COMP9032

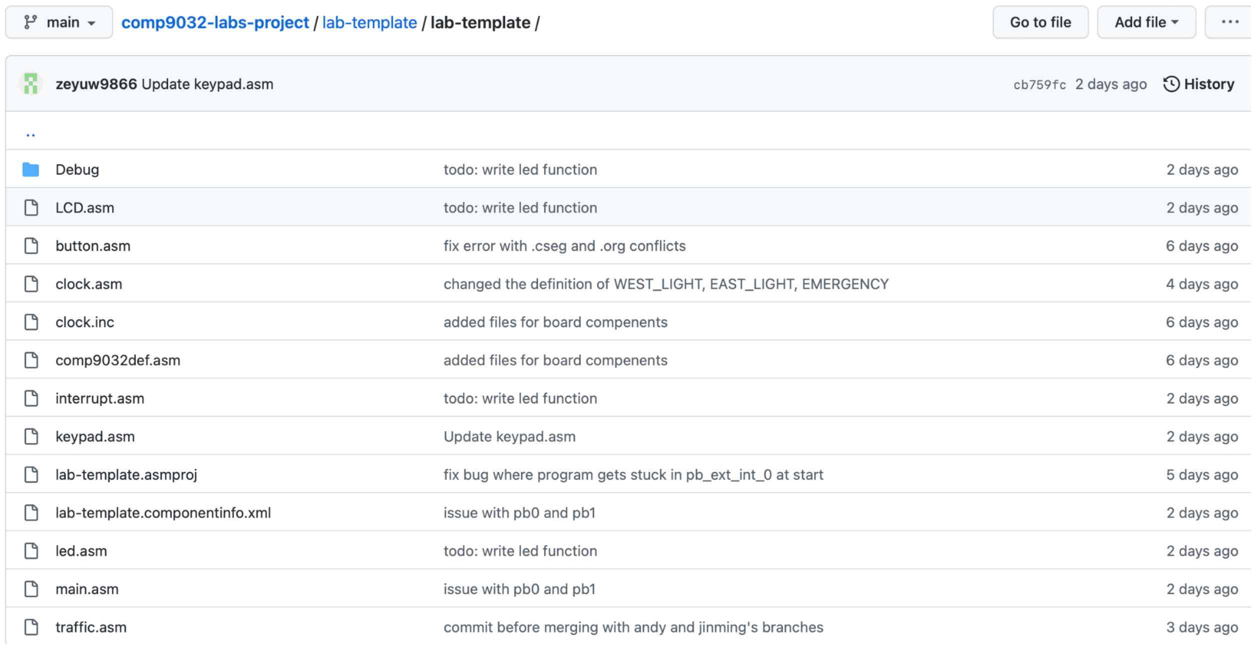
# COMP9032 Project

Oscar Feng, Haozheng Sun, Jinming Liu, Li-Hung Kuo, Zeyu Wang, Haifeng Li

## 1. Introduction

The project requires us to simulate a traffic system via AVR lab board. We need to use keypad, LCD, LED, PB0/PB1, Timer. In order to improve the readability of the code and to facilitate the division of labor, each external device and the corresponding interrupt function was developed by a different person, while the traffic system logic was mainly designed by Oscar and Haozheng. At the same time, we used github for collaborative development, with each person creating a branch dedicated to the corresponding module. Finally, Oscar and Haozheng did the code merging and function calling.

According to the task allocation and function division, we set up the project files as is shown below.


main
comp9032-labs-project / lab-template / lab-template /
Go to file
Add file
zeyuw9866 Update keypad.asm
cb759fc 2 days ago
History

| | | |
|---|---|---|
| .. | | |
| Debug | todo: write led function | 2 days ago |
| LCD.asm | todo: write led function | 2 days ago |
| button.asm | fix error with .cseg and .org conflicts | 6 days ago |
| clock.asm | changed the definition of WEST_LIGHT, EAST_LIGHT, EMERGENCY | 4 days ago |
| clock.inc | added files for board compenents | 6 days ago |
| comp9032def.asm | added files for board compenents | 6 days ago |
| interrupt.asm | todo: write led function | 2 days ago |
| keypad.asm | Update keypad.asm | 2 days ago |
| lab-template.asmproj | fix bug where program gets stuck in pb_ext_int_0 at start | 5 days ago |
| lab-template.componentinfo.xml | issue with pb0 and pb1 | 2 days ago |
| led.asm | todo: write led function | 2 days ago |
| main.asm | issue with pb0 and pb1 | 2 days ago |
| traffic.asm | commit before merging with andy and jinming's branches | 3 days ago |

## 2. Program Structure

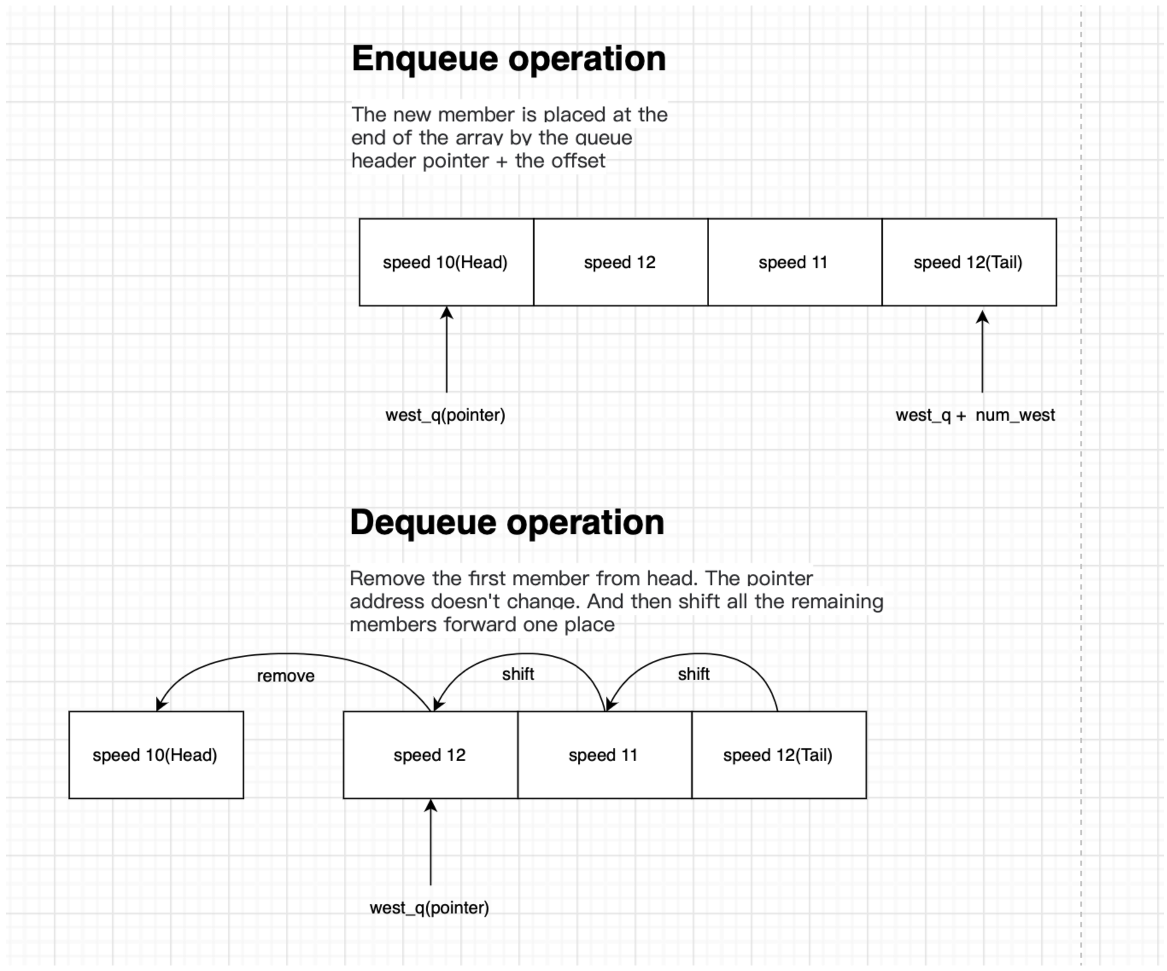
The simulation of the traffic flow on the road is represented by a combination of predefined array variables and functions. The road, along with the lines of cars that may form on the west and east ends, are represented in memory by arrays. Each array element represents a car and holds a number: the speed of the car. In addition to the arrays of car speeds, we also have an array that keeps track of the positions of each car on the road.

## 2.1 Enqueue and Dequeue

For the arrays, we've written *enqueue* and *dequeue* operations to facilitate easier insertion and deletion of elements and to make the arrays behave like queues (Last In, First Out).[1]

When a car is detected and wishes to enter the road, we first insert it into its respective array on one of the sides of the road using the *enqueue* operation. When the traffic light for that side turns green and there are no more oncoming cars on the road, we *dequeue* it from that side and *enqueue* it onto the road.

[1] *A standard implementation of an array-based queue data structure might have a* head *and* tail *pointer. However, we have elected to include only the bare-minimum number of additional data variables needed to implement a queue. Namely, we only need the maximum capacity of the array and the current size of the array. Such a queue implementation may not be very efficient in its* dequeue *operation, but it does the job for the purposes of this project.*


Enqueue operation
The new member is placed at the end of the array by the queue header pointer + the offset
speed 10(Head)
speed 12
speed 11
speed 12(Tail)
west_q(pointer)
west_q + num_west
Dequeue operation
Remove the first member from head. The pointer address doesn't change. And then shift all the remaining members forward one place
remove
shift
shift
speed 10(Head)
speed 12
speed 11
speed 12(Tail)
west_q(pointer)

## 2.2 Road Length and Car Speed Scaling

In the original assignment specification, the length of the road was defined to be 50 m. The slowest car speed, 10 km/h, is equivalent to approximately 2.77 m/s. Since it is easier to store and manipulate integers, we have elected to multiply both the length of the road and the speed of the car (in m/s) by a factor of 5. Thus in our simulation, the road is 250 “units” long, and the 10 km/h car travels at a speed of 13.889 ≈ 14 units/s. Cars enter the road with an initial position of 0, and are deemed to have left the road when their position is greater than 250, or if the carry flag is set after updating its position.

## 2.3 Data Definitions

Because of dividing the whole program in several parts. We use Data Space more rather than registers. The registers we use are mostly used to store temp values in functions and main loops.

We can divide the data into 3 parts :status data,counter data and queue data.

### 2.3.1 Status data:

*traffic_flow* : Traffic flow shows the status of the traffic. It can be 1,2 and 4 as decimal.

0b0000 0001: West light on; The traffic is eastbound.

0b0000 0010: East light on; The traffic is westbound.

0b0000 0100: The traffic is in an emergency.

*delayintrans* : delayintrans shows the status after reverse the traffic flow, if there are cars on the road.It can be 0 and 1.

0b0000 0001: There are cars on the road. Cannot let cars in.

0b0000 0000: There are no cars on the road. Can let cars in.

### 2.3.2 Counter data:

*temporary_counter:* Incremented every time there is an interrupt from Timer0. Once *temporary_counter* equals 250, then 1 second has passed.

*three_sec_counter:* Incremented every time 1 second has passed. Once three_sec_counter reaches 3, then we call the function *let_car_in*.

*three_min_counter:* Incremented every time 1 second has passed. Once three_min_counter reaches 180, then we call the function *reverse_traffic_flow*.

### 2.3.3 Queue data:

*west_q* :  An array of size *MAX_QUEUE_SIZE*. Contains the speeds of the cars lined up in the west end.

*east_q* : An array of size *MAX_QUEUE_SIZE*. Contains the speeds of the cars lined up in the west end.

*passing_pos* : An array of size *MAX_QUEUE_SIZE*. Contains the positions of the cars currently on the road.

*passing_speed* : An array of size *MAX_QUEUE_SIZE.* Contains the speeds of the cars currently on the road

# 3. Simulation

Every second, the system updates the status of the cars on the road. First, we iterate through the cars on the road and add their speeds to their current position. If their updated position is greater than or equal to the position of the car before them in the road/queue, then we detect that as a collision. If no collisions were detected, then we dequeue the cars that have left the road.

## 3.1 Hardware Timer

The program has an internal timer which dictates the rate at which all traffic flow events happen. The internal timer uses the 8-bit hardware counter *Timer0*. We use a *prescaler value of 256* to achieve a clock period of (256 / 1.6e6) = 0.016 ms or 16 µs.

One complete cycle through the hardware counter takes 256 * 0.016 = 4.096 ms. Thus, 250 interrupts from the timer represents the duration of approximately 1 second (1024 ms, to be exact).

## 3.2 Simulation Timer

The interrupt service routine *simulation_timer_isr*, located in *interrupt.asm*, counts the number of interrupts from *Timer0* and increments a number of counters defined in the data section.

# 4. Inputs

## 4.1 Push Button

1) Hardware setting in *button.asm* : The PIN0 and PIN1 of PORTD, get inputs of PB0 and PB1.

2) Software: *interrupt.asm*:

create functions: *pb_int0_isr, pb_int1_isr*

used variables: *num_west, num_east, traffic_flow, num_passing*

The interrupt service routine  pb_int0_isr and  pb_int1_isr, located in *interrupt.asm*. Once it is detected that the buttons PB0 and PB1 are pressed as sensors that the car wants to pass the road, then we push this car into the queue and wait to released on the road.

Because when the button is pressed, it bounces several times due to physics, so we count how many times it reflects and treat this number as one.

If Pb0 is pressed, we put the car into west_q; otherwise, if Pb1 is pressed, we put the car into east_q.  When we push the car into the queue, we keep tracking the number of cars already in the queue to ensure that if the queue is full we won't let the car in, then increment the queue size to fit the car in.

## 4.2 Keypad

1) Hardware setting: PORT L for get inputs

2) keypad.asm programming:

Create functions:KEYPAD_RESET

Used variables :temp0,temp1

Logic description:To determine exactly which button was pressed, columns are checked sequentially from column0 to column3. When a pressed button is detected in a column, it is checked from row0 to row3 in order to determine the exact location. Afterwards, the buttons are given different values and meanings depending on the number of rows of buttons identified. For example, pressing button 1 will return the number 14 to the main function, and as only buttons 1-6 and * are used in this simulation, the function will return the number 0 if the current position of the button is not within this range, thus disabling the rest of the buttons.

## 5. Outputs

### 5.1 LED

1) Hardware setting: PORT C

2) Software Programming: led.asm

create functions: *led_reset, led_simulation*

used variables: *traffic_flow*

logic description: The *led_reset* function is called in the main.asm file to reset the LED at the beginning. The *led_simulation* function is called once per second via the overflow interrupt of the timer. It will load the value of *traffic_flow* and then output the corresponding pattern.

### 5.2 LCD

1) Hardware setting: PORT F for data, PORT A for control

2) Software programming: LCD.asm

create functions: *lcd_reset, lcd_display*

used variables: *num_west, num_east, traffic_flow, num_passing*

The *lcd_reset* function is called in the main.asm file to reset the LCD at the beginning. And then there is a timer interrupt service routine which will call *lcd_display* every second to refresh the LCD. When it refreshes the LCD display, *lcd_display* gets the number of waiting vehicles at the end entry from *num_west* or *num_east* variables first and then converts it into character for output. Then it will print "<<" or ">>" based on the *traffic_flow* value. In addition, it gets the value of *num_passing* which is used to control the number of cycles that print a "=" sign.

## 6. Conclusion and Reflections

There were two behaviors of the program which ultimately did not meet the specifications:

1. The project specification states that when a car enters either side of the road, if the road is empty and the opposite side is empty, then the traffic light should automatically change to let the car in. In the program, the traffic light only changes every three minutes. However, this was merely an oversight due to the lack of time. We could have easily added this feature in the function *let_car_in*, where the program can simply check if the road and opposite queue are empty, and if so, change the traffic lights and let the car in.
2. We did not have time to fix the bouncing problem encountered with the push buttons. Normally, debouncing could be easily achieved by simply adding a small delay of around 150ms between each read of the PORT D pins. However, because the push buttons were configured as interrupts, simply adding a delay in the ISR did not work. It would appear that, even though global interrupts are automatically disabled before the start of an ISR, any interrupt that arrives in the middle of an ISR's execution would simply be queued in the system and would trigger once the current ISR finishes execution.

We implemented a partial solution to the bouncing issue at the time of the board demonstration which would work only some of the time. The idea was to introduce a counter which would be incremented every time the push button interrupt was triggered. Once the counter reached a certain value, then the actual push button function would be executed. However, a counter alone would not suffice; one needs to clear the counter within a predefined period of time (something in the range of 100-200 ms), because interrupts that are queued up can still increment the counter.