Ishan Dane NetID: idane
Ismail Memon NETID: ishmemon

Lab 6

Design Procedure

In lab 6, we created an alarm clock that had a visual stimulus on top of an auditory one and allowed the user to switch between two different audios. We did this by creating a module that counted every clock edge as 1 second, which allowed our 24 hour clock to run. When the set alarm time was equal to the clock time, the alarm would "ring" (play the sound, light the light, and show the animation) for a total of 31 clock edges (so like 31 seconds). We did this by creating multiple ROMs using the IP catalog, each one representing a different musical note. We then cycled through the notes producing two different sounds from different combinations of the notes.We then created a flashing visual animation on the alarm clock display through 2 cases: draw and clear, which alternated continuously to create a flashing effect. The image was created by providing x and y parameters with r, g, b parameters to the video_driver that would implement the visual design on the display. Finally, we connected that audio piece with our visual display, so that when the alarm rang it would enable the sound to play and the display to be activated, creating a light, audio, and animation show when the alarm rang.

Clock Counter

The very first module we made was the universal_clock module. This module incremented a counter on each clock edge, representing a second of time. The counter and therefore clock could only be reset by the user through Key 0, otherwise the clock kept going on. The counter would automatically reset at the value 86,400 (which is the number of seconds in one day). After that, we created a display_clock module, which took in that seconds counter and displayed the hour and minute (the minutes were displayed as 00, 15, 30, or 45 so that we could actually see the results during our demo) on the HEX. The display format was in military time, so 5:00 pm would be represented as 17:00. We did this part by using if statements to convert the data of the seconds counter into minutes and hours, and then converting those binary values into decimal. We used the seg7 display provided to us to actually display the values on the hexes.

Setting the Alarm

The user enters the time they want the alarm to ring via the switches. The user also uses military time, and uses switches 6-2 to represent the hour (from 0 to 24) and switches 1 and 0 to represent the minute (which will be either 00, 15, 30, or 45 for [1:0] switch values 00, 01, 10, and 11, respectively). The alarm_time module checks if the input by the user and the current seconds counter are equal. If so, the ring signal (which is an output for the alarm_time) is asserted. To allow for the alarm to actually be practical, we chose for it to ring for 31 clock edges (so 31 "seconds"). We implemented that by using or operators. The user's entered time has to be equal to the current time or the current time plus one second, or the current time plus two seconds, and so on. This allows the signal to be asserted for a longer time so that the user can actually see the flashing lights and hear the alarm tone and wake up. Figure 1 below shows a graphic of how this works.

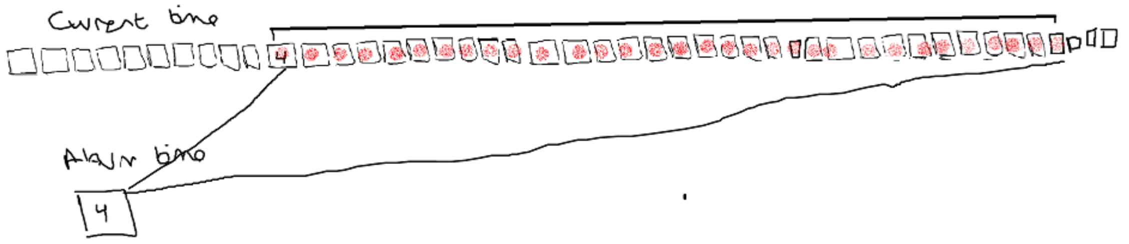


Figure 1: This graphic shows how the alarm_time works. Once it determines equality, 30 consecutive values of the current time are also asserting the output ring (the ones that assert ring are red) so for a total of 31 clock edges the ring is asserted.

Alarm Sound Memory Unit

To create the sound for the alarm, we leverage the audio package provided to us in lab 3. We also used the same instructions to create read only memories (ROMs) to allow us to have different notes. Like in lab 3, one ROM contained one note, but now we had 6 different ROMs representing 6 different notes. The address was incremented each clock edge so we could go through the different values contained in the ROMs. We passed the output of the six ROMS to a soundmix module. This module was a simple FSM that cycled through each of its 6 inputs and asserted one of them at each time (depending on which state it was on). This allowed us to cycle through the different notes. A diagram of the FSM for soundmix is shown below in Figure 2.

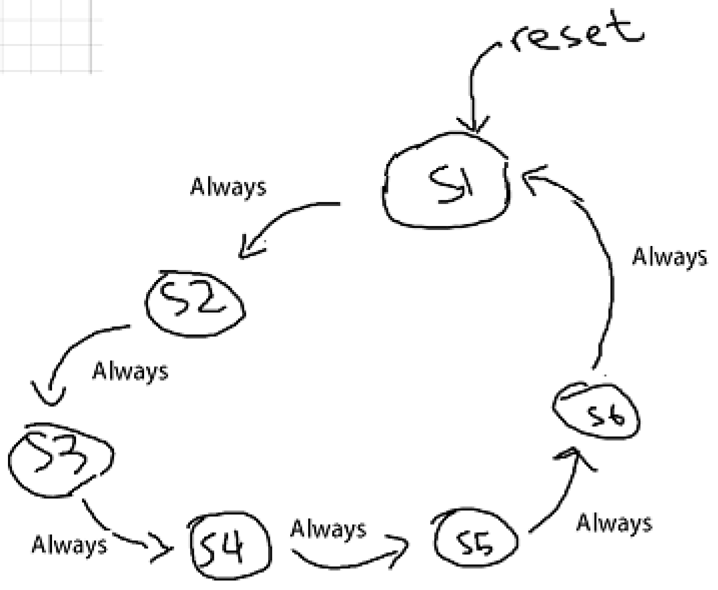

$$\begin{tikzcd}
 & & \text{reset} \arrow[d] & \\
 & S2 \arrow[dl, "\text{Always}"] & S1 \arrow[l, "\text{Always}"'] & \\
S3 \arrow[r, "\text{Always}"'] & S4 \arrow[r, "\text{Always}"] & S5 \arrow[r, "\text{Always}"'] & S6 \arrow[ul, "\text{Always}"']
\end{tikzcd}$$

Figure 2: A state diagram of the FSM for soundmix. As we can see it is cyclical and goes through the states regardless of any input/output.

The DE1_SoC (top module) instanties two soundmix modules, and passes them different inputs. The first one receives the 6 different notes to create a 6 note tune, the second receives one note for its first three inputs and another note for its second three inputs. This allows the second tune to be different since it only has two notes. Switch 9 is what the user uses to chose between the two different notes.

Alarm Visual Display

To create the visual display on the alarm clock, we used the new VGA video driver provided to us. In the DE1_SoC, we used if statement parameters for the x and y outputs provided by the video driver. Our parameters would then set the pixels we wanted to the color red to create the design we thought of. After drawing out our design on the VGA board and implementing it correctly through parameters, we created a flashing effect for the image. This was done by creating 2 cases: draw and clear. For the draw case, we placed all our drawing parameters from before and for our clear case, we set all x and y pixels to black, this would clear the board. Using an always comb block to set state transitions, we implemented it so that the states would alternate between each other thus creating the flashing effect we wanted. The next step was then picking out the right clock speed so that the flashing image was not too fast that you cannot see and also not too slow where nothing shows up for a while. To do this, we implemented the clock_divider module provided to us from EE 271 which allowed us to slow down our clock. We decided to go with divided[24] for our clock speed.

Overall System (Top level):

Our top level module is DE1_SoC.sv. In this module we brought everything together. We instantiated the universal_clock to continuously run our seconds counter, the display_clock to always display on the HEX displays and alarm_time to determine when the alarm should ring. We also had the six ROMs to generate our six different notes, and we had the soundmix to cycle through and provide us with two different sounds. We also used the provided audio and video_driver code to instantiate the VGA display and Audio Codec interface. We used the ring signal to act as an enable signal for the VGA and audio, so the animation would only appear and the audio sound would only play when the alarm rings. When the alarm rings, LEDR[9] also lights up. Our animation flashes on and off and so we decide to use the 24 Hz clock. This was a clock that allowed us to see the flashing and it was fast enough, so we had this constant clock for the video animation. The clock that controlled how long a second is is not necessarily the same clock, and this is to allow for different simulations using different clocks to work.

Block Diagram for our System

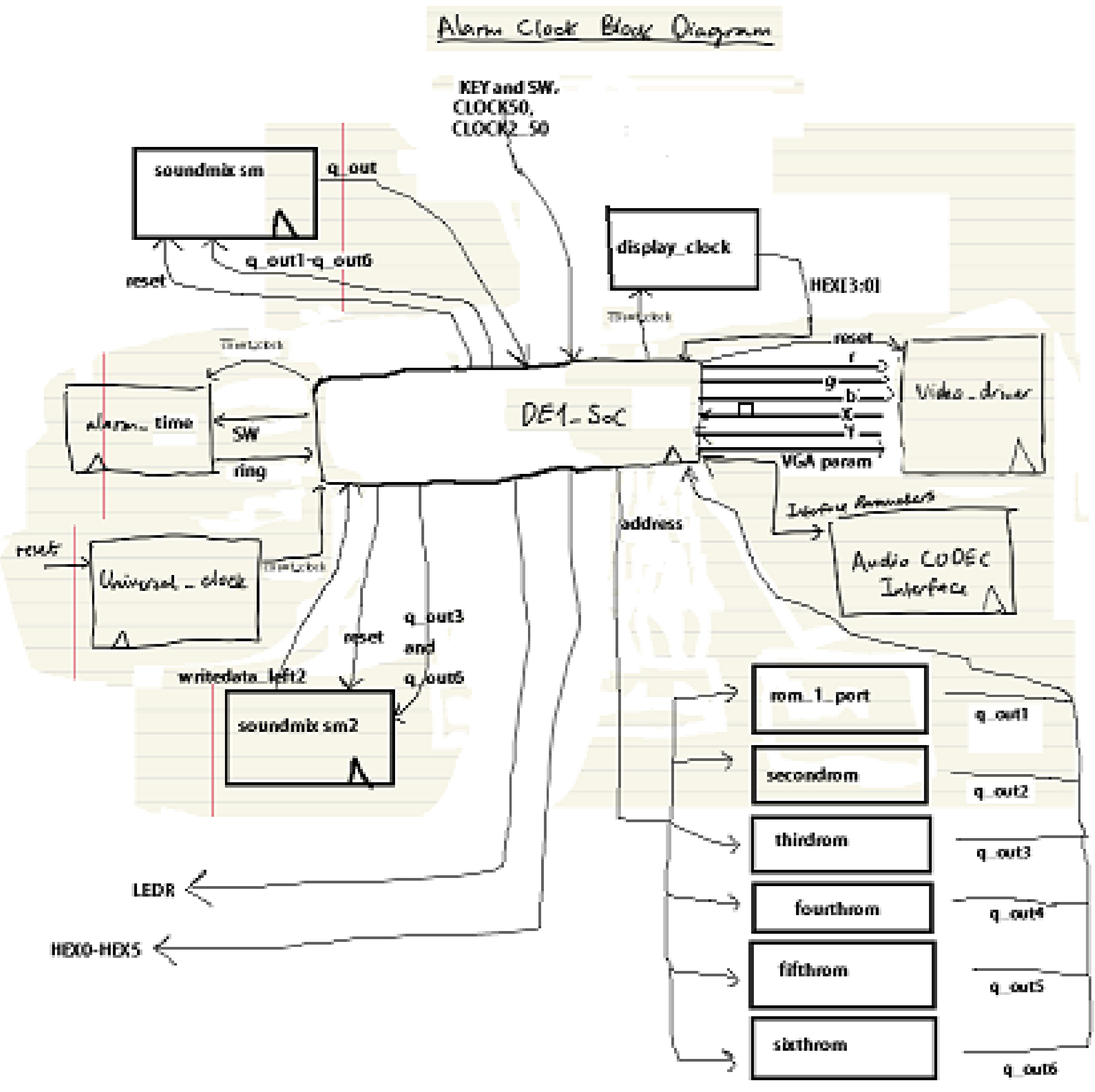
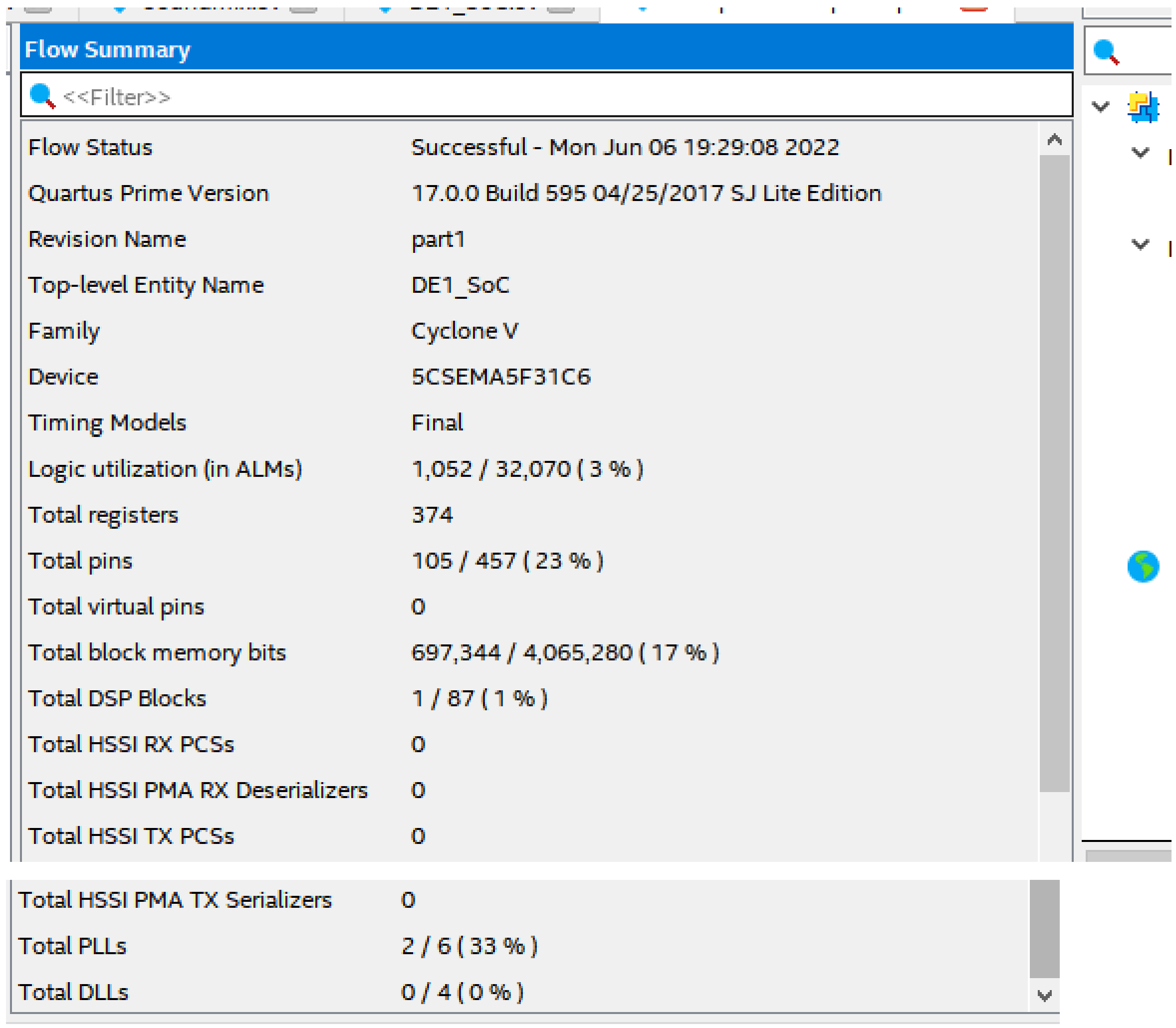


Figure 3: Block diagram for our complete system.

Flow Summary



Flow Summary

<<Filter>>

Flow Status	Successful - Mon Jun 06 19:29:08 2022
Quartus Prime Version	17.0.0 Build 595 04/25/2017 SJ Lite Edition
Revision Name	part1
Top-level Entity Name	DE1_SoC
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	1,052 / 32,070 (3 %)
Total registers	374
Total pins	105 / 457 (23 %)
Total virtual pins	0
Total block memory bits	697,344 / 4,065,280 (17 %)
Total DSP Blocks	1 / 87 (1 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	2 / 6 (33 %)
Total DLLs	0 / 4 (0 %)

Figure 4: Flow summary of our project.

Results

Universal_clock simulation

To test out universal_clock, all we did was let the simulated clock run for thousands of edges and we wanted to make sure that it would reset at 86,400. Figure 5 shows that this happens since we go until 86, 399 and the clock resets.

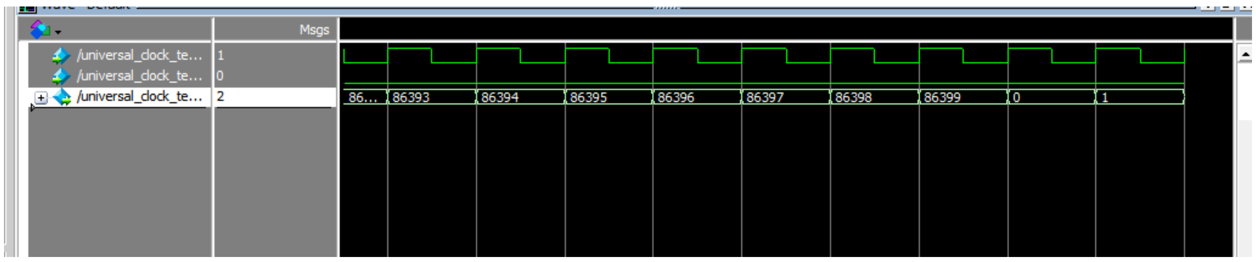


Figure 5: Simulation of the universal_clock, we can see it resets at 86,400.

Display_clock simulation

To test out display_clock, we tested out different values of the seconds counter input and checked to see whether the display hexes held the right numbers. The sequence we expected to see was 00:00. 00:15, 00:30, 00:45, 02:45, 11:45, 18:45, and 21:45. Figure 6 shows that this happens since the numbers do follow this pattern.

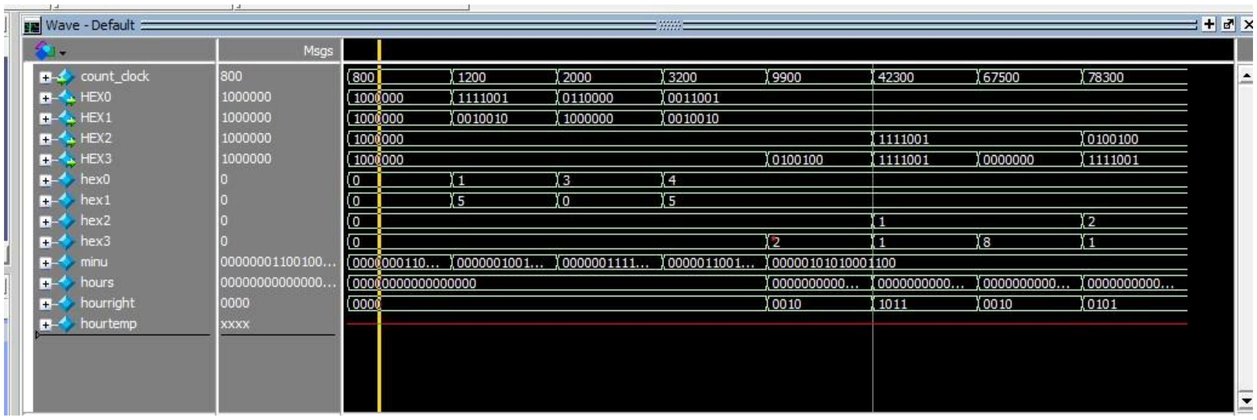


Figure 6: Simulation of the display_clock, we can see the numbers are expected (we are looking at the signals hex0, hex1, hex2, and hex3 not the capitalized versions since those are the values for the display lights).

Alarm_time simulation

To test alarm_time, we needed to make sure that ring is asserted when the user's input time equals to the current time and stays on for 30 clock edges after. We set the testbench to have the seconds counter to change values and first be less than the specified time, then equal, and then greater than. We wanted to make sure that ring was asserted for 30 more clock edges. The equality point was 900, so 900 was the specified value and we expect the ring to be deasserted after 30 more edges (and so 931 is when it should stop being asserted since the seconds counter is incremented once every clock edge). Figure 7 displays this.

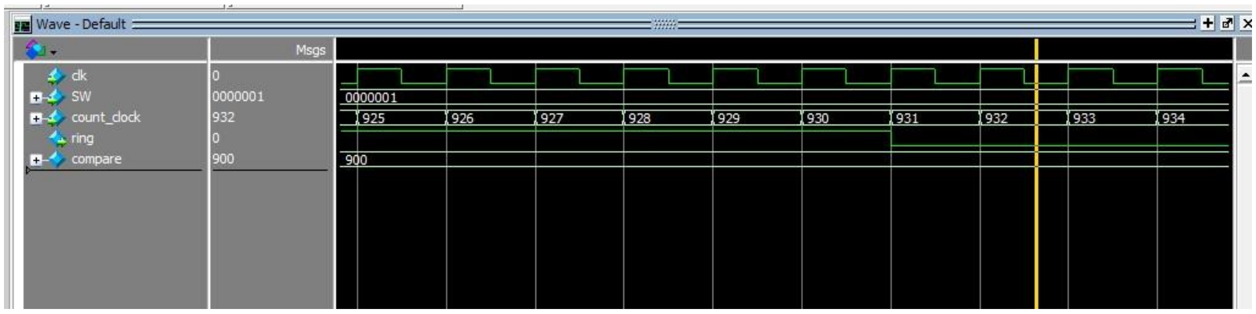


Figure 7: Simulation of the alarm_time, we can see that the ring is deasserted 30 “seconds” after the specified alarm time is equal to the current time.

Soundmix simulation

To test soundmix, we just made all the inputs different (so input q1 had the value 1, q2 had value 2, and so on) and we wanted to make sure that once the module is resetted, we start from q1 and go all the way to q6 and then just keep cycling. We see that is the case in Figure 8.

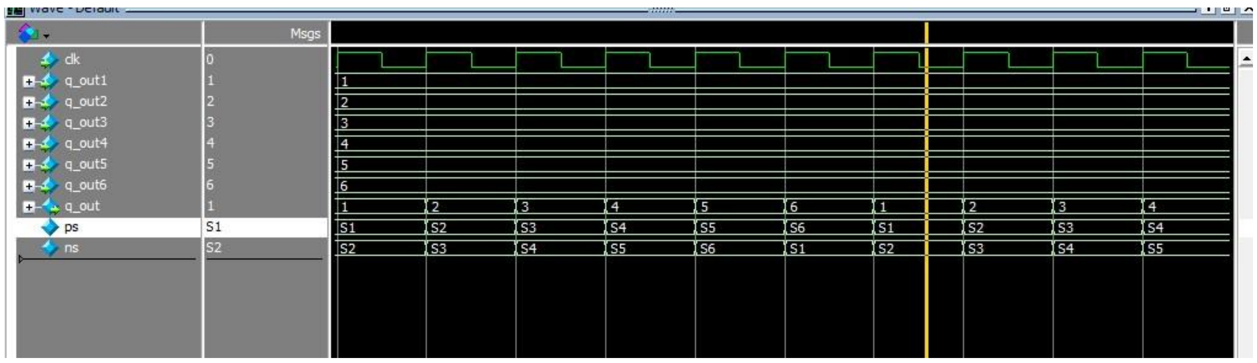
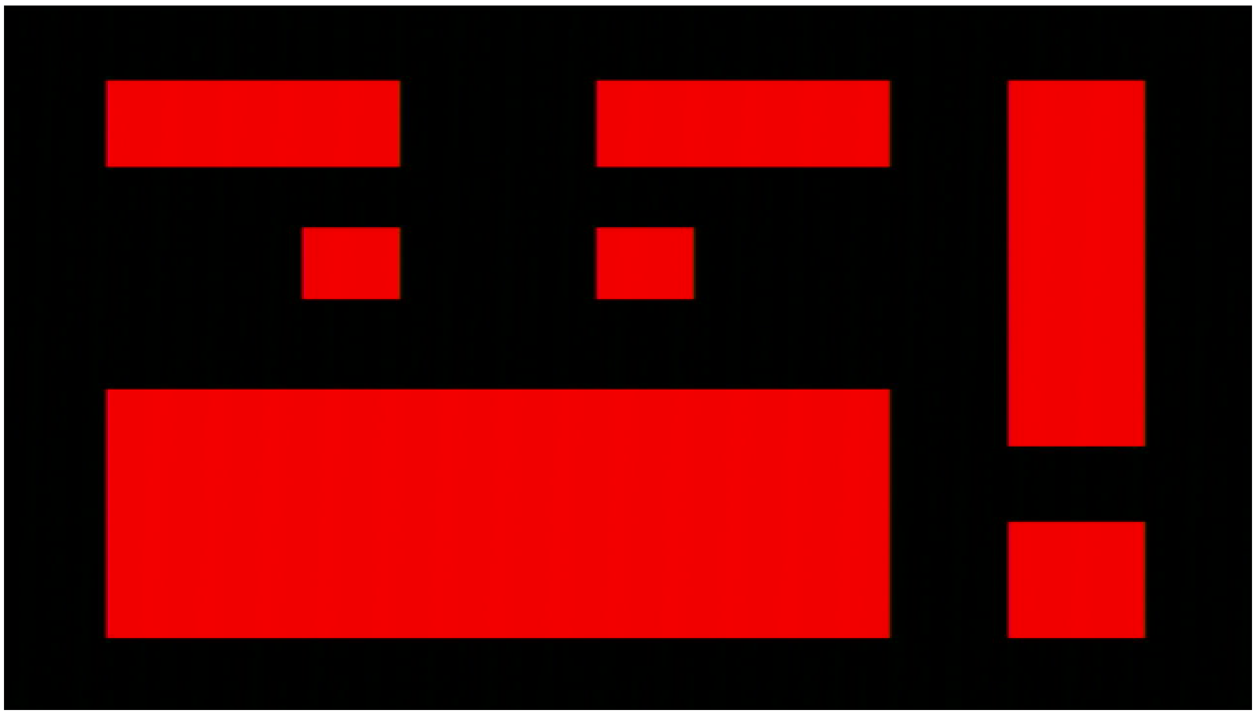


Figure 8: Simulation of the soundmix, we can see that the q_out output cycles from 1 to 6 and then goes back to 1.

Alarm Visual Display Simulation

For our VGA display, our main goal was to create a design on the board to accompany the alarm sound. Since this is done with parameters and setting r,g and b, we cannot simulate this on MODELSIM and the best way to make sure our design worked was to see it on the board for itself and adjust it from there. We switched between 2 cases of draw and clear as long as the alarm signal was ON. It was unnecessary to test for the flashing case since it was just alternating between 2 different states instead going through several different ones. What we did have to test for was if the flashing image was too fast, too slow or not happening at all which had to do with the clock we implemented into the design. This cannot be tested on MODELSIM which operates on CLOCK_50 for simulation, and it is much easier to test it on labsland through trial and error to pick the right clock speed.

Alarm Visual Animation:
Here is a picture of what our animation looks like. Using the VGA parameters for x and y, we decided upon this animation which flashes on and off continuously as long as the alarm is on. The animation goes from this image of a red face and exclamation mark to a black screen and back to this image in a continuous flashing motion to wake up the user by providing a visual effect on top of the auditory one from the alarm.



Experience

We found this lab to be alright. After 5 difficult labs, we learnt a lot and were able to apply our knowledge from the previous labs into our project. Since we already went through all the debugging issues and all nighters figuring them out on our previous labs, we went into our final project with the experience of being able to quickly debug parts here and there as problems that would show up were similar to previous labs, specifically our 2 port RAM lab. What took a significant chunk of time in this lab for us was not understanding the new VGA video driver provided. We thought for the longest time that we still had to use our line drawer module from the previous lab to create our designs on the VGA. We had misread the new video driver instructions thinking that the only difference between the video driver and the previous VGA framebuffer was that you were able to color in your lines as it was drawing by setting x and y. Therefore we tried to create the same design animation above with the same flashing effect through several different cases on the animator. However, we kept running into problems with the VGA not drawing what it was supposed to or not drawing at all. Therefore we spent a significant amount of time trying to debug the animation and figure it out. We were then told by a TA that we did not need to do any of that as the new video driver scans the x/y of the board and colors can be straight inputted into the pixels without the need of the line drawer. This definitely made everything much easier and we were able to create our animation with almost no debugging besides figuring out clock values and coordinates.

On the audio side, we had to figure out how to create an actual tune using memory. We had already learned how to play a one note tune using the IP catalog ROM for lab 3, and now we had to create multiple modules so we could play different notes. We did not realize that we had to create new entire modules for the different notes and could not just use the same ones again. We also had to figure out how to make a series of notes play in sequence, however, that ended up being quite simple since it was just an FSM. The clock was simple, since the seconds counter just incremented each clock edge and all we had to do to assert ring was use an equals statement. What was difficult was getting the clock to display right. We had to convert the seconds into minutes and the minutes into 15, 30, and 45 minutes and then convert those into hours. We also had to display those values on the HEX. That was quite difficult since there was quite some debugging too since there were small things that could just become big errors and cause incorrect numbers. Putting it all together, we needed a way to make the animation visible and flash fast enough, so we finally decided to have an independent clock that would be controlling the animation, which will keep it from changing pace and just be a steady flashing animation.

Partner Work Summary:

For our project, we split our alarm project up into visual and audio parts. One person worked on ensuring the alarm was creating a tone and the other on the image being displayed on the VGA by the alarm. We then worked together to connect and implement our designs onto the DE1_SoC to actuate the alarm clock with a clock counter and alarm system. From there, we worked together on debugging issues that would arise during compilation and on labsland.

This lab took us approximately 35 hours, broken down as follows:
Reading – 1 hours
Planning – 4 hours
Design – 3 hour
Coding – 9 hours
Testing – 3 hours
Debugging – 15 hours (including time spent trying to figure out the line_drawer module)