# Pelican Crossing Controller

EE1A2 – Microprocessor Systems & Digital Logic

Course lecturer: *Sandra I. Woolley*

**Student ID numbers:**

*Yousef Amar* 1095307

*Mohamad Sobhy* 1048258

# Contents

# Introduction

### *Project Overview and Summary*

This exercise involves the specification, design, implementation, verification and evaluation of a pedestrian crossing embedded system application similar, if not identical, to actual pelican crossings.

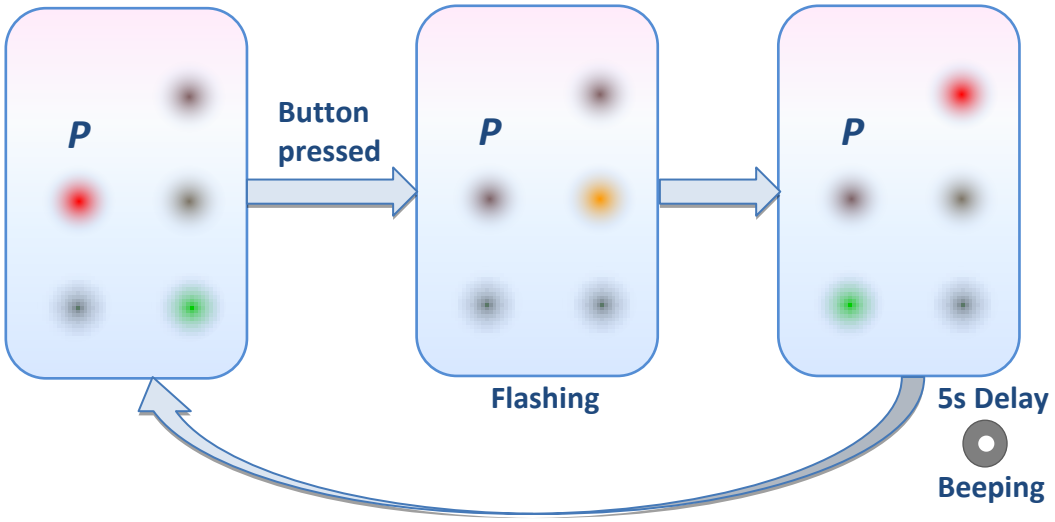
However, instead of continuously polling the pedestrian button like in the past exercise, this code should implement the use of an interrupt as this requires less processing power. There were no other constraints.

### *Contributors*

This project was handled by Yousef Amar (1095307) and Mohamad Sobhy (1048258) under the supervision of Sandra I. Woolley and the aid of her lectures.

# Controller Specification

The program should start with both the green traffic light and the red pedestrian light on simultaneously. It remains idle until a button is pressed which causes the light sequence in *fig. 1* as well as the attached video, to occur. The pedestrians are given 5 seconds to cross and a buzzer is set off (for further timing details see *Program Design*). It subsequently returns to the default output and remains in its idle state until allowing the button to once again be pressed.



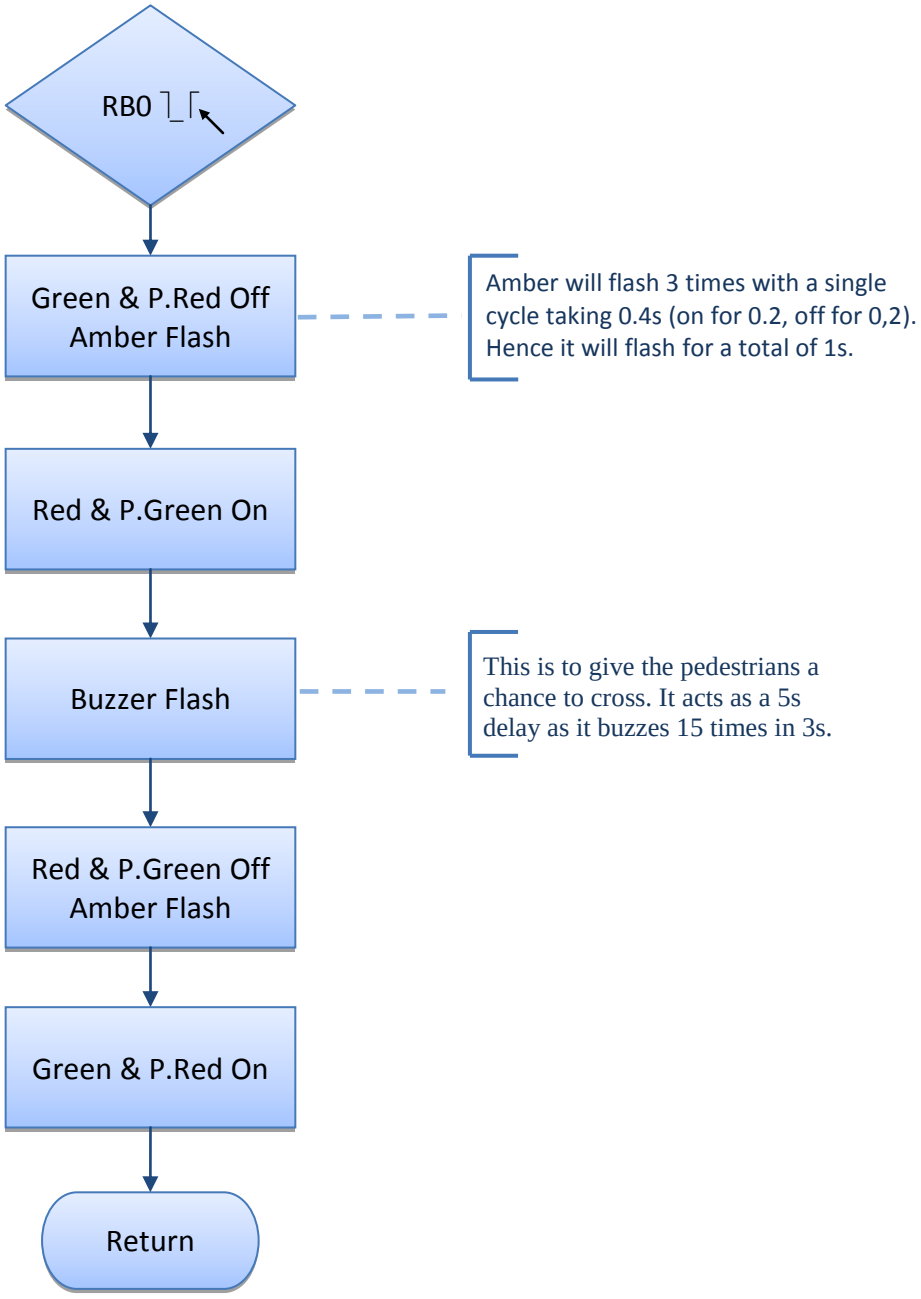

*[fig. 1]*

# Program Design

The pseudo code for the initial steps is as follows:

1. Turn green and ped. Red on.
2. Start and endless loop.

Of course one would have to enable the RB0 interrupt as well as do standard tasks such as configure the I/O bits, but the main idea is that once the (rising) edge-triggered interrupt is triggered by pressing the active-low button wired to RB0, the routine in *fig. 2* would commence.




*[fig. 2]*

# Verification

### *Test Specification*

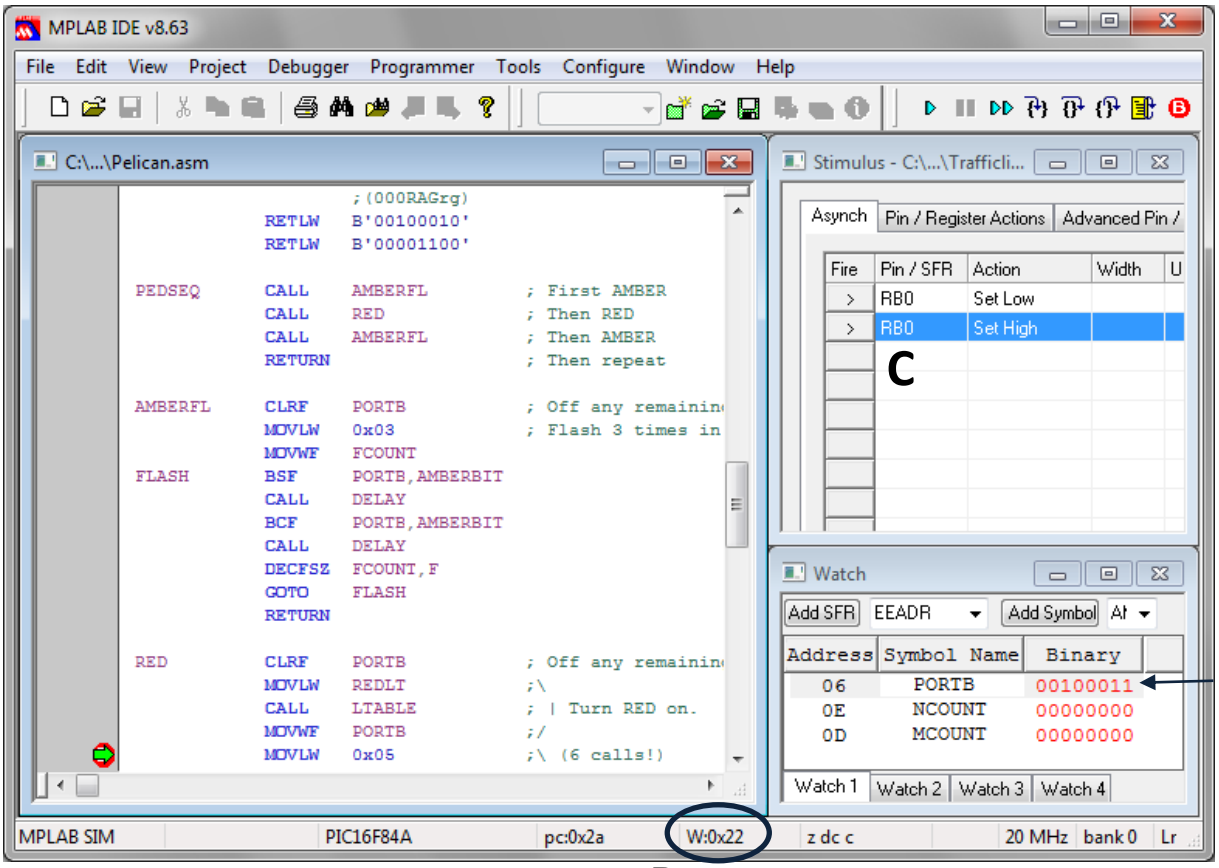
The tests to check whether the program was working as it was supposed to were simple. We needed to make sure of a number of things:

- The LEDs are in their default state upon powering up the microcontroller.
- The correct light sequence, with the correct timing is initiated upon releasing the button.
- The user cannot spam pressing the button and it would only once again be receptive to input when the sequence is over and the lights are back in their default state.

### *Functional Testing*

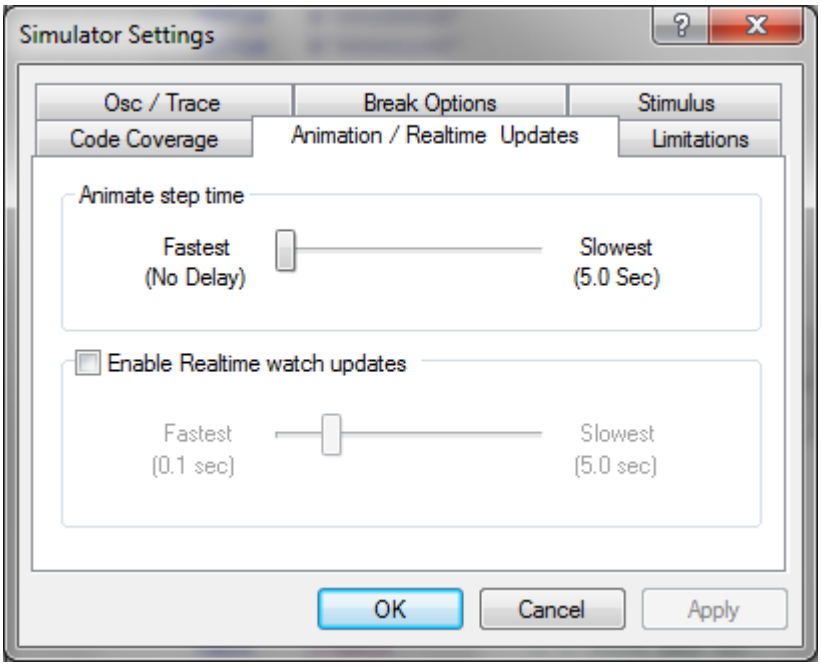
In order to make the testing process as quick and efficient as possible, we refrained from using the actual microcontroller until the end. Pinpointing bugs or errors became much easier with the use of the glitch-ridden, yet functional MPLAB SIM. Debugging became closer to what one would do with a high-level language; we could go through the code line by line, without having to worry about fried chips of faulty connections.

As can be seen in *fig. 3*, everything could be done without the use of hardware.




*[fig. 3]*

This allowed us to watch the I/O bits (**A**), the working register (**B**) and change the inputs all in real time (**C**) without the time consuming part. Furthermore we could change the speed at which the simulator steps through while it's running (*fig. 4*).




*[fig. 4]*

**Results and Conclusions**

The most major discrepancies we encountered did however prove that running a program on MPLAB SIM may simply not be enough. The most significant was the use of RA4 as an output. For lab to we had used RA0 as an input and all outputs were on PORTB exclusively. For lab 3 however, we needed to use RB0 to employ the interrupt. Out initial solution was to switch all outputs to PORTA and use RB0 as our input. The simulator made it seem like it would work perfectly fine. Only after the physical testing did we realize that RA4 cannot be used as an output. The way we dealt with this is to simply use PORTB exclusively.

Simple coding errors were easily debugged using the simulator yet there were some things we could not have been able to solve without it. For instance we would decrement the working register directly within our delay sub-routines (DECFSZ W,W) and we were convinced it would work. When it didn't we had to go through the code and watch the working register very carefully. It turned out, for some reason, that whenever it returns from a sub-routine, the working register would be set to 0xFF. One could overcome this by backing up W before calling a function but we opted to simply decrement a file register with the value W had in the beginning instead.

We thought of adding a delay to debounce the input button but the LED sequence itself acted as one since the interrupt flag is cleared at the end. Eventually, the program worked just as outlined earlier and met all specifications.

# Evaluation

As the PIC we were meant to use was pre-specified, we had the freedom to move within the boundaries of the resources the PIC16F84A provided. The following are the specs of our program:

| | |
|---|---|
| **Code Footprint:** | 65 bytes from the 1kB available (~6.35%) |
| **Call Depth:** | 5 levels call depth |
| **Register File Usage:** | 4 general-purpose registers used |
| **I/O Usage:** | Pins 0-6 of PORTB (8 bits) |

Clearly we could have been more frugal in terms of excessive nesting and redundant file registers yet since we know the limitations of the PIC16F84A, and have to use it anyway, we decided to instead make our code more readable instead.

# Summary

All in all we believe to have met the goals we were set out to achieve. More importantly, this project was a bridge to furthering our knowledge in assembly and programming PICs and served that purpose well. At this point we could easily use the same code we used for the flashing amber light and wire up the buzzer to another pin to make it even more realistic.

From watching actual pelican crossings, I noticed several things:

- The timing is different
- Amber doesn't flash on some and turns on together with red before the transition
- There is a significant delay before and after a person presses the button to avoid a complete traffic standstill
- The red pedestrian light is always on even when amber is and only switches to green a while after the red traffic light has been on
- Things change depending on time of day

Although our program clearly does not go into that much depth, we are still able to see how it could be done. All it would take is adding delays in the correct places, tweaking sequences a little or implementing the use of a way to measure time be it external or internal. The reason we didn't was because the requirements didn't exactly state it and we assume expanding our software in this manner would become much simple when we start using C.

*End.*