# Tones from Keyboard and Temperature Sensor

Using PS/2 and I2C with a Mono Audio Output and a Buzzer

Arsha Ali, Zhenye Li, Stefanie Kozera
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
e-mails: arshaali@oakland.edu, zhenyeli@oakland.edu, stefaniekozera@oakland.edu

***Abstract*—the purpose of this project is to communicate and interface with sensors that will result in tones being generated on the mono audio output and through a buzzer. The hardware for this project consists primarily of a USB standard keyboard, the temperature sensor, a buzzer, and the mono audio output present on the Nexys 4 DDR board. With the use of a switch, the user will be able to choose between generating tones using the keyboard or using the temperature sensor. Another switch is used to choose between the buzzer and the mono audio output. While using the keyboard, a unique tone will be heard for fifteen different keys, with a sixteenth variation of no tone. A qualifying key that generates a tone will appear across all four seven-segment displays, with no tone and nothing appearing across the seven-segment display for all other keys. For the temperature sensor, one of sixteen tones will be generated based off the current temperature reading. This can be heard through the buzzer or the mono audio output. The major finding is that a USB keyboard uses PS/2 for communication and the temperature sensor uses I2C for communication. These communication protocols and manufacturer specifications require carefully implemented finite state machines. The conclusions clearly indicate that different data communication protocols are powerful tools to interface with sensors and to drive outputs. Recommendations to the user include thorough testing of both the keyboard and the temperature sensor.**

## I. INTRODUCTION

This report will outline the methodology, experimental setup, results, and conclusions from undertaking this project.

The motivation to generate tones and display figures on the seven segment display stems from the desire to showcase both a visual and auditory side of hardware. With the pressed keyboard key or the temperature being displayed on the seven segment display, the audience can clearly see the outcome and proper functioning of the hardware components. In addition, the audience can hear an output from either the mono audio output or the buzzer. For the keyboard side, a noise is only heard while a key is pressed. For the temperature sensor side, a tone is continuously heard that relates to the current temperature that is detected.

Topics discussed in class that are present in this project include the functionalities of several hardware components and their implementation in VHDL, such as finite state machines, decoders, and multiplexers, among others. Also, this project draws from the concept of interfacing with external peripherals using the PS/2 and I2C communication protocols.

Although discussed briefly in class, implementation of the circuit that interfaces with the keyboard had to be thoroughly learned. Also, the interaction with a temperature sensor was studied for a more complete understanding. Furthermore, the generation of pulse width modulation (PWM) and pulse density modulation (PDM) signals had to be thoroughly understood.

The applications of this project include using a keyboard to generate tones, which is a form of a personal piano. Tones generated from the temperature sensor serve as an auditory aid for an estimate temperature. These tones can be heard either through the mono audio output or a buzzer based on the user's preference.

## II. METHODOLOGY

Figure 1 documents a high level architectural structure of the system. Figure 2 shows the details of the keyboard side of the system. Figure 3 shows the details of the temperature sensor side of the system. Interfacing with the keyboard and the temperature sensor was understood through class notes [1]. The components that interfaced with the peripherals were downloaded and modified/implemented from the course website [2-4]. The datapath also consisted of other basic components [5]. The components are explained in further detail below.

### *A. my_ps2read*

This block interfaces with a standard keyboard using PS/2. The PIC24FJ128 chip inside the Nexys 4 DDR emulates a PS/2 protocol for the FPGA. The inputs to this block include the PS/2 clock and data, as well as the FPGA clock and a resetn signal. On falling edges, the data is captured. The format frame of the data is a start bit, which is '0', 8 bits of data with the LSB transmitted first, an odd parity bit, and a stop bit, which is '1'. For the data, there is a setup and hold time that must be complied with. The signal ps2c, for the PS/2 clock, goes through a filter and a falling edge detector in an FSM issues a '1' every time it detects a falling edge on the clock signal. If there is a falling edge and the start bit is '1' on the data line, then this represents the

start bit. A counter from 0 to 9 is used to shift in the next 10 bits of data using a right shift register.

The filter makes sure that ps2c is constant for 8 clock cycles before the output signal is changed. This is to reduce the chance for glitches that could be misinterpreted as falling edges. This circuit uses a right shift register to shift in ps2c, an 'and' gate, a 'nor' gate, a decoder, and register to output the filtered version of the clock.

The output of this block is the data out as 10 bits. This consists of the stop bit, the parity bit, and the scan code of the pressed key. The data out is configured in this format due to the way that the data is shifted in. When all 10 bits following the start bit have been shifted in, a done signal is set to '1' to indicate that the output data is valid.

The scan code is transmitted to the block every 100 ms. Once the key is no longer pressed, a keyup scan code will be outputted, followed by the scan code of the previously pressed key.

### *B. my_ps2keyboard*

This block consists of *my_ps2read*, registers, and an FSM that checks for when a pressed key has been released. Figure 4 details the flow of this FSM. The FSM checks the done signal from *my_ps2read* to see if that lower 8 bits are valid. After this, the FSM will check to see if the 8 bits represent the key up scan code, which is typically 0xF0. If not, this means that a key is being pressed, so an enable signal is sent to the register to capture these lower 8 bits as the scan code. This enable is also the data input of the done register which is always enabled, so a done signal is issued indicating that the scan code is valid. If the key up code is issued, the signal keyup is set to '1'. This signal keyup is only '1' when the key up scan code is received. Since the protocol follows the key up code by repeating the scan code of the pressed key, state two is used to check when this scan code has been received such that it can be ignored by the output signals.

### *C. fsm_keyup*

Figure 5 details the flow of this FSM. This FSM issues a high signal once the first scan code is valid by checking the done signal of *my_ps2keyboard*. This signal, called dEn, stays high until the keyup scan code is received, at which point the signal goes back to '0'. Effectively, this signal is '1' while a key is pressed. The purpose of this finite state machine is to serve as an enable for the *keyboard_decoder* and the *frq_decoder*.

### *D. keyboard_decoder*

This decoder uses the scan code from *my_ps2keyboard* to output seven bits that will be given as the input to the *serializer*. The output represents which leds should be illuminated on the seven segment display. This decoder is only enabled while a key is pressed. This is done by using dEn as the enable. Thus, when no key is pressed or an invalid key is pressed, the decoder outputs all 0's.

### *E. frq_decoder*

This decoder uses the scan code to output four bits and an SD signal that will be given as the input to *my_audio*. This decoder is only enabled while a key is pressed. For qualifying keys that are pressed, a unique four bits are outputted with an SD of '1'. These four bits will eventually control the variation rate for a sinusoidal output from the mono audio output.

### *F. serializer*

The serializer inputs map to each of the eight seven segment displays. For the serializer, the same output from the *keyboard_decoder* is mapped to each input, such that the same configuration will appear across all eight of the seven segment displays. This input is inverted before going to the leds of the seven segment display. The serializer component consists of its own FSM, multiplexor, and counter. These components serve to enable each of the seven segment displays for 1ms. At this speed, it appears as if all of the eight seven segment displays are illuminated at the same time.

### *G. my_audio*

To get a PDM signal with changing duty cycle, this block is used. This circuit uses *my_pwm*, counters, and a built-in LUT to generate a PDM signal. This PDM output will eventually be connected to the mono audio output. This block takes four bits as the input frequency, which controls the variation rate of the changing duty cycle. These four bits translate into a maximum counter value. Sample code is shown in Figure 6. Each time that this maximum count is reached, the output of an 8 bit counter increases by 1 between 0 to 255. The frequency for the output signal is set to 10kHz. Thus, the four inputs bits control how quickly the duty cycle changes. The output of the 8 bit counter is scaled, and fed as the duty cycle input to *my_pwm.* The output of this block is therefore a PDM signal.

### *H. my_pwm*

There are 15 *my_pwm* components that are pre-initialized with a 50% duty cycle and a frequency. Some sample code is shown in Figure 7. Each of these 15 blocks will produce a different tone on the buzzer, since different tones on the buzzer are heard by different frequencies. The duty cycle controls the volume. TPWM is the period of the PWM signal in units of the FPGA clock cycles. Using Equation 1, 15 frequencies ranging between 20Hz to 12kHz are generated.

**Equation 1:**

$$\boldsymbol{Tpwm = TPWM/fclock}$$

This component consists of an FSM, a register, and an embedded counter. Essentially, the output PWM signal is toggled every DC clock cycles and TPWM is used to check if one period is complete before the sequence starts again.

The output of each of these blocks is one bit, part of the output PWM wave. Each of these are the input for the *16-to-1 Multiplexor*.

### *I. 16-to-1 multiplexor*

The inputs to this multiplexor are the one bit PWM output signals from *my_pwm.* The last input to this multiplexor is a constant '0' bit, which will produce no tone on the buzzer. The selector for this multiplexor is the scan code that is outputted from *my_ps2keyboard.* Thus, pressing a qualifying key generates a unique tone that is sent to the buzzer.

### *J. 2-to-1 multiplexors*

These *2-to-1 multiplexors* are used to produce a sound on either the mono audio output or the buzzer, but not both at the same time. The selector for this multiplexor is a switch on the Nexys board. There is a multiplexor for AUD_SD, which will either enable or disable the mono audio output. There is also a multiplexor for the output to the buzzer. By correctly mapping the multiplexor inputs with '0', the switch is used to control whether the mono audio output or the buzzer will be used to hear the tone.

### *K. tempsensor_i2c*

This block handles interfacing with the temperature sensor and outputting 16 bits of data pertaining to the current temperature. This block consists of an *fsm_tempSensor* that configures the temperature sensor for 16-bit mode and outputs either the low and high byte of the temperature or the status and ID. *wr_reg_adt7420* handles I2C communication with the temperature sensor. There are also two 8-bit registers for the high and low byte of the temperature before these signals are outputted.

### *L. wr_reg_adt7420*

This module controls I2C communication with the ADT7420 temperature sensor on board the Nexys board using the signals SCL and SDA. A start bit, a read/write bit, and two eight bit buses relating to an address and data are provided as inputs, and eight bits of data are outputted. In addition, there is an output done bit and an output err bit. The done bit is issued when the operation of either writing or reading data is completed. This circuit is meant to convert the timing protocols native to the ADT7420 temperature sensor into an I2C signal. For this protocol, the start condition is when there is a high-to-low transition on SDA while SCL is high. This will start the data transaction. The stop condition is when there is a low-to-high transition on SDA while SCL is high. For reading data from the temperature sensor, the master must write a 7-bit address for the ADT7420 followed by a write bit. The acknowledgement will come from the slave. The internal register address is provided next, followed by another acknowledgement from the slave. A repeated start condition is next, followed again by the device address, and a read bit. The acknowledgement comes from the slave and then the 8-bit data comes from the slave. The master issues a not acknowledge on this data, and then the stop condition is asserted. This is accomplished through the use of three state machines, left shift registers, counters, basic logic gates, and buffers.

The *fsm_scl* generates a clock signal, rising and falling edge detectors, and a delayed falling edge signal. The delayed falling edge signal is to allow data to be kept for a specific hold time as outlined by the manufacturer. The clock signal is connected to an active-low tri state buffer before going to SCL. The period of the clock signal is determined by the generic input SCL_T. Data is to be placed on this delayed falling edge and captured on rising edges.

The *fsm_ack* detects the acknowledge bit which is sent by the I2C slave. If the acknowledgement does not arrive, an error signal on err is issued. If the acknowledgement bit has arrived from the slave, a done signal is set to high and used by *fsm_main* to know when the next write/read cycle can begin.

The *fsm_main* complies with the I2C protocol by issuing various signals to the other components in this circuit. It is important to note that data is transferred with the MSB first so left shift registers are used.

### *M. fsm_tempSensor*

This FSM configures the temperature sensor for 16-bit mode by writing 0x03 on the 0x80 internal register. This FSM also issues the addresses for either reading the high and low byte of the temperature or reading the status and ID. Signals for the I2C protocol are also issued and passed to *wr_reg_adt7420.*

### *N. temp_decoder*

Four *temp_decoder* blocks are used as part of the datapath. Each decoder takes 4 bits as the input, which are the bits relating to the high and low temperature reading (odata_h and odata_l) after being split from 8 bits to 4 bits. The output of these blocks are 7 bits that correspond to which leds should be illuminated on the seven segment display to represent those 4 input bits in hexadecimal representation. The outputs are passed to a *serializer* block before being sent to the seven segment display.

### *O. Mono Audio Output and Buzzer for Temperature Sensor*

The bits 7 downto 4 of odata_l are used to control the tones that are generated from the mono audio output and the buzzer. These bits are passed to *my_audio* and are used as the selector for the *my_pwm* outputs.

## III. EXPERIMENTAL SETUP

The setup that was used in order to verify the functionality of the project was first to plan out how each signal would be produced. From there, each new component would be constructed and implemented.

A keyboard file was first obtained from the course website. This file was studied and modified for the intent of

this project. A serializer was then connected, followed by the mono audio output components, and then the buzzer components. Each step of the process was simulated and verified for accuracy. The same steps were followed for the temperature sensor. The preliminary file was obtained from the course website. Next, the serializer was built, followed by the audio outputs.

The software Vivado 2018.3 was used during the creation of the program. The software allows the use of timing and functional simulations to verify the intended functioning of the project. The keyboard simulation was analyzed by creating a test bench of the overall keyboard file. The test bench consisted of two sample scan codes that may be produced by the keyboard. The output parameters were determined from the simulation. When the code initially did not produce the expected results, the functional simulation was used to trace signals and correctly identify the source of error. This allowed for deeper understanding of the hardware components and led to the creation of the *fsm_keyup*. Without this finite state machine, a tone would be generated and heard even after the key had been released. This allows for a tone to only be generated when one of the valid 15 keys are pressed. All other keys, as well as no key pressed, do not produce a tone.

A testbench for the temperature sensor side of the project was also used to create a simulation. Instead of creating the I2C signals, the test bench began from providing data for odata_h and odata_l. From these simulations, desired results were verified.

Although the simulations were showing the desired results, when the entire circuitry was tested on the Nexys board, there were issues in the mono audio output. All other aspects worked as expected. The code responded to inputs from the switches, placed information across the seven segment display, and generated a tone on the buzzer. It was later learned that the tri state buffers that were originally inside the *my_audio* component had to be removed. This was because the tri state buffers sit at the I/O peripherals. Instead, the output from these components for both the keyboard and the temperature sensor side were fed through a multiplexor, and this output was then the enable of a tri state buffer.

The expected results are to generate tones only when a qualifying key is pressed on the keyboard or to generate tones continuously using the temperature sensor. A switch should choose between a tone heard through the buzzer or the mono audio output. Another switch should choose between the keyboard or the temperature sensor, and a third switch will choose between the temperature of the temperature sensor or the status and ID of the temperature sensor. The results are discussed in further detail in the following section.

## IV. RESULTS

Figure 8 shows the simulation of the keyboard component and Figure 9 shows the simulation of the temperature sensor component.

For the keyboard testbench, two scan codes were simulated. The first scan code was 0x1C which is for the letter “a” followed by the 0xF0 for the keyup scan code. The switches input was “100”, meaning to hear the output from the mono audio output from the keyboard. While the scan code is 0x1C, a valid key is being pressed. Thus, the signal dEn is ‘1’ and is enabling the *keyboard_decoder* and the *frq_decoder*. The output signal AUD_SD follows dEn exactly. The internal signal for AUD_PWM is a PDM signal changing between high and low states. The actual output is first passed through an active-low tri state buffer so the actual output changes between ‘0’ and ‘Z’ for high impedance as desired. The internal signal for the buzzer is a PWM signal. However, due to the input switches, the actual output on the signal buzzer is ‘0’ while the valid scan code is present. Once the keyup scan code is detected, this means that the key is no longer pressed. At this point, the dEn and AUD_SD signals go to ‘0’. The internal buzzer signal (signal b) also goes to ‘0’. The signal AUD_PWM is still changing, but since AUD_SD is ‘0’, this signal is not heard.

For the temperature sensor testbench, instead of providing the I2C signals, one temperature was simulated. The arbitrary temperature was 0x12F6. The 4 bits that determine the tone of the output are “1111”. The switches input was “011”, meaning to hear the output from the buzzer from the temperature sensor. In practice, sw(2) should have been set to ‘1’, but it does not matter here for simulation purposes as the incoming data is provided from the testbench. The internal signal AUD_PWM is changing between high and low states. The actual output AUD_PWM is changing between low and high impedance states. However, since the switches are set to hear from the buzzer, AUD_SD is low and the PDM signal will not be heard. The simulation shows how the internal PWM signal (signal b) for the buzzer is the same as the actual output from the buzzer. From these simulations, it is also clearly seen that the output for the buzzer is a PWM signal while the output for the mono audio output is a PDM signal.

The results were achieved by correctly integrating various hardware components and external peripherals as learned in class. The keyboard and temperature sensor are able to accurately generate tones from the buzzer and the mono audio output. A switch is used to toggle between these two modes. When a key is pressed on the keyboard, for only that time the appropriate figure appears across the seven segment displays and the tone is heard. After releasing the key, the seven segment display is empty and no tone is heard. Tones are continuously generated by the temperature sensor. As the temperature increased or decreased, the tones changed as desired. The tones are accurately heard either through the mono audio output or the buzzer, as indicated by another switch. A third switch

chooses between either the temperature or the status and ID of the temperature sensor. The results were as expected, and all results were accounted for and explainable.

## CONCLUSIONS

The main point that has been learned while doing this project is that communication protocols are powerful and useful methods to interface with external peripherals. It was also learned that such protocols can be effectively implemented in hardware, and the implementation should abide by the manufacturer's specifications. After thorough testing for results, no issues remain to be resolved. The intended outcomes for this project were achieved. Further work could include the addition of generating tones using one direction of the on-board accelerometer. In addition, the RGB leds could be used to change colors based on the current tone that is being generated. There is no limit to other peripherals that could be added to this existing project. The number of different tones could also be expanded to include more keys from the keyboard and more bits from the temperature reading.

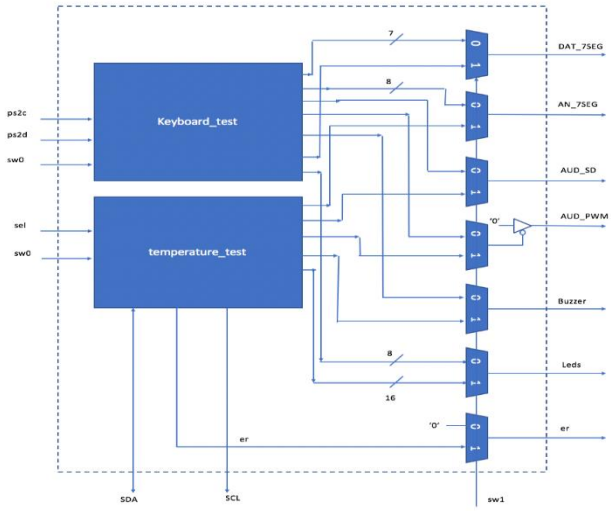


Figure 1: High-level architecture of the entire system

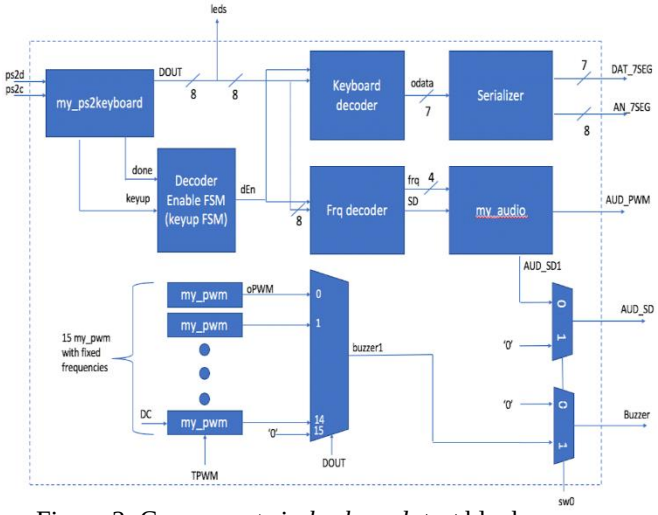


Figure 2: Components in *keyboard_test* block.

## REFERENCES


[1] D. Llamocca, "Unit 3-External Peripherals: Interfacing," Mar. 2019, pp. 2-3, 7-14., https://moodle.oakland.edu/pluginfile.php/5005747/mod_resource/content/7/Notes%20-%20Unit%203.pdf

[2] D. Llamocca, "PS/2 Keyboard Controller (XDC included)," http://www.secs.oakland.edu/~llamocca/VHDLforFPGAs.html

[3] D. Llamocca, "ADT4720 Temp. Sensor (I2C)-Basic Control (XDC included)," http://www.secs.oakland.edu/~llamocca/VHDLforFPGAs.html

[4] D. Llamocca, "PWM: Tone control. Mono audio output with Low-Pass Filter (XDC included)," http://www.secs.oakland.edu/~llamocca/VHDLforFPGAs.html

[5] D. Llamocca, "Unit 2: Concurrent Description." *VHDL Coding for FPGAs*, slides 7-10., http://www.secs.oakland.edu/~llamocca/Tutorials/VHDLFPGA/Unit%202.pdf


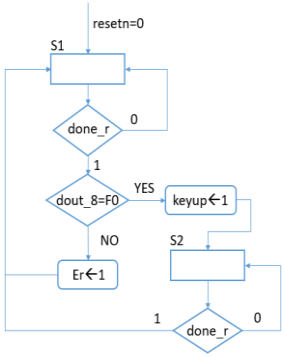


Figure 4: FSM to detect key up scan code

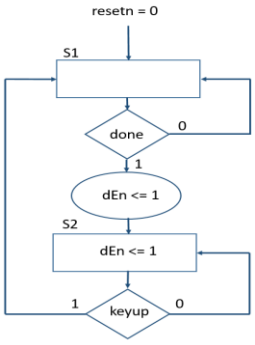


Figure 5: FSM in *fsm_keyup* component

```
with frq select
    TM <= 63 when "0001",
          89 when "0010",
          127 when "0011",
          180 when "0100",
          255 when "0101",
          361 when "0110",
          511 when "0111",
          723 when "1000",
          1023 when "1001",
          1447 when "1010",
          2047 when "1011",
          2895 when "1100",
          4095 when "1101",
          5792 when "1110",
```

Figure 6: Variation rate (max. count) for PDM signal

```
f18: mypwm generic map(TPWM => 12500) --8 kHz
          port map(resetn => resetn, clock=>clock,
          DC => "0110000110101O",oPWM => m); --6250
f19: mypwm generic map(TPWM => 10000) --10 kHz
          port map(resetn => resetn, clock=>clock,
          DC => "0100111000100O",oPWM => n); --5000
f20: mypwm generic map(TPWM => 8333) --12 kHz
          port map(resetn => resetn, clock=>clock,
          DC => "0100000100011l",oPWM => o); --4167
```

Figure 7: PWM signals with different frequencies

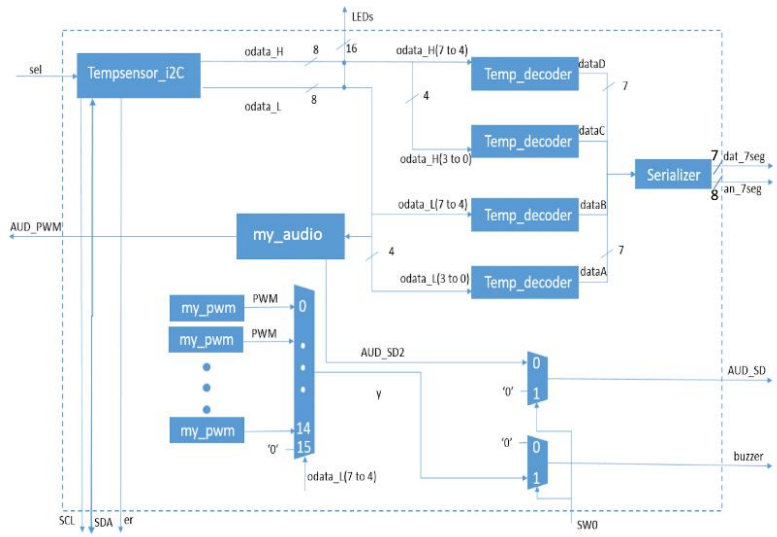


Figure 3: Components in *temperature_test* block.

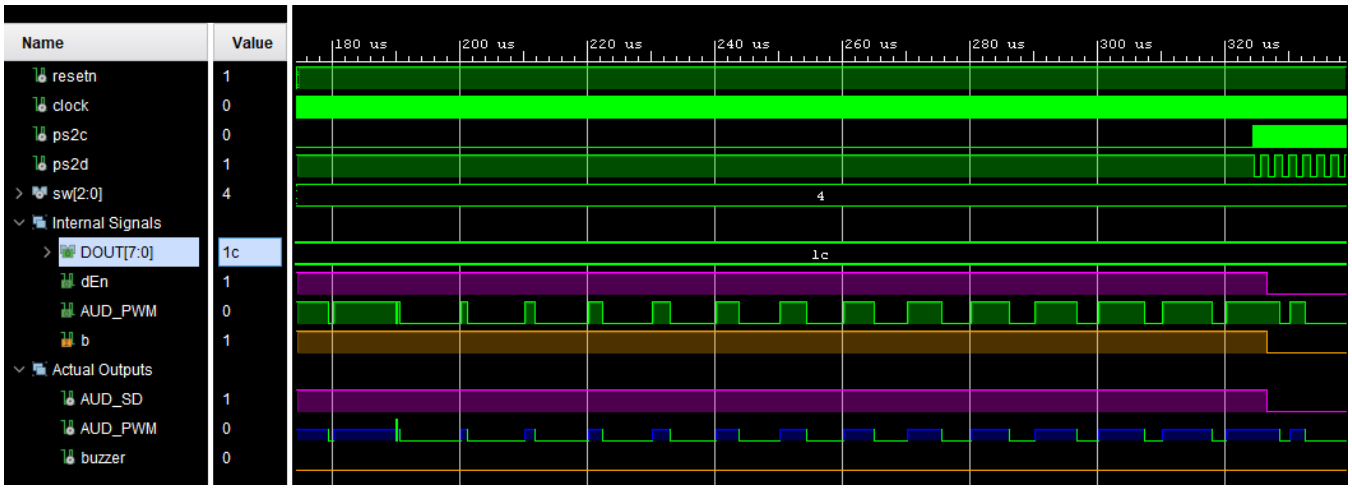


Figure 8: Keyboard component simulation

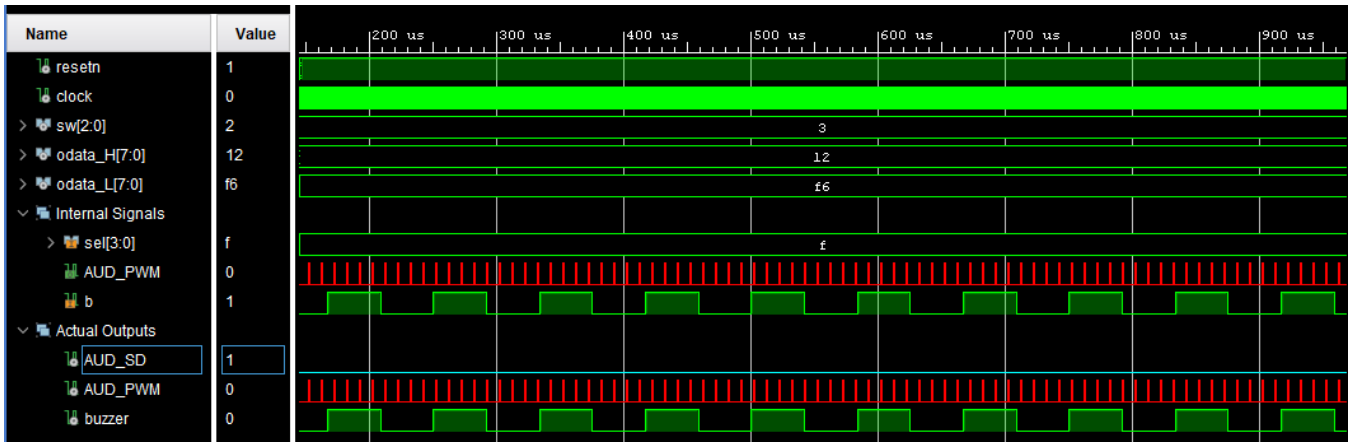


Figure 9: Temperature Sensor component simulation