Automated 4-Way-Road Cross Section

1st Vytaras Juraska
Electronic Engineering
Hochschule Hamm-Lippstadt
Lippstadt, Germany
vytaras.juraska@stud.hshl.de

2nd Gordan Konevski
Electronic Engineering
Hochschule Hamm-Lippstadt
Lippstadt, Germany
gordan.konevski@stud.hshl.de

3rd Giuseppe Scalora
Electronic Engineering
Hochschule Hamm-Lippstadt
Lippstadt, Germany
giuseppe.scalora@stud.hshl.de

4th Adam Sulak
Electronic Engineering
Hochschule Hamm-Lippstadt
Lippstadt, Germany
adam.sulak@stud.hshl.de

***Abstract*—The focus of this paper will describe the procedural steps presented in order to develop a fully automated control system for a 4 way cross-road section. Firstly the motivation and the problem will be discussed with a fair analysis on what is there to be solved. Afterwards all the requirements that need to be met, will be listed and described accordingly. The implementation part will be split into two separate sections in order to differentiate the hardware implementation from the software implementation, both respectively realized with the use of 2 different means, ModelSIM and the FreeRTOS kernel, including all its libraries.**

I. Introduction

Road transport automation has the sole purpose of contributing to the key objectives of the EU transport policies, mostly to increase safety, efficiency of traffic, avoidance of traffic congestion on intersections and improve the time frames necessary to do all of this. Beside these formal aspects, some social aspects are covered, for example the improvements in comfort from the users end including elderly people and impaired people. The automation discussed in this context encapsulates the applications of autonomous driving and interactions/interfacing with an intelligent environment. This last concept must take care of vehicle to vehicle communication and vehicle to infrastructure communication. Many factors can also be taken into consideration when working within the infrastructure context, such as like traffic controllers, traffic lights, pedestrian crossing sections, which all together make for a perfect real environment. Many applications of autonomous driving are already on the road but when it comes to connect this to infrastructure interactions, the technological advancement has still a lot of work to do, despite that, it is commonly accepted that automation in this field will become a reality sooner or later and that it will play a key role in future transport systems.

II. Motivation

The purpose defined for this scope deals with the search and development of a solution to manage the traffic of autonomous cars inside a crossroad-section. It is important to consider the behaviour of these autonomous vehicles in any considered scenario and model the reactions and most efficient ways to deal with the queuing and scheduling of each vehicle in order to accept them by the environment and fluently guide them out of the cross-section in complete safety. This project study will assume that the cross-section handled starts from a well defined distance and that individual sensors, for the autonomous driving of each car, are already handled by the manufacturers.

III. Analysis of the problem

In the problem analysis it is important to define what exactly is there to solve and develop. In order to understand this, it is helpful to take a look at how a real life cross-section behaves and looks like. The main concern is how to accept and register cars that wants to enter the system and how to lead them safely to the exit of the section. For this purpose a queuing system has to be implemented, taking care of handling stacking cars entering the cross-section afterwards, to dispatch every single car, a collision avoidance system must be developed which will help to not let the car ”crash” nor overdrive on some other car‘s path. In the end every single car has to be dispatched and assigned a priority according to their own arrival time, therefore a scheduling algorithm has to be included.

IV. Requirements and modeling

Before any other precedents are set, it is crucial to define and delineate the specific requirements that the project must meet, alongside with their respective modeling, which will be divided in three parts:

- Environmental Requirements
- Technical Requirements
- Vehicular Requirements

A. Environmental Requirements

The environment is set to be ideal, which will eliminate a lot of inevitable errors, but an effort will still be made into making this project compatible with the real world, at least conceptually. The environment would essentially be composed of a single crossroad in which the scenarios would take place. The elements of the crossroad are as follows: There are four separate bidirectional roads (each with a single detectable lane) which all meet in the middle. Each road will be tagged as either North, West, South or East and it would be done so liberally, meaning that the position of the roads do not have to point toward, for example, true North. That would mean that every other following crossroad would also be marked as such, although our scenarios will only be contained in a single one. Cutting through the roads will be vertical lines that would be used as indicators, or just simply markers, that would mark the distance from the line - to the center. There are three such

lines, each with their own color code (blue, red and black) and the full details of their nature are described below. Figure 1, shows how the cross-road looks like in our specific case.

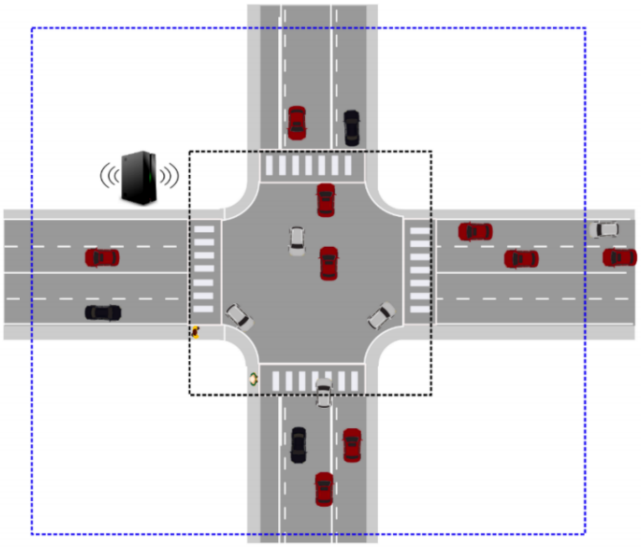

Fig. 1. Visual representation of the cross-road

B. Technical Requirements

The main technology behind the project will be a ”communication box”, although its usage will be described further below. The following metrics have to be met:

- each road is at least 200m long
- there is 150m distance between the blue line and the black line
- however, there is 75m between the blue line and the red line
- there must be a buffer distance of 5 meters from the red line
- the scheduling time constraint is limited to 5 seconds
- each car would have their speed limited to 50 km/h, fixed, after passing the blue line

C. Vehicular Requirements

The vehicles are expected to run in quite a decentralized fashion. Since the environment is set to be ideal, we presuppose that there will be universal standards put in place. The vehicles would use the communication box as a secondary source of information, while the road as a primary one, and the standards are there to define how exactly the vehicles would interpret certain data points. But what makes the vehicles particularly ”decentralized” is the fact that the decision making process is left to them, i.e. the communication box cannot interfere with the decision making process.

V. Initial Communication

The initial communication between the vehicles and the communication box is established the moment the vehicle detects the initial vertical line (blue line). The vehicle then

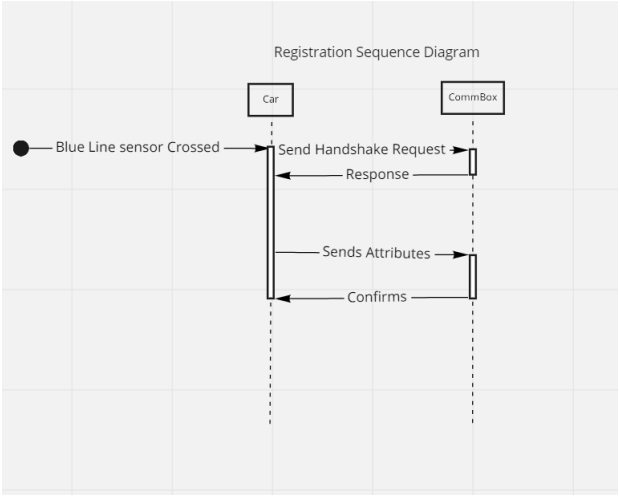

Fig. 2. Sequence Diagram

detects the communication box trough a Fire and Forget UDP signal, after which it sends a handshake request and awaits a response. Once the vehicle receives a response and establishes a successful first connection, it proceeds to send information about its attributes:

- current speed
- destination (nothing too data heavy, it would be as simple as choosing the Northern or Southern road, for example)
- size of the vehicle (for safety, collision-detection purposes)

Once the attributes have been received by the communication box, the vehicle proceeds by executing on further information sent by the connection requests that follow after, specifically: how the speed of the vehicle needs to be configured (either lowered or sped-up), in which direction the vehicles is allowed to drive and any information necessary to avoid collision with other vehicles. These instructions are memorized and put in/taken from the heap of the communication boxes onboard memory and scheduler. The scheduler allows for the simultenious communication with more than one vehicle at one time.

VI. Scheduler

The scheduler is best described with these diagrams:

VII. Implementation

A. ModelSIM Implementation

1) Concept Definition: Collision avoidance is a crucial part of this project, the most feasible way that will be presented is simply the implementation and creation of a hardware interface which will deal with input signals sent by the cars and output signals that will be handled in order to check whether there is a collision or not. With the help of VHDL the concept is simply put into code, the idea being to have cars sending a combination of 4-bit binary which will vary according to the direction (Figure 6).

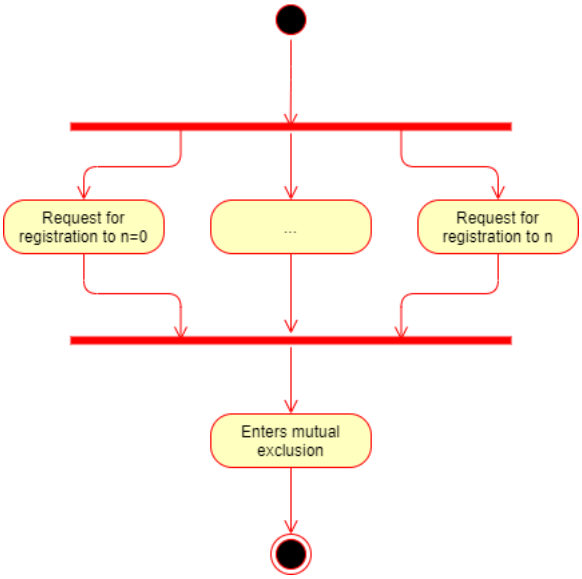

Fig. 3. Data Collection Diagram

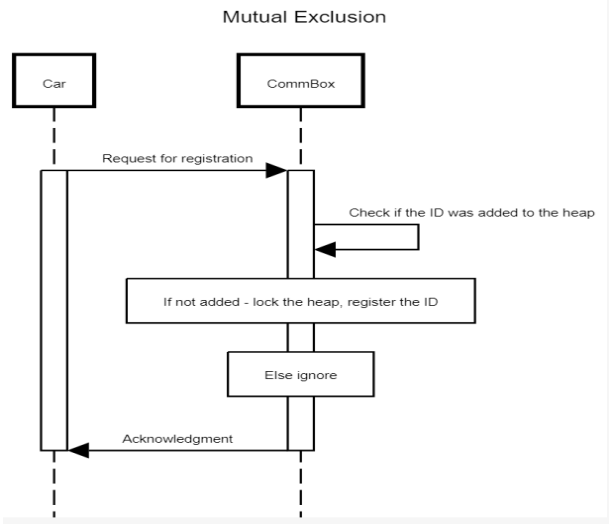

Fig. 4. Mutex Diagram

For our specific purpose we used the conventional cardinal directions, hence, if a car comes from north and has to go south the direction will be assessed as NS and its respective input sent will be ”0000”, shown next is the complete table with the corresponding directions assigned to the 4-bit combination.

This 4-bit combination will be handled by the algorithm developed in VHDL model sim and compared to another car‘s direction in order to check for the collision. The table below will show all the possible collisions.

Taking this specific concept of collision we noticed that we can simplify the output answers greatly, theoretically we could simplify the answers in three separate output blocks, but in order to have a much more simpler solution to VHDL, we just imagined the whole graph to be mirrored diagonally.

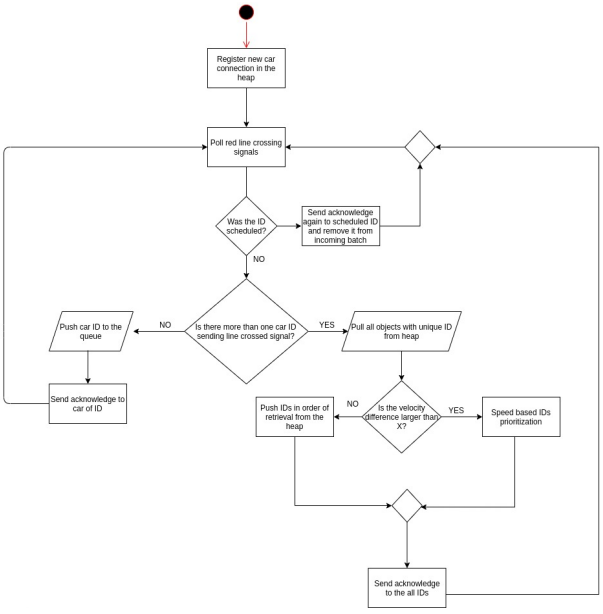

Fig. 5. Activity Diagram for the Scheduler

N	W	S	E	
0	0	0	0	NW
0	0	0	1	NS
0	0	1	0	NE
0	0	1	1	WS
0	1	0	0	WE
0	1	0	1	WN
0	1	1	0	SE
0	1	1	1	SN
1	0	0	0	SW
1	0	0	1	EN
1	0	1	0	EW
1	0	1	1	ES
1	1	0	0	-
1	1	0	1	-
1	1	1	0	-
1	1	1	1	-

Fig. 6. cardinal directions table

2) Code Implementation: When considering the required IO for our system, since the directions of each car will be represented by 4 bit system - we implemented 8 inputs in total, 4 bit directional representations for the two cars, which collision has to be checked. In terms of an output, we considered the most optimal solution is to have a single output, which would represent weather the collision is present or not.

Later on, we define all of the cardinal directions as signals, dedicating the 4 bit combination of each car to the direction representation. Hence, the original 4 bit input is just to represent the 12 possible directions, from which a car could come from and go to. Since we define that for 2 separate cars, each of them have a unique representation and so overall we have 24 directional signals defined.

Finally, we represent the logic of the collision table to the VHDL by expressing solutions, when the main output is considered to be equals to one. For instance, taking a look at the Figure 7 we see, that if the first car comes from west to south and the second car is coming from north to south, then our collision, in this term our output is one. This is done with

	NW	NS	NE	WS	WE	WN	SE	SN	SW	EN	EW	ES
NW				0	0	0	0	0	1	0	1	0
NS		X		1	1	1	0	0	1	0	1	1
NE				0	1	1	1	1	0	0	1	1
WS	0	1	0				0	0	0	0	0	1
WE	0	1	1		X		1	1	1	0	0	1
WN	0	1	1				0	1	1	1	1	0
SE	0	0	1	0	1	0				0	0	0
SN	0	0	1	0	1	1		X		1	1	1
SW	1	1	0	0	1	1				0	1	1
EN	0	0	0	0	0	1	0	1	0			
EW	1	1	1	0	0	1	0	1	1		X	
ES	0	1	1	1	1	0	0	1	1			

Fig. 7. collision table with relative outputs

a all of the cases, where the collision is considered to happen and as referred before, since the graph is repetitive, only half of the solutions have to be implemented. Hence having this defined concludes the code implementation of VHDL

B. FreeRTOS Implementation

1) The choice of FreeRTOS: When it comes to software implementation of this project, the most feasible decision was to use the open source FreeRTOS kernel and all its respective libraries. The kernel is mainly C based and it provides many useful and interesting functionalities linked to the implementation and development of real time systems under the software aspect. In our implementation we made use of the FreeRTOS to create our own scheduler which handles two separate Queues of entities. The entities which belong to our system are the cars which enters the system at any not given point in time. The idea is to register the cars as soon as they cross the first line present in the cross section, done so, the cars will be pushed into what we called "Unscheduled Queue". Since the cross section has only 4 lanes with a single line per direction, hence the highest possible number of cars which have to be registered at the same time is 4. The last mentioned case is also assumed to be a worst case scenario since in real life it could be rare to have such situation happening. To solve this problem we created a buffer which is making sure that within a certain time span, 50 ms was set for our scenario, the cars entering the system will be buffered accordingly and sorted into the unscheduled queue. After the scheduler performs some conditions checks in the queues it can finally decide to push the cars into the scheduled queue. At this point it will be needed to check whether the cars are free to go or a collision will happen with a different car in the same system. The detailed explanation of how the scheduling and queues work is handled in the next paragraphs.

2) Scheduling and queues: As already mentioned, two different queues are used in our software. In sequential order, the cars will be first handled and pushed into a buffer which will make sure we can register cars even with a small notice time. This means that if multiple cars cross the line at almost the same time point they will still be sorted inside the queue according to the registration time. For the case to be handled, we came up with the conclusion of using a simple FIFO algorithm, first in, first out, to get rid of the cars according to their arrival time. Now let us take into account a simple scenario in which two cars reach the initial line at t_1 = 12.4 s and t_2 = 12.7 s. In real life the difference will be almost unnoticeable but in a real time system the difference will be highly relevant, therefore setting a delay for the polling time equals to t_p = 50 ms, makes sure that we can register cars that enter the system as long as the time difference is not smaller than 50 ms. The buffer can only contain maximum 4 cars (worst case scenario), which leads to a maximum buffer filling time of t_p^{MAX} = 200 ms, and once it gets full, or the cars will pass through the second line, the information received will be pushed to the unscheduled queue and the buffer will be emptied and reset. From this point, 3 possible cases can happen:

1) The cars will be assumed to need a certain amount of time before reaching the second line, hence, once they reach the line, the buffer will be emptied automatically and the cars moved to the scheduled queue.
2) The buffer has been filled even before the first car registered crossed the second line, therefore the cars will be automatically pushed into the scheduled queue after a short time delay and the unscheduled queue will be emptied.
3) A 5th car enters the system before the unscheduled queue is emptied, therefore the cars will be preemptively pushed into the scheduled queue and the unscheduled queue will be emptied to make space for the new cars registering, with a short time delay.

By handling these 2 possible cases, we make sure that there will always be space for registering new cars approaching and, if no major accidents happen, the scheduler will keep doing its job periodically. A snippet of code will be shown below in order to understand how the algorithm formally works.

```
19  void  vSchedulerTask(void *pvParameters) {
20
21      Car_t xReadBuffer[4];
22      BaseType_t xStatus;
23
24      for( ;; ) {
25          xStatus = xQueueReceive(xUnscheduledCarsQueue, &xReadBuffer[0], portMAX_DELAY);
26
27          int buffer_fill = 1;
28          for (int i = 1; i < 4; i++) {
29              vTaskDelay(pdMS_TO_TICKS(50));
30
31              if (uxQueueMessagesWaiting(xUnscheduledCarsQueue) == 0) {
32                  break;
33              }
34              else {
35                  xStatus = xQueueReceive(xUnscheduledCarsQueue, &xReadBuffer[i], 0);
36                  buffer_fill++;
37              }
38          }
39
40          if (buffer_fill == 1) {
41              console_print("Single car received, pushed immediately...\r\n");
42              xReadBuffer[0].is_scheduled = true;
43              xStatus = xQueueSendToBack( xScheduledCarsQueue, &xReadBuffer[0], 0);
44          }
45          else {
46              qsort(xReadBuffer, buffer_fill, sizeof(Car_t), car_comparator);
47
48              for (int i = 0; i < buffer_fill; i++) {
49                  console_print("Car %d scheduled\r\n", xReadBuffer[i].id);
50                  xReadBuffer[i].is_scheduled = true;
51                  xStatus = xQueueSendToBack( xScheduledCarsQueue, &xReadBuffer[i], 0);
52              }
53              console_print("Many cars received, all pushed...\r\n");
54          }
55      }
```

Fig. 8. Scheduler handling section of code

3) Resolver and red line crossing tasks: Some extra tasks that helped the realization of the software, are respectively the

resolver and red line crossing task. The red line crossing task is simply a task method which generates random cars to be added and registered within the system. They will be assigned a random ID, direction, poll time and a boolean value on a variable called ”is-scheduled” to check whether the car has been already scheduled or not. The working principle is fairly easy to understand and it is shown below:

```
9   extern QueueHandle_t xUnscheduledCarsQueue;
10
11  void  vRedlineCrossedTask(void *pvParameters) {
12
13      BaseType_t xStatus;
14      TickType_t delay = ((rc_settings_t*)pvParameters)->delay;
15      int from_lane = ((rc_settings_t*)pvParameters)->from_lane;
16
17      for ( ;; ) {
18          volatile TickType_t call_time = pdMS_TO_TICKS(xTaskGetTickCount());
19          Car_t car = {
20              .id = (rand() % 10000) + 1,
21              .direction =  3*from_lane + (rand() % 3),
22              .poll_time = call_time,
23              .is_scheduled = false
24          };
25          xStatus = xQueueSendToBack( xUnscheduledCarsQueue, &car, portMAX_DELAY);
26          vTaskDelay(pdMS_TO_TICKS(delay));
27      }
28  }
```

Fig. 9. Simulating and generating car tasks at the redline

Instead the resolver task handles the presumed collision of cars, since the whole project has been done virtually without the use of any physical hardware beside the PC, one of the main impediments was to create an interface between the hardware logic implemented in VHDL and the freeRTOS. The collision handled via software is simply comparing the polling times between cars entering the system at relatively close time and if the time is smaller than a given value then one of the two cars will be slowed down to avoid the collision. In a future implementation we would like to actually create our own interface to send the data from VHDL to freeRTOS via hardware and simulating the collisions handling properly. Below is a screenshot of the code part that handles the collisions [1] [2].

```
12
13  void vResolverTask(void *pvParameters) {
14
15      Car_t xReadBuffer[2];
16      BaseType_t xStatus;
17      bool last_stoped = false;
18
19      for ( ;; ) {
20          if (last_stoped) {
21              xReadBuffer[0] = xReadBuffer[1];
22              last_stoped = false;
23          }
24          else {
25              xStatus = xQueueReceive(xScheduledCarsQueue, &xReadBuffer[0], portMAX_DELAY);
26          }
27
28          xStatus = xQueueReceive(xScheduledCarsQueue, &xReadBuffer[1], pdMS_TO_TICKS(50));
29          if (xStatus == pdFALSE) {
30              console_print("Car id: %d let through\n", xReadBuffer[0].id);
31          }
32          else {
33              const int ms_threshold = 300;
34              if (abs(xReadBuffer[0].poll_time - xReadBuffer[1].poll_time) < ms_threshold) {
35                  console_print("Car id: %d let through\n", xReadBuffer[0].id);
36                  console_print("Car id: %d stoped\n", xReadBuffer[1].id);
37                  last_stoped = true;
38              }
39              else {
40                  console_print(
41                      "Car id: %d and car id: %d let through\n",
42                      xReadBuffer[0].id, xReadBuffer[1].id
43                      );
44              }
45          }
46      }
47  }
```

Fig. 10. Resolver task handling the collisions

VIII. Simulation Results

A. ModelSIM results

Considering the results of our already written VHDL representation of the collision checking, which we in the end decided to call - Traffic Controller, we noticed, that simulating the code itself, the answers were corresponding to the collision table and testing the inputs one by one, we saw that the code was working as it intended, but to represent all of the inputs correspondingly and truly showing, that the solution works, was not as easy we initially thought.

To really prove, that all of the collisions were realised in our system, we had to create a test-bench, which worked as follows: since single cars direction is represented with 4 unique inputs, one set of inputs were considered as a fixed value, while other set of inputs was running through all of the possible outcomes. After doing that we increase the value of the fixed set by one and run all of the possible outcomes with the other set again. This has been done throughout all of the values, which we use as directions, which means, that the combinations ”1100”, ”1101”, ”1110”, ”1111” were not considered, but since the system is designed to output a 0 at all of the other undefined outputs, the outcome would just be 0 at the output through these values. Essentially we simulated all of the possible outcomes, shown in the Table 7, even considering the areas marked as ”X”.

The exact simulation results of this described test-bench are shown in the Figure 11. Inputs from ’a’ to ’d’ in alphabetical order represent the change of the fixed sets values, when the inputs from ’e’ to ’h’ also in alphabetical order represent changing set of values. Variable ’x’ is the output, which represents collision status, if it is equal to one - the collision is detected.

B. Software FreeRTOS results

For the FreeRTOS code simulation, the whole team agreed on using Visual Studio Code as a mean of compiling and executing the software. The libraries belonging to the FreeRTOS Kernel have been added and stored in the submodule of our GitHub page. The simulation ran on the software, aims to show how all the tasks together work accordingly, inside a main.c file which contains all the external defined methods, and shows what happens when the system is initialized, hence the first tasks of nature ”car” are generated and therefore reach the system. In this specific example 4 cars are randomly created and then pushed to the scheduled queue, through the buffer modeled in the implementation, in real time there are also occurrencies of other cars being generated which will properly be handled by our scheduler in the correct arrival order. Another important detail which can be seen from the results is the allowance of cars through the system or the inability to cross because there is a risk of collision. As you can see from the figure, when 4 cars have been received the buffer will automatically be refreshed to make space for other incoming cars. Until the very end of the code it is noticeable how the tasks are handled periodically and nicely interrupted if

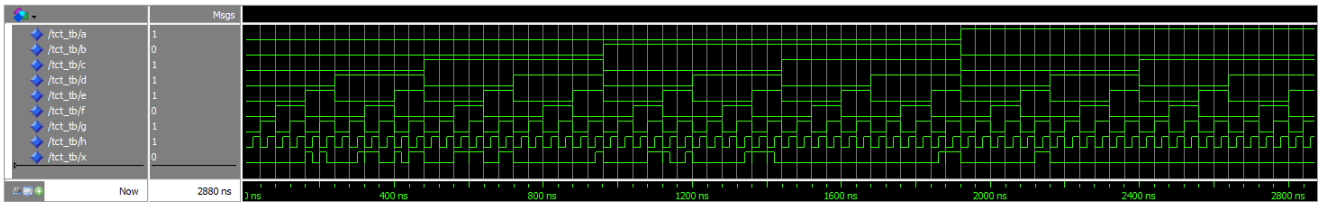

Fig. 11. Waveform of all of the possible outcomes of Traffic Controller

another task of nature ”car” is generated. The actual terminal output is shown in Figure 12.

```
adam@y700:~/Programs/RT/H_S_codesign$ ./build/codesign
Car 9536 scheduled
Car 7304 scheduled
Car 2056 scheduled
Car 9261 scheduled
Many cars received, all pushed...
Car id: 9536 let through
Car id: 7304 stoped
Car id: 7304 let through
Car id: 2056 stoped
Car id: 2056 let through
Car id: 9261 stoped
Car id: 9261 let through
Single car received, pushed immediately...
Car id: 7358 let through
Single car received, pushed immediately...
Car id: 814 let through
Car 1179 scheduled
Car 2702 scheduled
Many cars received, all pushed...
Car id: 1179 let through
Car id: 2702 stoped
Car id: 2702 let through
Single car received, pushed immediately...
Car id: 7592 let through
Single car received, pushed immediately...
Car id: 1178 let through
Car 4653 scheduled
Car 4035 scheduled
Many cars received, all pushed...
Car id: 4653 let through
Car id: 4035 stoped
Car id: 4035 let through
```

Fig. 12. A running simulation of the code

IX. Conclusion

In this paper we discussed our theoretical approach towards the problem of automating a four way cross-road section and showed our solution for the implementation via the use of ModelSIM, in terms of hardware handling and FreeRTOS Kernel related to the software management and development. The results obtained fulfilled the requirements defined in the beginning of the whole project while leaving some room for further improvements and future implementations.

References

[1] Barry, R. ”Mastering the FreeRTOSTM Real Time Kernel, A Hands-On Tutorial Guide.” sl]: Real Time Engineers Ltd., 2016c. Resource Management (2016): 233-264.

[2] FreeRTOS - Market leading RTOS (Real Time Operating System) for embedded systems with Internet of Things extensions. (2021, May 12). FreeRTOS. https://www.freertos.org/

X. Affidavit

I, Giuseppe Scalora alongside with Vytaras Juraska, Adam Sulan and Gordan Konevski, herewith declare that we have composed the present paper and work by ourself and without use of any other than the cited sources and aids. Sentences or parts of sentences quoted literally are marked as such; other references with regard to the statement and scope are indicated by full details of the publications concerned. The paper and work in the same or similar form has not been submitted to any examination body and has not been published. This paper was not yet, even in part, used in another examination or as a course performance.