatic pixel output
input	`Ext1_active, ext2_active`	Phone active signals from each Line Interface
input	`Ext1_dial_digit [3:0], ext2_dial_digit [3:0]`	Dialed digit from each LI
input	`connection`	High when both phones are in a call.

The VGA module generates an XVGA video signal displaying a character text. The screen displays status information for each phone and the presence of a connection. The module was created by combining and customizing the Lab 3 pong video code and a sample VGA character generator handout. The sample code had to be modified to accommodate the larger screen and different clock frequency. It was also modified to not use a character text buffer, and instead generate the text dynamically given a row and column. A series of case statements determines what character should appear given a screen coordinate. This is suitable since the text displayed is very limited and static (only a few symbols and numbers change).

```
Phone Exchange

EXT 1  ↳   2  >2
EXT 2  ↳   0  >1
```

The above graphic shows a typical screen image. The phone icon indicates the phone is active, the next number indicates what was last dialed (Ext 1 called 2, Ext 2 hasn't dialed). The arrow indicates what other phone this extension is connected to.

Character ROM

Generating characters requires a raster font stored in memory that can be referenced when drawing the VGA output. 6.111 provides such a font as a COE file. I modified it to include two telephone icons, depicting a phone on-hook or off-hook. This was then loaded into a new block memory device and incorporated into the `vga` module.

Some problems were had along the way where no characters were outputting on to the screen. It turned out that I had made a typo and set a non-existent wire as the character ROM's clock. Thus, no look-ups were performed. Changing it to `vclock` fixed the issue. The memory access may not be properly pipelines, however, it looks fine visually.

Miscellaneous

A couple of other modules are used throughout for utility purposes. One such module is debouncer. The hookswitch signals are aggressively debounced to prevent noise and glitches in the phone line from accidentally taking a phone off-hook or registering a falsely dialed digit.

A clock divider is also present to provide longer-period signals. The line interfaces use this divider to generate a pulse every second to generate the 2s on, 4s off ring cadence.

Conclusions

Future Features

One feature that would have been excellent to include, had time permitted, is a voicemail system. Such a system would allow users to leave recorded messages in case the called party does not answer. Users can later check their voicemail inboxes by calling an unused extension number like "0". Messages could be saved in ZBT memory due to its large size. Had I had an extra day, I could have implemented this.

Another improvement discussed was including DTMF (touch-tone) dialing capabilities. This would involve some more advanced audio processing to decode digits from their corresponding waveforms. Alternatively, hardware DTMF decoders are available, like the Mittel MT8870 IC. This device outputs a 4-bit value corresponding to the dialed digit.

Overall Thoughts

Overall, I am pleased with the outcome of the project. It functions as a nice phone exchange and satisfied a long-standing desire to build such a system. I was able to overcome all the technical challenges along the way, but fell short on time to implement the entire set of feature I had wanted to see.

If it wasn't dependent on the labkit, I'd like to set it up between my dorm room and a friend's. It is rewarding to see a project come to life and see people amused by it. However, the large amount of analog design inherent to this project makes me see why VoIP solutions have become so popular.