FPGA Implementation of Simple Quality of Service (QoS) based Queuing

Afer Olkay
Department of Electrical & Electronics Engineering
Middle East Technical University
Ankara, Turkiye
olkay.afer@metu.edu.tr

Mustafa Soylu
Department of Electrical & Electronics Engineering
Middle East Technical University
Ankara, Turkiye
e216715@metu.edu.tr

Hüseyin Yalçın
Department of Electrical & Electronics Engineering
Middle East Technical University
Ankara, Turkiye
yalcin.huseyin@metu.edu.tr

***Abstract*— This electronic document simply explains how to design and how to use a basic FPGA implementation of a simple quality of service-based queuing.**

Keywords—data, buffer, VGA, receive, transmit, drop, QoS

I. INTRODUCTION

Today's electronic devices, such as computers and IOT devices, transfer data from one another, and amount of transferred data which are transferred between two electronic devices increasing rapidly. Therefore, the data traffic between these electronic devices needs to be controlled because we need to ensure that the data transferred between these electronic devices must meet some important criteria; in our project they are reliability and latency requirements. For example, in the case of military messages, these two criteria are of vital importance. One would like to minimize the data loss as much as possible, so we need to develop some algorithms with this purpose. Or sometimes speed is important, and we need to deliver data to a destination rapidly. To satisfy these requirements, a simple quality of service-based queuing on FPGA is designed to control the data traffic.

II. PROBLEM DEFINITION

A. Problem Summary

We were expected to design a simple quality of service based queuing system, using FPGA board. We needed to design a system that has four buffers, each differs from another in latency and reliability criteria, and each has some allocated space to store data up to 6 data packets. Our system must place the inputted data into correct buffers and then choose the most optimal data as output every 3 second such that overall latency and reliability are satisfied as much as possible.

In other words, some portion of data may be lost or dropped, and some data packets have higher importance related to others, so we need to make sure lost data are not data of importance. On the other hand, some data loses its importance with passing time, therefore we need to make sure they are the ones transmitted first.

Also, we should be able the follow how much data is being hold in buffer or how much data has been received, dropped, or transmitted. To achieve this, we must use an LCD monitor and drive it with VGA interface.

B. Latency Requirement

As we talked in *Problem Summary*, some of the input data needs to be delivered to output immediately whereas for some data speed is not an element of essence. In our design, Buffer 1 has the highest speed prioritization, followed by Buffer 2 then Buffer 3 and finally Buffer 4. Which means according to the degree of importance of latency, received data should be placed in an appropriate buffer.

C. Reliability Requirement

As we talked in *Problem Summary*, some of the input data needs to be delivered to output for sure, regardless of when it is delivered, whereas for some data, reliability is not an element of essence. In our design, Buffer 4 has the highest reliability prioritization, followed by Buffer 3, then Buffer 2, and finally Buffer 1. Which means according to the degree of importance of reliability, received data should be placed in an appropriate buffer.

D. General Requirements

Generally, it is aimed to drop as few packets as possible and make sure dropped packets has lower reliability importance. Similarly, packets with higher low-latency importance must be transmitted as quickly as possible. Moreover, visualization of the four buffers with their content and the number of transmitted packets from each buffer as well as number of dropped & received packets belonging to each buffer is a must. This way, we can follow the algorithm and judge whether it works as intended or not.

III. PROJECT SUMMARY

When we examined the requirements of the project, we decided to design a finite state machine consisting of three sub-modules for the data input and storage. These sub-modules are input module, buffer module and output module. They are connected to each other combinationally. Our algorithm is written in the "**overall. v**".

A. Input Module

In this module, we need to take the 1 and 0 inputs serially after pressing start button and transmit them to the other modules as a 4-bit output. For this, we designed a small state machine consisting of two states which are called idle state and input state.

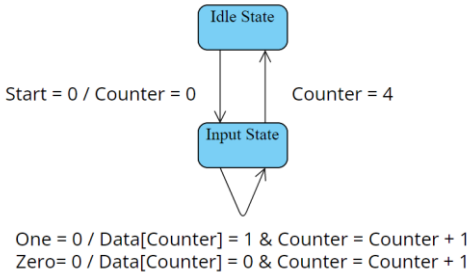

Figure 1. State diagram of input module

The state machine starts in the idle state. We assigned the start, one and zero inputs to the buttons. Since the buttons are active low, they give 0 when we press. So, when we press the start button, the machine goes to input state and sets the counter to 0. Each time we press the one and zero buttons, inputs are entered into the data serially and the counter increases by 1. When the counter reaches 4, 4-bit data is obtained and the machine goes back to the input state.

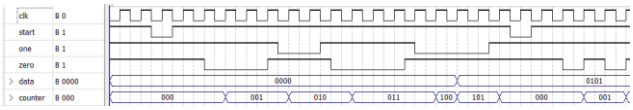

Figure 2. Simulation of input module

B. Buffer Module

In this module, we take the data obtained from the first module as input and insert it into one of the four buffers according to the most significant two bits.

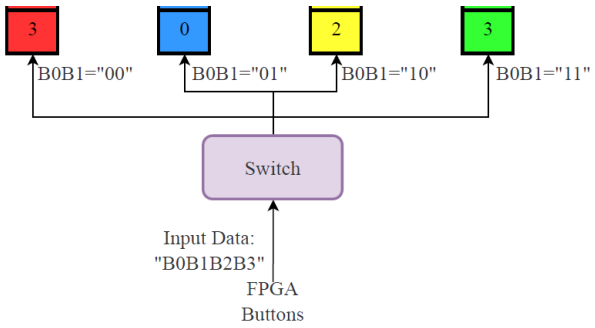

Figure 3. Buffers according to the data's first two bit

As seen in above [figure 3], if the first two bits of the data is '00', the data should be inserted in Buffer 1. If the first two bits of the data is '01', the data should be inserted in Buffer 2. If the first two bits of the data is '10', the data should be inserted in Buffer 3. If the first two bits of the data is '11', the data should be inserted in Buffer 4. We used if-else statements in order to achieve this criteria.

A counter is defined for each buffer to count how many data packets are present in each buffer. These counters are called bf1_counter, bf2_counter, bf3_counter and bf_4 counter. If there is a data entry to that buffer, the counter of that buffer will increase by one. We have defined two 6-bit registers to hold six 2-bit numbers in each buffer. These registers are called bf1_0, bf1_1, The first register is defined for the most significant bit and the second register for the least significant bit of the data.

We have defined three counters for each buffer to keep the number of data received from input module, the number of data transmitted to output module and the number of dropped data from buffers. As a result, we have twelve counters for this purpose. These counters are bf1_read, bf1_dropped and bf1_total.

When data is entered into the buffer, the total counter of that buffer is also increased by one.

Each time new data comes into the buffer, the data needs to be shifted up. If a buffer is full but new data comes to that buffer, the previous incoming data moves up one by one and the data in the 6th register that came first is dropped. Therefore, the dropped counter belonging to that buffer is increased by one.

C. Output Module

In this module, one data needs to be read every 3 seconds. We have defined a counter which is called modified_clk_counter for this. When this counter reaches 150 million, a data read from buffers. The reason why the counter counts up to 150 million is that the on-board clock works with a frequency of 50Mhz. The on-board clock hits 150 million in 3 seconds.

Every time modified_clk_counter hits 150 million, a data selected from buffers and it is read accordance with the latency and reliability requirement. The data read is a 4-bit binary number. The first two bits indicate from which buffer it was read, and the last two bits indicate what the data is.

Buffer 1 > Buffer 2 > Buffer 3 > Buffer 4

Figure 4. Latency presedence

This means that the lowest delay is desired for Buffer 1.

Buffer 4 > Buffer 3 > Buffer 2 > Buffer 1

Figure 5. Reliability presedence

This means that the lowest packet drop rate is desired for Buffer 4.

Buffer 1	Buffer 2	Buffer 3	Buffer 4
6	4	2	1
10	8	5	3
12	11	9	7
13	14	16	19
15	17	20	22
18	21	23	24

Figure 6. Data reading precedence

Data reading priority is shown above [figure 6] from 1 to 24. As we don't want any data loss in Buffer 4, it has higher priority when it is almost full. We created our algorithm using if-else statements. When it is time to read data, the number of buffer counters we have defined before is checked in order. First, it is checked whether there are six data packets in Buffer 4. If there are six data packets, the first data is read, and it is waited for 3 seconds to read data again. If there are less than six data packets, it is checked whether there are six data packets in the Buffer 3. This check continues according to the table above [figure 6] and is repeated every 3 seconds. When the data is read, the counter of the buffer from which the data is read is reduced by one and the read counter is increased by one.

D. VGA Synchronisation

We need to monitor what are the content of the buffers and the numbers that's shows the amount of received, dropped, and outputted data. To achieve this, a monitor is used and driven via VGA interface at 640x480 resolution.

In the project, a whole module of Verilog code ("vga_sync.v") is dedicated for the adjustment of VGA signals. Below, the required signals to drive the monitor [figure 7,8] and our simulation results [figure 9,10, 11] are shown.

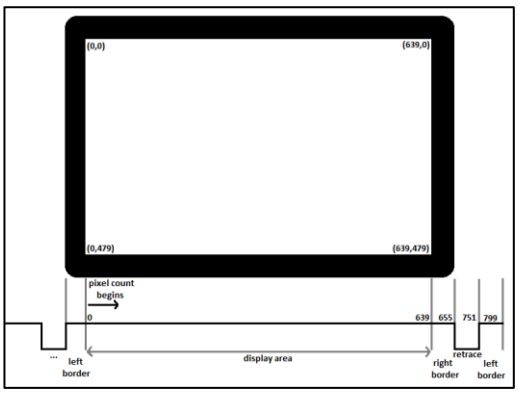

Figure 7. Required pixel counts for horizontal synchronization.[1]

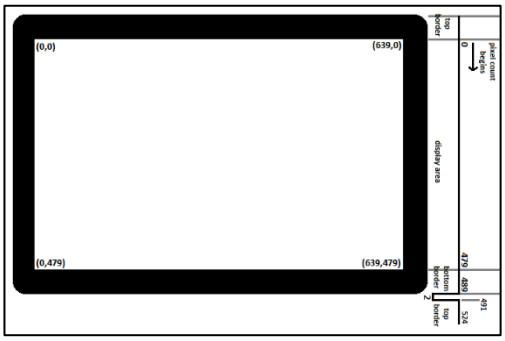

Figure 8. Required pixel counts for vertical synchronization.[1]

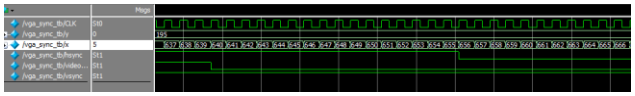

Figure 9. Simulation results for synchronization signals.

Figure 10. Simulation results for synchronization signals.

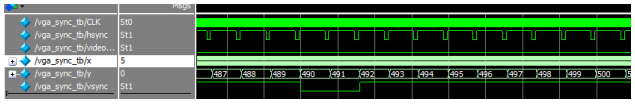

Figure 11.Simulation results for synchronization signals.

Moreover, to achieve 60 Hz frame rate, it is required to supply the monitor with 25 MHz signal. This signal is obtained by dividing the FPGA's internal 50MHz clock using a single D flip flop. Therefore, the module that we created takes only the internal 50 MHz clock as an input and provides output signals called v_sync, h_sync, video_on and x_loc, y_loc. While v_sync and h_sync are supplied directly to the VGA pins to sync up with the monitor, video_on and x, y location signals are used in the main module to decide when and to where to send the corresponding bits to RGB pins of the VGA interface.

E. RAM Implementation

Some prearranged images are stored in the read-only-memory (ROM) blocks that is created. These ROM modules are named as follows in the project:

- "**deneme.v**", "**deneme2.v**", "**deneme3.v**", "**deneme4.v**": These files created to store the general 640x480 pixel frame [Figure 12] that needs no change throughout the running program and contains buffer frames, arrow pointers and words that indicates what the numbers on the right represent. The image is divided into 4 parts and converted to ".tv" files by MATLAB only to be read via "readmemh" function into the ROM of the FPGA [3]. The reason it is divided into 4 parts is to parallelly compile and create the ROM files and cut down the compilation time by ¾.

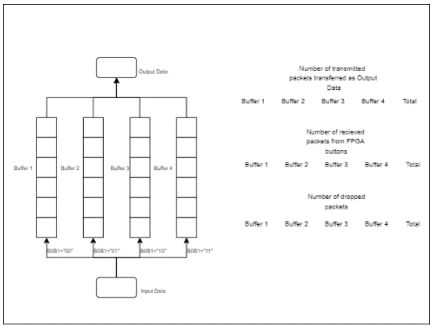

Figure 12. Constant 640x480 pixel file that represents general frame of the display.

- "**pure_numbers.v**": This ROM file stores the numbers 0-9 with white background. They are required to express the number of dropped, received, and transmitted packets. They can be seen below [Figure 13].

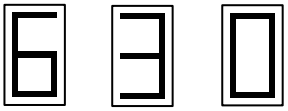

Figure 13. Few of the white background numbers with 9x15 pixel size.

- "**colored_numbers. v**": Some numbers are needed to fill in the buffer frames with correct colours when the inputted data placed into the correct buffer with the correct number. They come in four different colours for 4 different buffers and numbers "0", "1", "2", "3" to represent different data packets. Each number printed on each colour has been stored (4 colours*4 numbers=16 images) to cover all scenarios. Here some examples [Figure 14]:

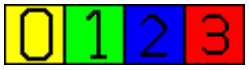

Figure 14. Some of the numbers with coloured backgrounds with size of 30x30 pixel.

F. Display Selection

As explained in the "*Ram Module"* section, project has essentially has 3 memory blocks, which means when display is driven, data to be fed to RGB inputs of the VGA is taken from one of the three blocks. All needs to be done is to decide which ROM to take RGB data from, and which number or color to print to a specific area. Algorithm that is used in this project is coded into the "print.v" file and explained in the diagram below [figure]:

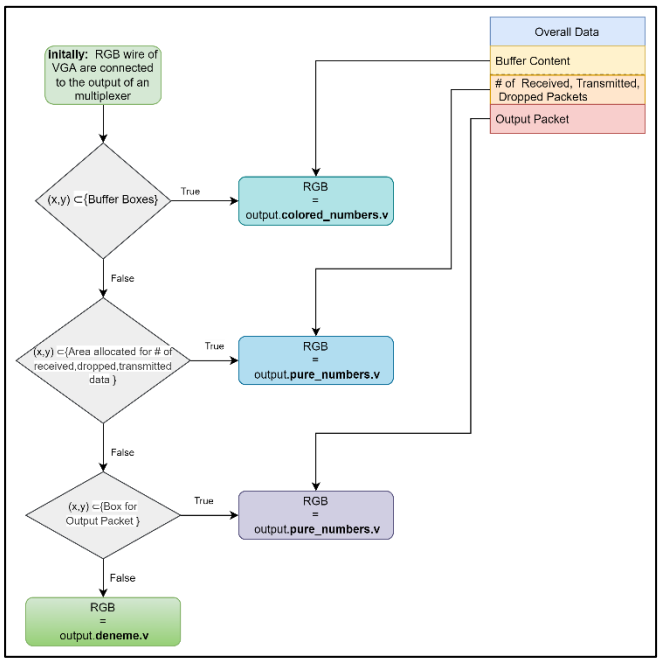

Figure 15. Diagram of the algorithm coded in the "print.v" file.

First thing to notice in the diagram is that overall buffer related data is always fed to the relevant modules. For example, the signal "Buffer Content" is only fed to "coloured_numbers" module, since buffer content must be displayed with numbers with colored backgrounds whereas "Output Packet" data is only fed to "pure_numbers" .

In the screen printing algorithm, the module "vga_sync" supplies the current x and y pixel locations [(x,y)]. In addition, we can form the areas (in pixel adress) belonging to "buffer boxes", "box for ouput bits" etc. by opening our frame picture on programs such as GIMP, PAINT or even websites like pixilart [2]. Then, all we have to do is decide if the x,y coordinates are in the formed areas and make sure the multiplexer conveys the correct signal to RGB pins of the screen.

G. Working System

If the all the modules of the projects are created and correctly wired with each other, project should work. A screenshot of the working program is shown [Figure 16] in below:

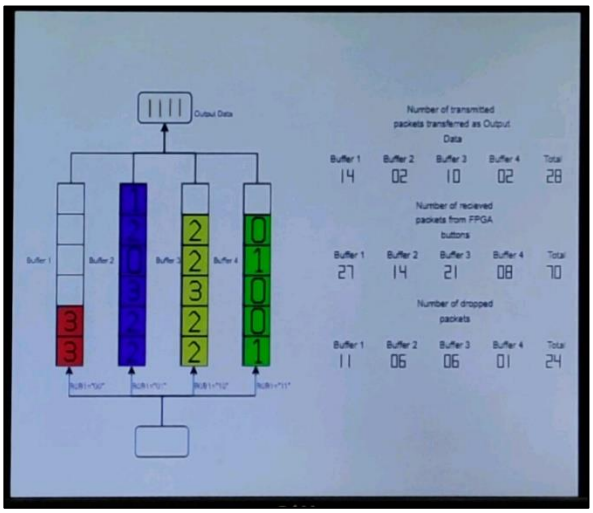

Figure 16. A random screenshot of the project.

IV. CONCLUSION

To conclude, we made our simple Quality of Service (QoS) design with all its requirements. We have three input buttons and a program to drive VGA interface to screen outputs. Our algorithm correctly meets the latency and reliability requirements. Every moment of the processes can be observed accurately on the monitor. Data shifts and colours in the buffers look fine. The number of received packets, the number of transmitted packets and the number of dropped packets can be seen simultaneously for each buffer.

REFERENCES

[1] Embedded Thoughts. (2016, December 30). *Driving a VGA monitor using an FPGA*. Embedded Thoughts. Retrieved July 6, 2022, from https://embeddedthoughts.com/2016/07/29/driving-a-vga-monitor-using-an-fpga/

[2] *Beginning FPGA graphics*. Project F: FPGA Dev. (2022, June 13). Retrieved July 6, 2022, from https://projectf.io/posts/fpga-graphics/

[3] Montvydas, & Instructables. (2017, October 4). *Image from FPGA to VGA*. Instructables. Retrieved July 6, 2022, from https://www.instructables.com/Image-from-FPGA-to-VGA/