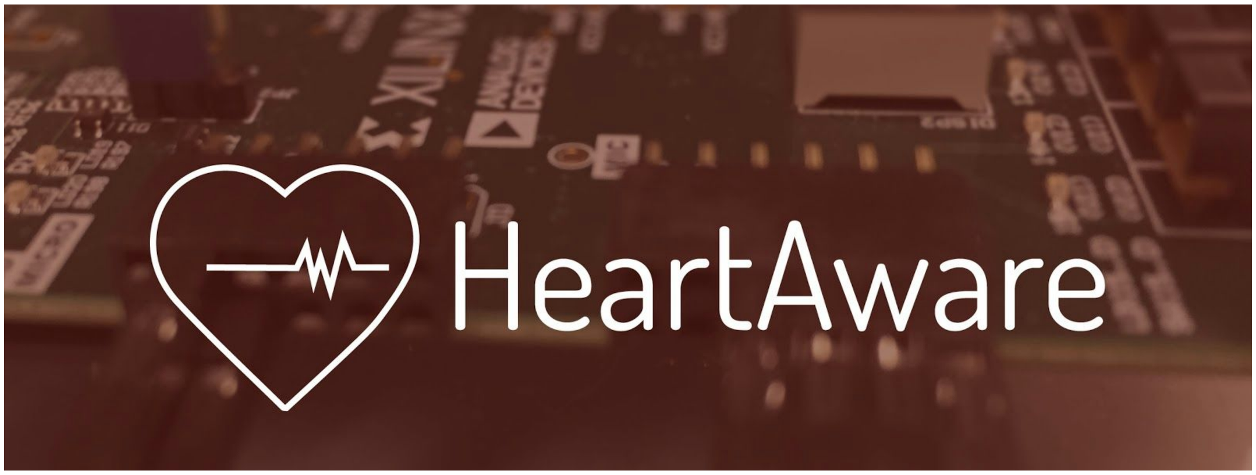


HeartAware
Final Project Report
6.111 Fall 2015

Michael Holachek and Nalini Singh
Submitted December 9, 2015

Project Abstract
Pulse oximetry is a popular non-invasive method for monitoring a person's blood oxygen saturation. The goal of this project was to build a simple pulse oximetry system capable of calculating and displaying a user's real time heart rate using a finger pulse oximeter sensor, analog circuitry, VGA display, speaker, and Artix 7 FPGA on a Nexys 4 DDR board. Using the pushbuttons and switches on the Nexys 4, the user is able to control the system interface and filter parameters.

Acknowledgements
We'd like to thank 6.111 lecturer Gim Hom for his excellent instruction and support during development of our project, as well as Miren Bamforth, our project advisor, for her guidance related to planning and troubleshooting our project. Additionally, the support of other TAs and fellow 6.111 students kept us going through the late nights and occasionally frustrating debug processes. We couldn't have done it without you!

Table of Contents

I. Introduction

This final project report describes the final design and implementation of HeartAware. In addition, it shares our findings related to FPGA development and the process of debugging.

The first section of this report provides a high level system overview. The goal of this project was to build a simple pulse oximetry system capable of calculating and displaying a user's real time heart rate using a finger pulse oximeter sensor, analog circuitry, VGA display, speaker, and Artix 7 FPGA on a Nexys 4 DDR board. We wanted to improve on the user interface of typical medical pulse oximeters by providing a verbal announcement of heart rate in addition to beep sounds. That said, HeartAware is only for information purposes and not for medical use, as we did not build any special redundancy into our features or conduct the reliability tests required for a medical device.

The second section of this report details the design, implementation, and status of each module. We achieved the commitment and goals for all blocks of our project, although time constraints prevented us from implementing our stretch goals. However, ultimately, the key features of our project worked well for many users.

The third section of this report describes our testing and debugging process. Although Verilog modules are hardware-independent, many of the 6.111 Labkit hardware modules were different from the Nexys 4, and the Nexys 4 and Vivado provided some unique challenges that were previously solved for us on the Labkit. Thus, the Nexys 4 had a relatively large "startup" cost and added to the difficulty of this project. This was particularly rewarding, however, since every small feature had to be discovered, written, debugged, and integrated by us.

The fourth section of this report describes several design decisions we had to make during the implementation of this project. Functionality was demonstrated at the final checkoff, although not all features were implemented as modularly or intuitively as originally designed; some workarounds were required due to hardware limitations. In this section we share the complications discovered while building the project, which led to sometimes fundamental changes to our module design.

Finally, our conclusion reviews the goals of the project, its current status, and further direction.

Please note that all code, Vivado project files, and assets related to HeartAware are available at http://github.com/holachek/heartaware for reference.

II. Final System Overview

At a high level, the final system is comprised of five main blocks, as described in the original project proposal, shown in Figure 1. An analog signal from the pulse oximeter is amplified in analog circuitry and converted to a digital signal, which is then input to the signal processing block. The signal processing block detects peaks in the signal and calculates the current heart rate. The processed waveform and calculated heart rate are displayed on a VGA monitor, and the current heart rate is periodically announced over a speaker.

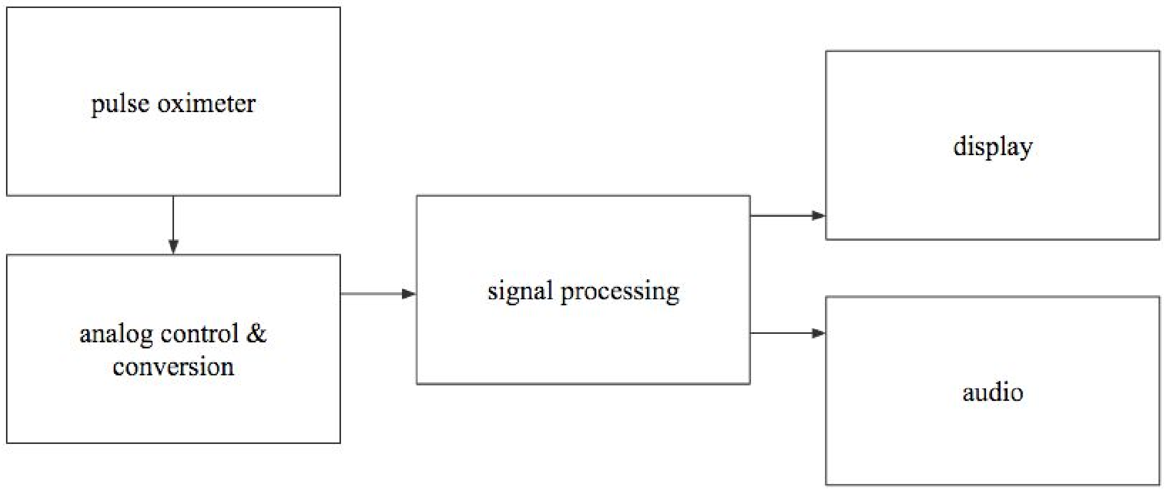


Figure 1: A simplified block diagram of the five system modules.

At the top level, our system has four states, determined by the *system_status* variable. The system starts in a boot state, where it loads audio assets from the SD card.[1] After a few seconds, the system enters the capturing data state, where the user's live heart data is displayed. Should the user desire further analysis of a certain capture of samples, a pause mode can be entered by pressing the left directional button. While originally we planned for the system will transition to an error state if the sensor was disconnected, our breadboarded circuit did not have this automatic functionality. Nevertheless, the user can manually enter the error mode by pressing the up directional button. Pressing the down directional button clears pause/error state and reverts back to capturing data mode. See Figure 2 for a visual summary of the system, as

[1] See section III.E and VI.B for more information about SD card load times and audio buffering.

well as renderings of the final user interface in each mode.

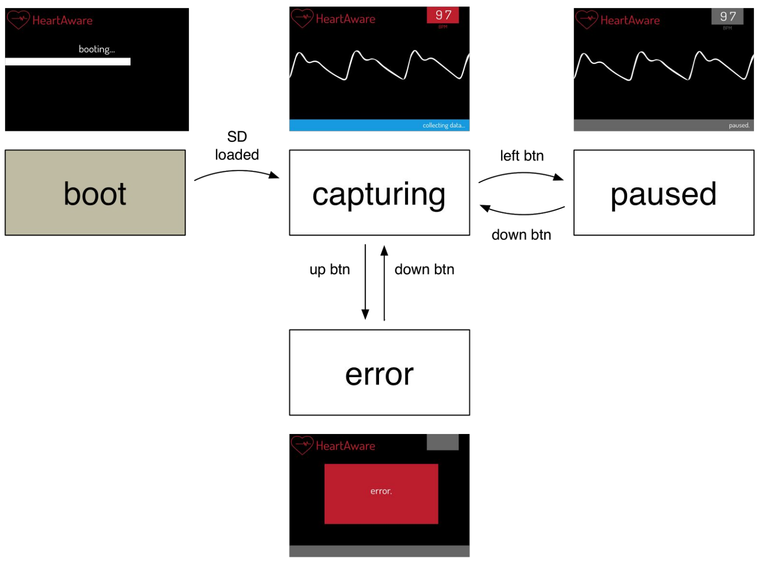


Figure 2. The final FSM implemented in our system.

A note about terminology: *blocks* are defined as complete system features, whereas *modules* describe a single Verilog module. In most cases, for organizational reasons, there is not a one-to-one correspondence between blocks and modules.

III. Block Implementation and Design

We will now describe in detail each block in depth and briefly cover design decisions we took during the development. In depth discussion over design decisions is left to section VI.

A. Pulse Oximeter

The first block is the pulse oximeter. The pulse oximeter sensor is a finger clip consisting of a red and infrared Light Emitting Diode (LED) and a photodiode, as shown in Figure 3. With appropriate driving circuitry, the LEDs will shine light through the user's finger and measure the absorbed light with a photodiode. Because oxygenated hemoglobin absorbs less light than pure hemoglobin, it is possible to convert the absorbed light to a reading of the level of oxygen saturation in the user's bloodstream. For HeartAware, we only used the red LED and not the IR LED, since infrared wavelengths are much less effective than red wavelengths at detecting differences in oxygenated and deoxygenated hemoglobin.[2] The pulse oximeter connects to the system with a D-sub 9 connector. No Verilog modules are associated with this block.

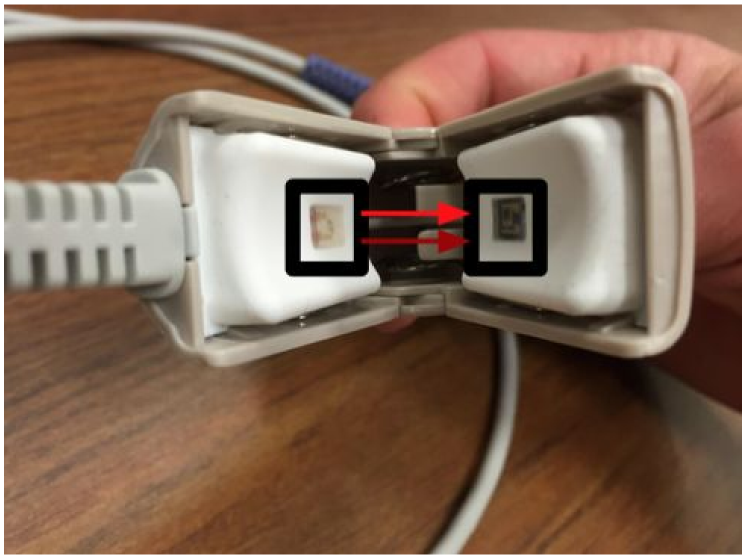


Figure 3. A typical pulse oximeter, with red and infrared LEDs under the left pad, and photodiode under the right pad

[2] http://www.nxp.com/files/32bit/doc/app_note/AN4327.pdf

B. Analog Control & Conversion (Michael)

The second block, the analog control and conversion circuitry, drives the pulse oximeter and converts the analog signal to a digital value for processing on the Nexys 4. The data flow of the input signal is shown in Figure 4.

The first stage of this block, the analog driving circuit, drives the red LED and receives light pulses to construct into a rough oxygen saturation signal through a series of JFET op amps. This design was provided by Gim Hom.

Because the output of this driving circuit was originally meant to be plugged into a microphone input jack, the signal is relatively low in amplitude and does not have sufficient offset for proper digitization by the ADC. Thus, the second stage of this block, the conversion circuitry, processes the initial signal through an op amp to amplify and offset the voltage to an appropriate 0-5V level. To convert the signal from the analog to the digital domain, we used an ADC0804 8-bit parallel output ADC. This integrated circuit converts an input voltage between 0 and 5V into a 8-bit number (thus our signal has 20mV resolution). We then connected the ADC output to a Nexys 4 Pmod port.

To allow for greater modularity and portability, we attempted to integrate all external circuitry on a custom Printed Circuit Board (PCB). While we were able to fabricate the PCB, time constraints made us unable to assemble it, and thus we reverted to the breadboarded circuit for our demo. However, we are confident that given enough time, we could have built and tested the PCB with our system. Our circuit schematics are attached to this report in Appendix A.

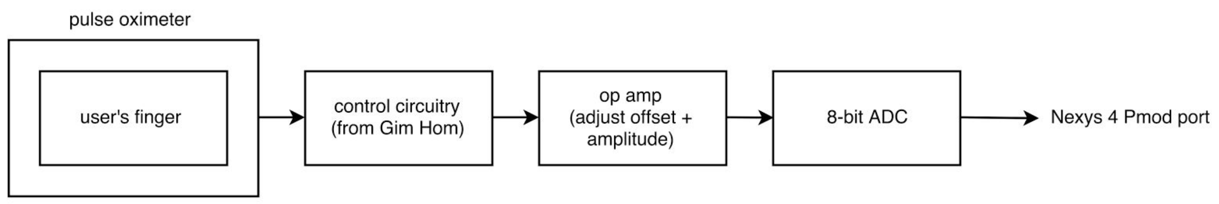

pulse oximeter
user's finger
control circuitry (from Gim Hom)
op amp (adjust offset + amplitude)
8-bit ADC
Nexys 4 Pmod port

Figure 4. Data flow of input signal

C. Signal Processing (Nalini)

The signal processing block converts the digital signal input from the analog control and conversion block to a continuously updating, real time estimate of the current heart rate. At a high level, this heart rate computation is done using three filters and a peak detector.

First, the input signal is lowpass-filtered to reduce noise in the signal. Next, a match filter is used to determine the correlation of the signal with itself; the output of this filter reaches a maximum whenever the signal mostly closely corresponds to a template waveform recorded by the user. A local maxima detector finds peaks in the output of the match filter, and a state machine calculates the number of clock cycles elapsed between successive peaks. This value is converted to a current heart rate value, and a third FIR filter is used to compute the moving average of the calculated heart rates, which is output to the user on the VGA display and over a speaker. A diagram of this block design is shown in Figure 5, and details of the implementation and design decisions for each of the lower-level blocks are described below.

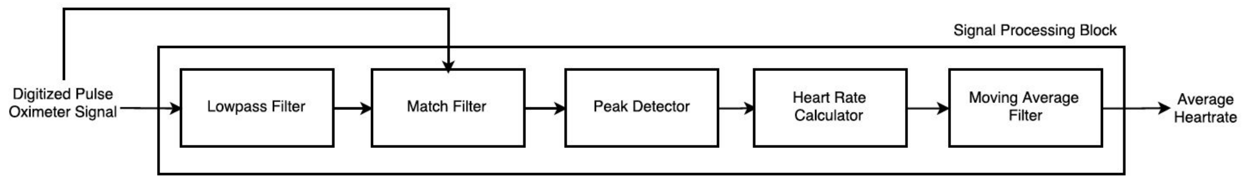


Figure 5. Signal processing filter stages

In the first stage of signal processing, a 100 Hz downsampled version of the input signal is fed through a lowpass filter; the filter implementation and associated coefficients can be found in the Verilog modules fir31_lp and coeffs31_lp_x, and the filter is instantiated within the signal processing section of the main heartaware module. Several sets of filter coefficients were tested for their ability to remove noise from the digitized input signal while maintaining the characteristic pulse oximeter signal waveform; in particular, the number of taps and the cutoff frequencies of the filter were varied and tested in Matlab. Originally, based on the results of these simulations, a simple 5-tap filter lowpass filter was chosen and implemented. However, the input pulse oximeter signal to this module varied significantly in shape and amplitude between different people; the performance of this filter on non-simulated input data was not adequate for the later peak detection stages. Upon testing additional filter designs using these more varied input waveforms, the filter coefficients from Lab 5a proved effective on all of these waveforms, so these coefficients were incorporated into the final design. This lowpass-filtered signal is input to the display block to be rolled across the VGA display.

Next, the output of the lowpass filtered signal was fed to a match filter. A match filter convolves a signal with a time-reversed version of a template in order to determine the correlation between the two; this correlation is highest when the template aligns with a given window of the signal, which occurs at the same period as the heart rate. The coefficients of the match filter are a time-reversed version of the signal itself. Our system allows the user to dynamically update these coefficients to reflect their personal pulse oximeter waveform by turning on switch 13. Upon turning on the switch, the 128 most recent samples from the lowpass filter are stored in FPGA BRAM in reverse-time order. These coefficients are accessed individually at 65 MHz and input to the module fir128_match, where they are multiplied by the corresponding signal sample on the same 65 MHz clock. This allows the entire 128-tap convolution to be easily completed within each 100 Hz clock cycle, when the result of the correlation computation is output. The 128 sample size (corresponding to 1.28 seconds at 100 Hz) was chosen to ensure that at least one entire heartbeat waveform would be captured for standard heart rate values.

Upon obtaining the match filter signal, the remaining steps of the current heart rate calculation are conducted in the hr_calculator module. In particular, the next step of the process involves detecting peaks in the output from the match filter. This is done by storing the most recent 50 sample outputs from the match filter in a buffer. As a result of the lowpass filtering done previously, the output from the match filter is a smooth waveform, typically with clear maxima. Thus, if the middle element in the buffer is larger than the surrounding 49, it is considered a maximum, and the output peak signal is pulsed high for one clock cycle. Testing this design indicated that several peaks were not being detected because the maximum value would be held stable for several clock cycles; as a result, we implemented additional logic to allow maximum values that are held stable for a certain number of clock cycles to be detected as maxima. Ultimately, after making this change, peak detection occurred relatively reliably, as shown by the logic analyzer output in Figure 6 below.

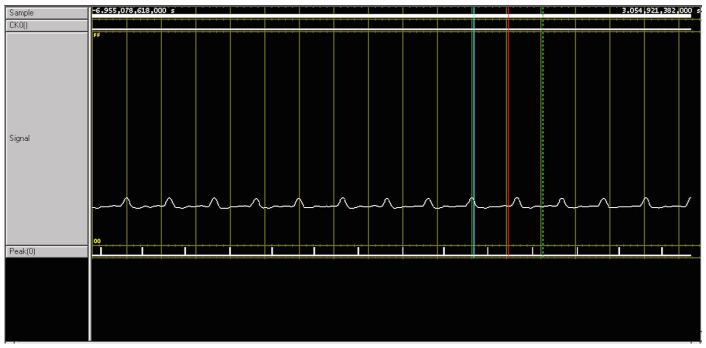


Figure 6. Logic analyzer output showing the lowpass filtered signal and corresponding peak signal calculated using the methods described in this paper. The period of the peak signal corresponds exactly with that of the initial waveform, indicating successful heart rate calculation

Additionally, we realized experimentally that, depending on the shape of an individual's pulse oximeter waveform, the peaks in the original lowpass filtered waveform are sometimes pronounced enough that this peak detection algorithm can be detected without the added complexity of the match filter. In this case, instead of feeding the match filter output as an input to the hr_calculator module, the lowpass filtered signal can be fed in directly.

As the peak signal pulses high with each heartbeat, the hr_calculator concurrently calculates the current heart rate. A counter tracks the number of clock cycles elapsed between peaks, and 6,000, the number of 100 Hz clock cycles in a minute, is divided by the number of elapsed clock cycles to obtain the heartrate.

Finally, the most recent 16 heart rate measurements are stored in FPGA BRAM, and averaged each time a new value is calculated. This averaged value is then output to the display and audio modules. The averaging is introduced to minimize artificial fluctuations in the output heart rate in the case that a particular peak is missed by the peak detector. The window size of 16 was determined experimentally as a value that simultaneously minimized artificial fluctuation while maintaining legitimate variations in heart rate.

D. Display (Both)

The display block is a major component to our user interface for HeartAware. At a high level, this block allows for multiple shapes and icons to be drawn on a VGA display, at 1024x768 XVGA resolution in 4-bit color. We used this block to display the state of the system, current heart rate and oxygen saturation waveform, as well as other graphic sprites. The Verilog modules associated with this block are xvga, main_display, display_modules, display_sprite_map. A visual overview of the display block is shown in Figure 7.

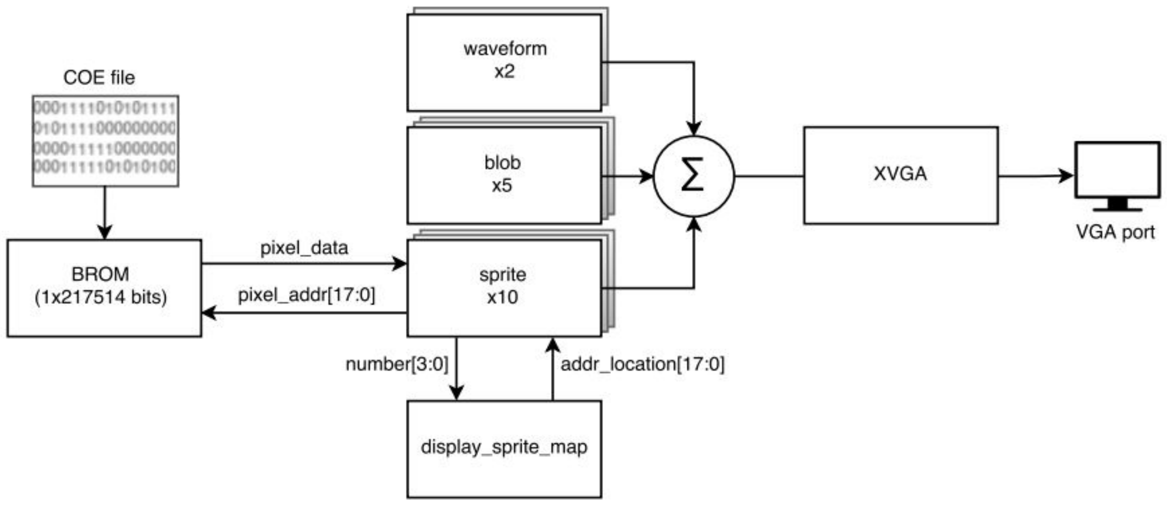


Figure 7: The display block diagram

The lowest level module is the xvga module, which handles timing and synchronization as per XVGA specifications.[3] In particular, the XVGA display refreshes at 60 Hz, with a pixel clock of 65 MHz, with indices hcount and vcount incrementing as each pixel refreshes. In the rest of the display modules, at any given time, the current values of hcount and vcount are used to determine what 12-bit RGB color is shown on the screen.

The user interface and pixel blending is defined in the main_display module. From our user interface mockups, we determined that we needed to display three types of elements: continuous waveforms, rectangles, and pixel sprites from an image. The combination of a number of these modules would let us build a simple yet intuitive interface. Here, we instantiate two waveform modules, five blob modules, and a scanning sprite block. The scanning sprite block checks if the display scan is currently in one of ten unique pre-defined sprite areas, and if so it will prompt the BROM to output the pixel data corresponding to the selected sprite.[4] In any given section of our display, we are capable of displaying one or many waveform, blob, or sprite modules in any combination; because we did not implement transparency, the pixels will overlap.

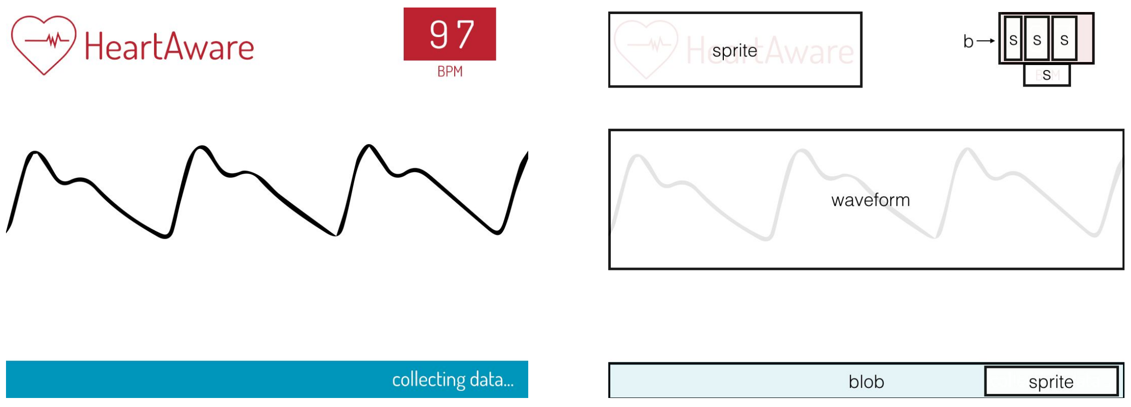
The display_modules module contains the definitions for the waveform and blob modules. Both modules have consistent interfaces: for example, if we wish to hide some element from the screen, we set the enable signal to low, thus it will output no pixels. We can also dynamically change the color by setting the 12-bit RGB color parameter.

The waveform module generates a rolling plot of the lowpass filtered blood oxygen saturation values. In particular, the 1,024 most recent outputs from the lowpass filter are stored in a dual-port FPGA BRAM as they are computed on the positive edges of the 100 Hz clock. Simultaneously, these values are accessed sequentially on the positive edges of the 65 MHz clock governing pixel display. During any given clock cycle corresponding to a particular pixel, the value read from the lowpass filter BRAM is mapped to a vertical pixel coordinate such that the entire range of 8 bit values maps to the middle horizontal half of the VGA display. Then, as the screen refreshes, if vcount matches this mapped value while hcount corresponds appropriately to the index at which the memory is accessed, the pixel is illuminated in the color specified in the waveform module instantiation. In order to achieve a "rolling" effect, instead of directly accessing the location in memory which corresponds to hcount, the memory access location, hcount_sliding, is shifted to the right by one pixel every time a new value is read into memory. As a result, the display appears to shift to the left by one pixel every time a new value is computed.

The blob module is similar to that of Lab 3: it can draw a rectangle of a certain width and height on the screen, starting at a certain (x, y) coordinate. We modified it to have a dynamic visibility/hidden display state, width, and color. This allowed us to implement a loading progress bar for the booting state, multiple modes for our user interface, and overlapping objects.

[3] www-mtl.mit.edu/Courses/6.111/labkit/vga.shtml
[4] See section VI for a discussion about the scanning sprite block approach.

Figure 8 shows the areas the modules required for a unified display interface.



(a) (b)

Figure 8. Breakdown of the graphical modules displayed during collecting data mode. Image (a) renders the display as normal, image (b) shows bounding boxes around display elements. There are a total of nine elements: one waveform module, two blob (b) modules, and six sprite (s) modules

For storing sprites, we used a Block Memory module from Vivado's IP core library. First, we created a sprite map with all the icons, text strings, and numbers we wanted to display made a monochrome 609 x 356 sprite map image bitmap, and converted it to a COE file with the 6.111 provided MATLAB script. This file was then loaded into the 1 bit wide BROM as an init file. The pixel data in COE file is 217,514 bits long, thus this is the bit depth of this BROM. The sprite module then looks up the appropriate start location in BROM, and given a width and height, can draw the relevant icon or text string. The sprite map was made to be as compact as possible while still allowing each sprite icon to fit into an unrotated rectangular bounding box. The sprite map is shown in Figure 9.

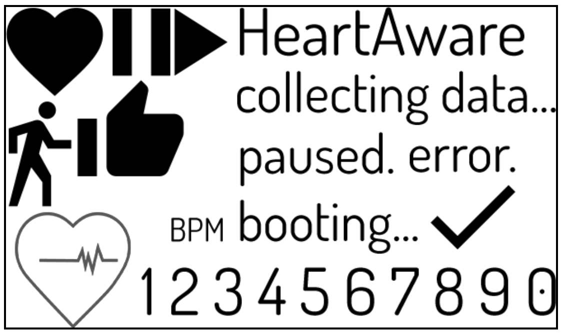


Figure 9. The icon sprite map

We also wanted to generate a number counter readout, for display of the user's current heart rate. For this, we made a sprite for each decimal place (hundreds, tens, ones). Given a heart rate number (0-199) we then had to solve two problems: extracting the digit for each decimal place, then converting that 0-9 value into an address for BROM lookup. The first task required a simple binary to binary-coded decimal (BCD) converter, which we had already built for Lecture Pset 8. We then implemented a display_sprite_map module that mapped a 0-9 number to a start and stop address on the sprite map.

E. Audio (Michael)

The audio block handles selective playback of audio from the microSD card. This functionality is used to play a sound upon finishing boot mode, a beep tone upon detection of the heart rate peak, and periodically announce the heart rate. The final audio block consists of a pwm_audio module, sd_controller module, FIFO module, as well as a audio_number_map lookup module. Figure 10 shows the implemented audio block, with the most important data path for playing back audio samples.

The pwm_audio module takes an 8-bit data sample and converts it to a square wave at a certain duty cycle.[5] The PWM audio module generates a 100 MHz clocked PWM signal; this signal is then output to an analog filter which ultimately arrives at the Mono Audio Out jack on the Nexys 4.

The sd_controller module handles interfacing with the microSD card.[6] Given a 25 MHz clock and an address of a multiple of 512, the module reads data samples from the memory in serial. Sound files are loaded from an unsigned 8 bit, 32 kHz sample rate WAV file written directly to disk, starting at memory location 0. To get the start and stop address for each sound region, we took the start and stop times of the audio sample and multiplied it by the sample rate. An important consideration to note is that all SD reads must start at multiples of 512 bytes, so we rounded all addresses to the nearest multiple of 'h200. For example, the word "fifty" in our WAV file starts at 4.8s and ends at 5.6s, thus the start address would be 'h25_800 and the end address would be 'h2B_C00.

The FIFO connects the SD card with the PWM output module. It provides an asynchronous interface for writing and reading 8-bit audio sample data. The 12-bit *fifo_count* signal is used to control when the SD card should read in a block of new data to the FIFO. We pre-load the buffer with 512 samples to ensure the buffer is always ready to output the next sample, regardless of SD access time or residual delays. Then, at every positive edge of the 32 kHz clock, we read from the FIFO into the PWM output module.

[5] The PWM_output module was provided by Gim Hom.

[6] The sd_controller module was provided by Jonathan Matthews. Note: the sd_controller module worked best with SD cards with a storage capacity 2GB or less, since smaller capacity cards are compatible with the simpler Secure Digital protocol version 1.1.

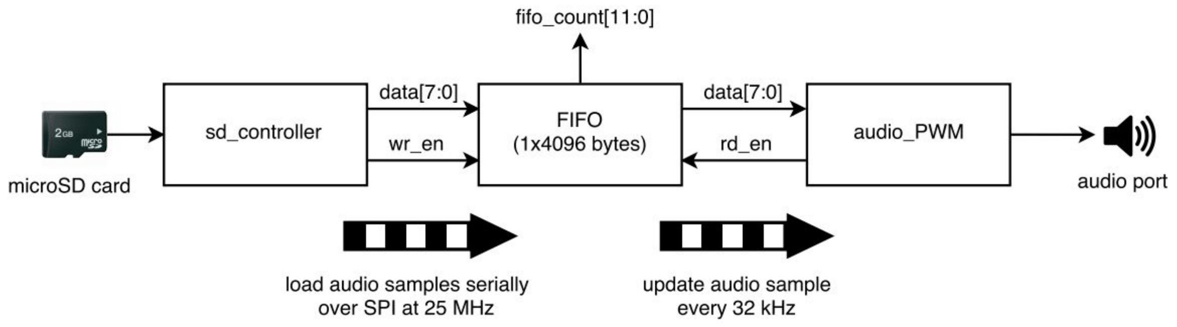


Figure 10. The implemented audio module

For the verbal announcement of heart rate, we recorded the English numbers from "one" to "one hundred ninety nine" as well as the phrase "beats per minute." Figure 11 shows the structure of our WAV file. To announce a heart rate such as "eighty three beats per minute" we use the audio_number_map module. Given a heart rate number, the module looks up the next number to announce, then outputs the heart rate number minus the sound just played. For example, an input of 83 would output the SD card addresses for the "eighty" sound, and a remaining number of 3. When the number reaches 0, the module plays "beats per minute" then stops until the next trigger.

It is crucial to ensure that the beep and the verbal heart rate announcements do not interfere with each other so sounds are not cut off. Thus, our logic within the audio block establishes a lockout on audio playback until any number announcement has finished.

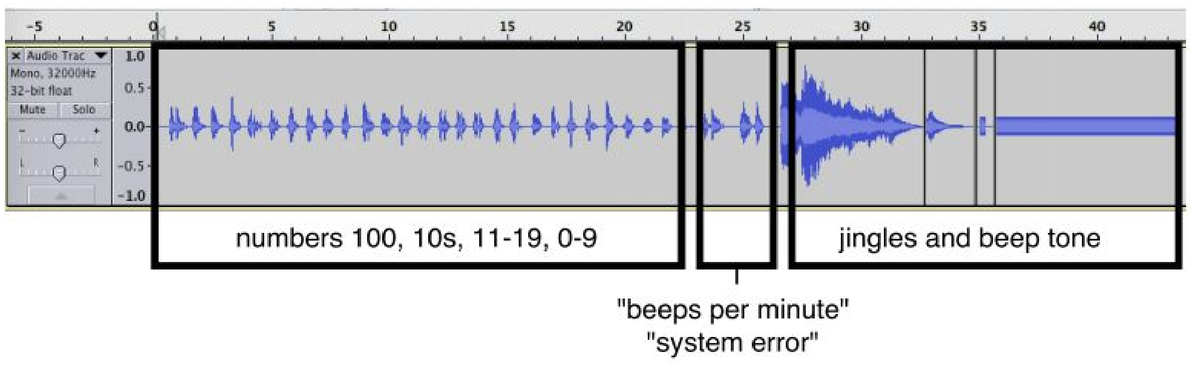


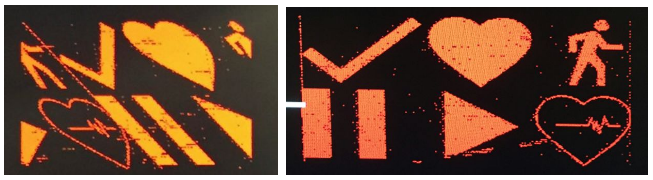
Figure 11. The mono 32 kHz 8-bit WAV audio file containing all project sounds

V. Testing and Debugging

This section will describe our the testing and debugging techniques we used during the development of our project. We relied on simulation heavily for development of the signal processing block. However, because our display and audio blocks relied heavily on external hardware, it was not feasible to simulate the functionality of each module. Thus, we instead relied on the digital logic analyzer in lab for capturing high speed signals for later analysis.

The most common problems we encountered across the entire project related to clocking issues, entering the right state at the wrong time, or never entering states conditions because of a glitchy signal. Another more general problem emerged near the end of our project; because we integrated a lot of modules, Vivado build times often took 20 minutes or more. This delay made it much harder to debug.

One specific challenge was building the sprite display logic. It took several days to troubleshoot the various errors of offset calculations, data flow, and pixel pipelines. An example of one problem we fixed can be seen in Figure 12.



(a) (b)

Figure 12. The first attempts at displaying sprites from BROM. In image (a), the programmed sprite width offset was one less than the actual image width, thus incorrectly rendering the sprites in a diagonal pattern. In image (b), the full sprite is displayed correctly, however there is still a minor problem with pixel noise in the graphic stream.

Apart from the digital design, we did have issues with the pulse oximeter signal on occasion. While a significant amount of simulation went into testing the signal processing for robustness against several different user's oxygen saturation profiles, it was hard to account for the real-world conditions and variability of heart signals.

VI. Design Decisions

This section will describe a few design decisions we had to make during the implementation of this project. Our system is meant to provide information to casual users, not medical professionals. As a result, our design is primarily intended to be intuitive to use. Our focus on user interface, instead of being absolutely precise or reliable, allowed us to implement most features completely. It also allowed us flexibility in the specific details of implementation.

For example, we were originally planning to check to ensure the pulse oximeter was connected to the system. This functionality was present on the PCB, however because we could not implement the PCB in time, we had to forgo this feature. However, we had built in the system error state, to prepare for the integration of this feature.

A. Audio: FIFO vs. Dynamic clocks

On our first attempt, we tried to slow down the clock speed of the SD card to match the audio sample rate. However, the SD card would not communicate at these speeds. Additionally, even at the default 25 MHz, SD card access times were variable, and dynamic clock speed require more calculation than expected. Thus, we implemented a more asynchronous approach by connecting the SD card to the audio output through a FIFO buffer. This approach worked quite well.

B. Main FSM: Boot state

Originally, we designed our system to have a nearly instant startup time. However, we found that initializing the SD card upon reset took several seconds, thus we had to build a boot state to ensure no access was required before the SD module was initialized. In order to provide feedback about the status of this stage, we included a loading progress bar.

C. Display: Transparency vs. Binary hidden/visible states

We implemented a hidden/visible enable signal for all display modules and sprites for ability to switch user interface states. However, a nice feature could have been transparency (as built into our pong game in 6.111 Lab 3). We considered this feature, however mixing many elements quickly became complicated and required a new timing system, as adding many multiplies would not keep up with our 65 MHz pixel clock. Thus, we preferred the simpler binary display enable option.

D. Display: Scanning sprite block

At first we tried to modularize each image sprite into its own instance. This worked well with only two sprites instances; Vivado was able to route the design and two sprites were indeed visible on the screen. However, adding more instances greatly increased the design build time. Thus, the modularized sprite approach was not feasible with a single BROM bank, most likely due to timing constraints or physical limits of the Artix 7 FPGA. Thus, we implemented a less sophisticated “scanning approach” which counted through all pixels of the display. If the current pixel was within a predefined sprite region, it would set the BROM address to the relevant sprite and set pixel data to the BROM data output. This “scanning block” approach led to similar output of the sprite module approach, yet allowed us to add many more sprites on screen.

E. Signal processing: Filter and algorithm design

Each stage of the signal processing block was experimentally adjusted to maximize the accuracy of peak detection and, thus, of heart rate calculation. The design decisions for each stage of the signal processing module are described in section III.C, which details these design decisions in the context of the overall signal processing dataflow.

VII. Conclusion

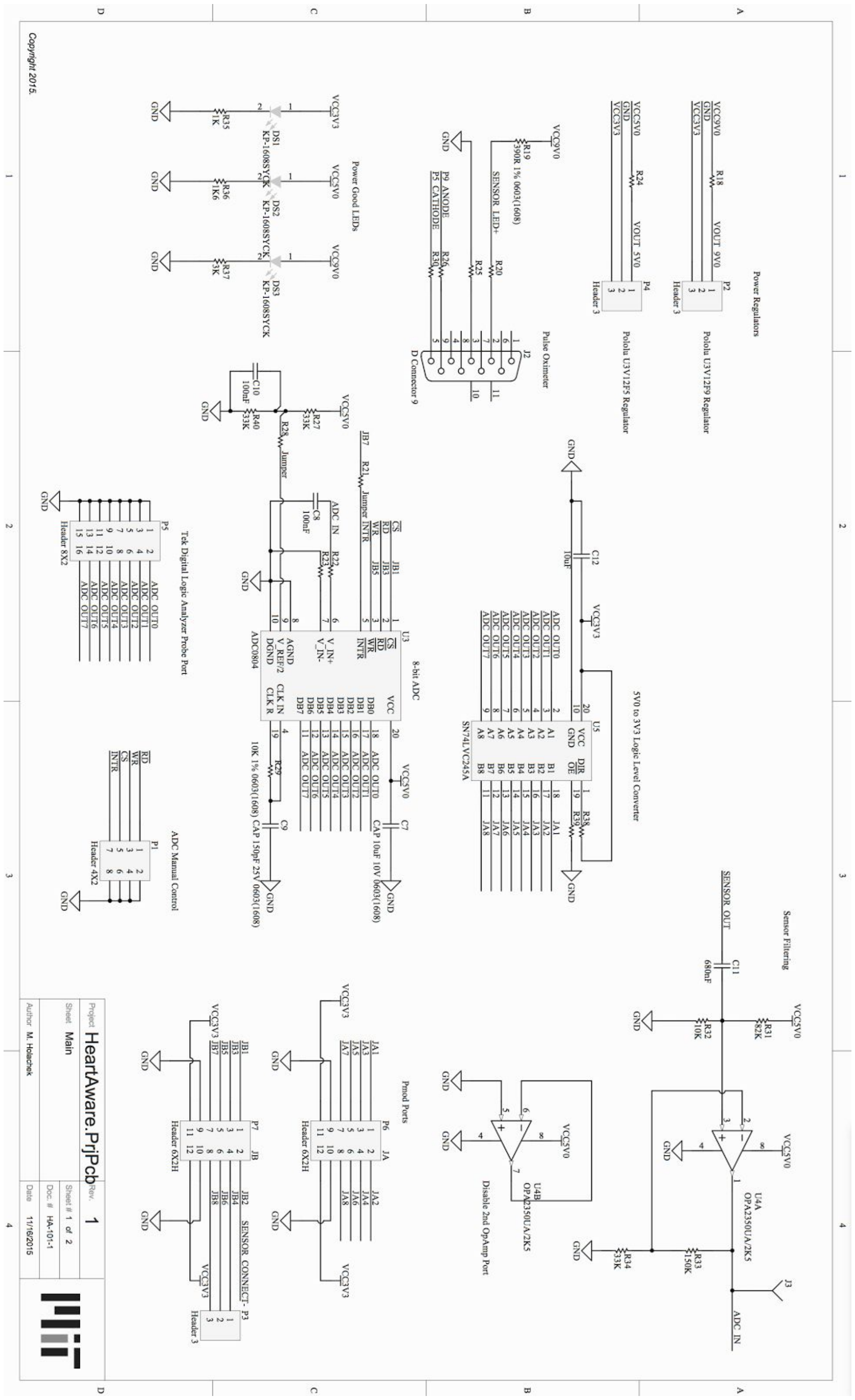
The HeartAware system successfully uses the Nexys 4 process an oxygen saturation signal to determine the user's heart rate and to display and announce it to the user. We believe that there were a number of factors that allowed us to succeed with this project. In particular, we spent significant time designing the system in detail prior to building it; this allowed implementation to proceed more smoothly and efficiently than would have been possible otherwise. Further, we worked well together as a team and effectively communicated all of the issues we ran into as they arose. This allowed both team members to stay informed of the others' progress and for both teammates to assist each other in troubleshooting.

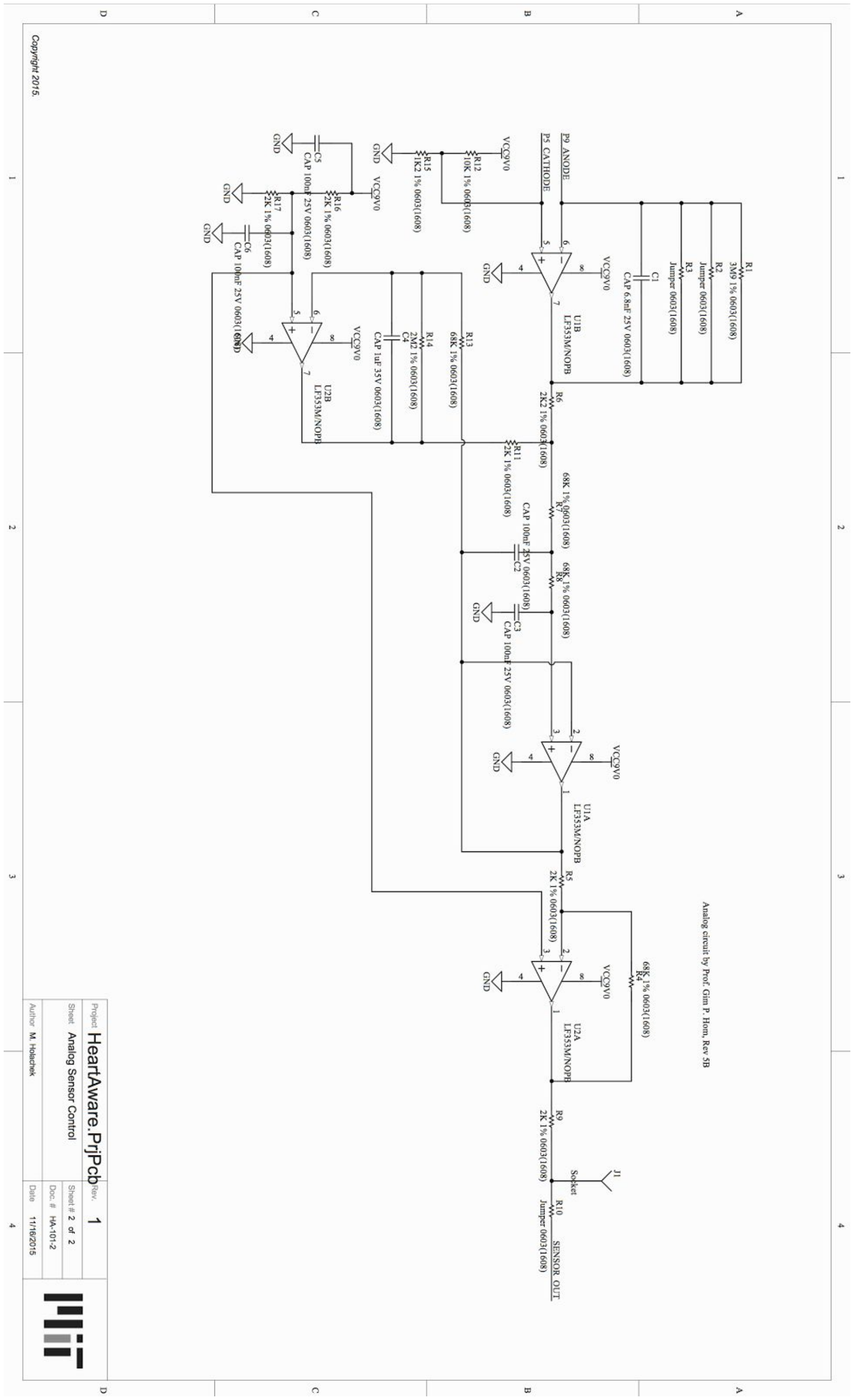
In implementing the system, we overcame a number of obstacles. First, our unfamiliarity with the Nexys 4 made interfacing the board with other system components, particularly the VGA display and audio output, difficult and time-intensive. Additionally, the debug process for the project was challenging. Though the logic analyzer was a useful tool, the restrictions on the number of Pmod ports available given the number of input and output ports being used for other components of our system, often only allowed us to analyze 16 bits at any given time. Additionally, the project grew to have very long compile times, making each debug iteration very costly. In practice, we observed great variety in the waveform shape and amplitude output from the analog-to-digital converter and preceding analog circuitry; this made design of robust peak detection methods and heart rate calculations difficult. However, we were able to overcome these challenges to implement a functional system that accomplished our initial design goals.

Had we had more time to extend the project, there were several additional features that we would have liked to implement. In particular, we had hoped to use a PCB instead of a breadboarded analog circuit for amplifying and digitizing the pulse oximeter signal. Additionally, several other interesting heart rate metrics, such as heart rate variability, could be calculated, and a more informative display, such as a histogram of RR intervals, could be displayed to the user. Further extensions for a pulse oximetry platform project could include detection of abnormal heartbeats or heart rates or implementation of a wireless system.

In designing and implementing HeartAware, we gained a tremendous amount of knowledge about FPGA design, and the experience of implementing a standalone, functional system was extremely rewarding. We look forward to further exploring FPGAs and digital design in the future.

Appendix A. Circuit Schematics




Copyright 2015.
P9 ANODE
P5 CATHODE
VCC9V0
R12 10K 1% 0603(1608)
R15 1K2 1% 0603(1608)
GND
R1 3M9 1% 0603(1608)
R2 Jumper 0603(1608)
R3 Jumper 0603(1608)
C1 CAP 6.8nF 25V 0603(1608)
U1B LF353M/NOPB
R6 2K2 1% 0603(1608)
R16 2K 1% 0603(1608)
R17 2K 1% 0603(1608)
C5 CAP 100nF 25V 0603(1608)
C6 CAP 100nF 25V 0603(1608)
U2B LF353M/NOPB
R13 68K 1% 0603(1608)
R14 2M2 1% 0603(1608)
C4 CAP 1uF 35V 0603(1608)
R11 2K 1% 0603(1608)
R7 68K 1% 0603(1608)
C2 CAP 100nF 25V 0603(1608)
R8 68K 1% 0603(1608)
C3 CAP 100nF 25V 0603(1608)
U1A LF353M/NOPB
R5 2K 1% 0603(1608)
R4 68K 1% 0603(1608)
U2A LF353M/NOPB
R9 2K 1% 0603(1608)
J1 Socket
R10 Jumper 0603(1608)
SENSOR_OUT
Analog circuit by Prof. Gim P. Hom, Rev 5B
Project HeartAware.PrjPcb
Rev. 1
Sheet Analog Sensor Control
Sheet # 2 of 2
Doc. # HA-101-2
Author M. Holachek
Date 11/16/2015
A
B
C
D
1
2
3
4