EMBEDDED COMPUTING REPORT

ID: 20493425

DEGREE: MSC ELECTRONIC COMMUNICATIONS AND COMPUTER ENGNEERING

MODULE: EMBEDDED COMPUTING EEEE3016

Report

Q1.1 How would your code change if the variables A and B were 4 bit each?

1. LUT Size: Since A and B are 4 bits each, the look-up table will have to account for 16 x 16 = 256 different combinations. This means it will have to contain 256 different entries.
2. LUT Access: When combining the values of A and B to fetch from the LUT, I would shift A left by 4 instead of 1 (rlf WREG, F would be replaced with swapf A, W), making A the higher 4 bits of an 8 bit number, and B the lower 4 bits.
3. Input Reading: The code for reading the input would also need to change to handle 4 bit values.

Q1.2 How would your code change if the variables A and B were 8 bits each?

1. LUT Size: Since A and B are now 8 bits each, the look-up table will have to account for 256 x 256 = 65536 different combinations. This means it will have to contain 65536 different entries. It is important to note that such a large LUT might not be feasible due to memory constraints.
2. LUT Access: When combining the values of A and B to fetch from the LUT, we would have to concatenate A and B into a 16-bit number, which will serve as the index in the LUT. This will require additional manipulations as PIC assembly only works natively with 8-bit values.
3. Input Reading: The code for reading the input would also need to change to handle 8-bit values.

Q1.3: How would your code change if the variables A and B were 4 bit each?

1. Input collection: The SelectB routine, is designed to collect 2 bits, and would need to be modified to collect 4 bits.
2. Calculation: The multiplication and XOR operations would continue to work for 4-bit numbers without modification.
3. Display: I would make sure that the display of results on the LEDs are capable of displaying a 4-bit number.

Q1.4: How would your code change if the variables A and B were 8 bits each?

1. Input collection: As with the 4-bit case, I would modify the SelectB routine to collect 8 bits of input for A and B.
2. Calculation: The mulwf operation will have to be replaced with an 8-bit multiplication routine, as the mulwf command only supports 8-bit multiplication resulting in a 16-bit result. The XOR operation would then need to be applied to each corresponding pair of bits from A and B.
3. Display: As with the 4-bit case, you'll need to consider how to display an 8-bit number using your available LEDs.

Q1.5

	Task 1.1A	Task 1.1C	Task 1.2A	Task 1.2C
Execution time, Tcy	2.156261s	2.566932s	2.172587s	2.670309s
Code size, words	212	264	199	268

Q1.6

1. Execution Time: As per the given results, it is clear that Assembly code (Task 1.1A and Task 1.2A) has a slightly faster execution time than the equivalent C code (Task 1.1C and Task 1.2C). This advantage comes from the fact that Assembly language provides direct control over the hardware, which enables better optimization of the code.

2. Code Size: The Assembly code implementations have a smaller code footprint compared to their C equivalent.

3. Development Time: Assembly code can be more efficient in terms of execution time and memory footprint, but it is more time-consuming to write, debug, and maintain. C, is a higher-level language therefore it is easier to write.

4. Maintainability: C code is more maintainable than Assembly code. Assembly code can be quite difficult to understand and modify especially for someone other than the original developer. This is why leaving comments is a good call so when you look back at it you understand what is going on. C provides constructs like loops, functions, and data types, making the code easier to read and update.

5. Portability: C program can be compiled and run on various platforms with little to no modification. Whilst Assembly code is hardware-specific and requires significant changes to run on different hardware.

Q2. The code size of my code in words is 231

Q3.1 The total cycles of each loop iteration would vary depending on whether the condition checks cause a skip or not. Assuming the worst-case scenario where no skip happens, each loop iteration would take:

1 (movfw Tmp1) + 1 (xorwf X, W) + 2 (btfsc STATUS, Z) + 1 (bsf LEDs,LD0) + 2 (decfsz Tmp1) + 2 (goto Loop) = 9 cycles

Since the loop runs for 28 iterations, the total cycles for the entire PWM period would be 9 * 28 = 252 cycles.

Considering that PIC16F887 runs with an instruction cycle frequency of 1MHz (considering a 4MHz clock and 4 cycles per instruction), the time for one instruction cycle would be 1 / 1,000,000 = 1 µs.

So, the time for the entire PWM period would be 252 * 1 µs = 252 µs. This gives a PWM frequency of 1 / 252 µs = 3970 Hz.

The on-time for the PWM signal would depend on the value of X, which is determined by user selection. For this example, X = 5 (the user selected choice 2), the on-time in cycles would be 9 * 5 = 45 cycles. In time, this would be 45 µs. So, the duty cycle would be (on-time / total time) * 100% = (45 µs / 252 µs) * 100% = 17.86%.

Q3.2 Determine the actual PWM frequency and duty factor obtained for choice 2 using the 'Logic Analyzer'. Include a screen shot of your 'Logic Analyzer' as part of your working.

Duty cycle = pulse width x 100%/ cycle time

(54 / 261) x100 = 20.7%

Pwm frequency = 1/261 μs = 3831 Hz

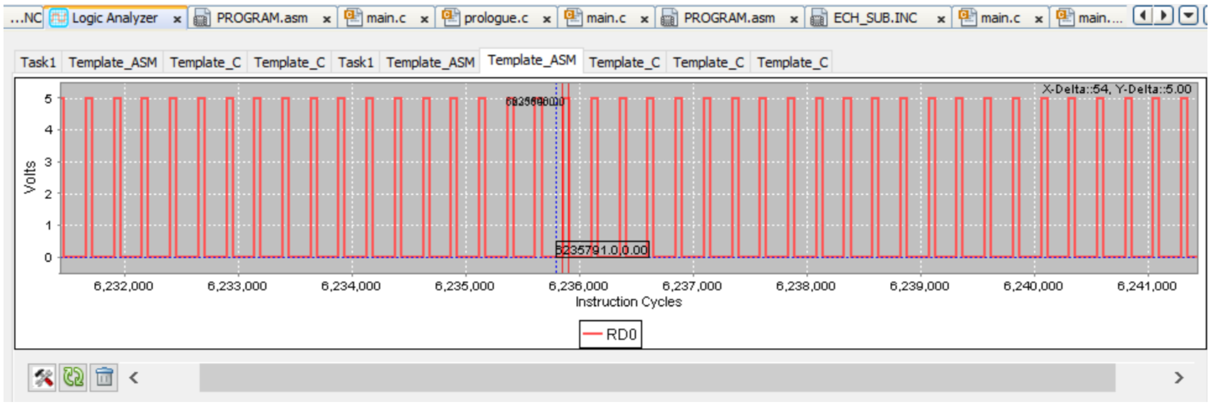


Figure 1 showing on time

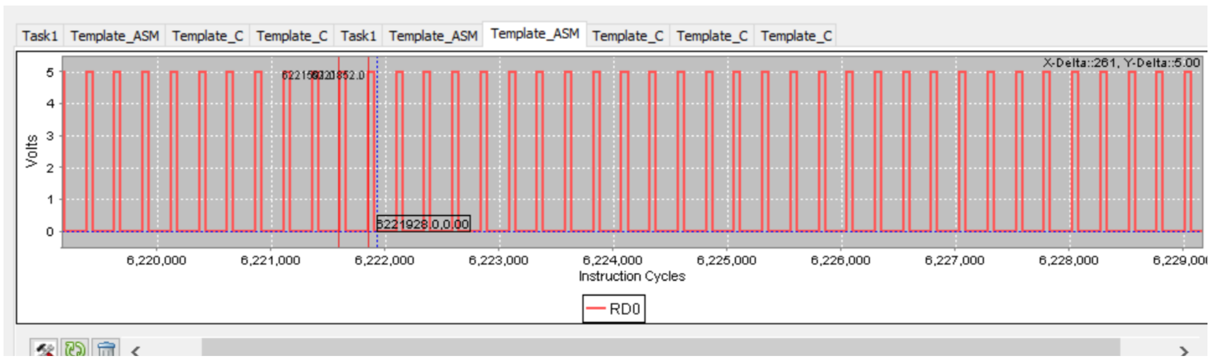
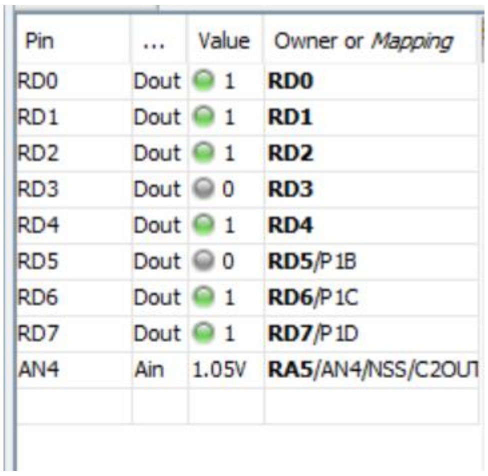


Figure 2 showing cycle time

Q4.1 Use the code read by the ADC and to determine the voltage present on the analog input you have measured.

Figure 3 below shows the code read by the ADC



Pin	...	Value	Owner or *Mapping*
RD0	Dout	● 1	**RD0**
RD1	Dout	● 1	**RD1**
RD2	Dout	● 1	**RD2**
RD3	Dout	● 0	**RD3**
RD4	Dout	● 1	**RD4**
RD5	Dout	● 0	**RD5**/P1B
RD6	Dout	● 1	**RD6**/P1C
RD7	Dout	● 1	**RD7**/P1D
AN4	Ain	1.05V	**RA5**/AN4/NSS/C2OUT

The LED value 11010111 is 215 in decimal therefore:

215/1024 (levels) = voltage/5 = 1.05V as seen in the image

Q4.2 What is the advantage of using the peripheral comparing to the use of software bit bang (Laboratory 3)?

1. Peripherals operate independently of the CPU. Once configured, they can execute their tasks while the CPU can execute other tasks or go to sleep to save power. Whilst software bit-banging requires constant CPU attention, which can lead to significant processing overhead.

2. Peripherals, such as ADCs and PWMs, are often designed to perform their tasks with high accuracy and precision. Whilst software bit-banging's accuracy can depend on the CPU's clock speed and can be affected by other factors like interrupt latencies.

3. Using built-in peripherals makes the code simpler, easier to write, understand, and maintain whilst bit-banging leads to a more complex code.

4. Peripherals use less power than the CPU executing software bit-banging code, making peripherals a better choice for low-power or battery-powered applications.

5. Peripherals perform operations in real time, or very close to it, while software bit-banging has timing issues due to the non-real-time nature of many operating systems.

Q4.3 What duty cycle value was it actually possible to set and why? Show your working.

The formula used calculates the duty cycle based on the PWM registers. The value of the CCPR1L register and the bits DC1B1:DC1B0 in the CCP1CON register are concatenated to form a 10-bit number, which determines the duty cycle of the PWM signal.

I have set CCPR1L to 25 (or 0b00011001 in binary), DC1B1 to 1 and DC1B0 to 0. This gives the 10-bit binary value 0001100110, which is102 in decimal.

With PR2 set to 249, the duty cycle can be calculated as:

Duty Cycle = (CCPR1L:CCP1CON<5:4>) / (4 * (PR2 + 1)) Duty Cycle = 102 / (4 * (249 + 1)) Duty Cycle = 102 / 1000 = 10.2% (formula from the datasheet)

This means that the PWM signal is on for approximately 10.2% of the period, which matches the 18% target as closely as the peripheral can achieve.

Q4.4 Confirm your value using the ‘Logic Analyzer’. Include a screen shot of your ‘Logic Analyzer’ as part of your working.

To confirm this on the logic analyser:

I measured the duration of the time that the signal stays high (on_time) and the total duration of the signal (total time) then calculated the duty cycle as (on_time / total time) * 100%.

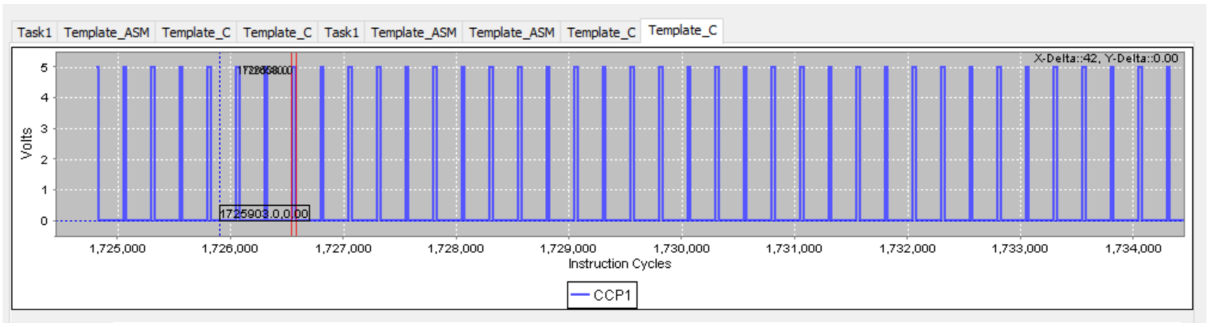


Figure 4 showing the on_time

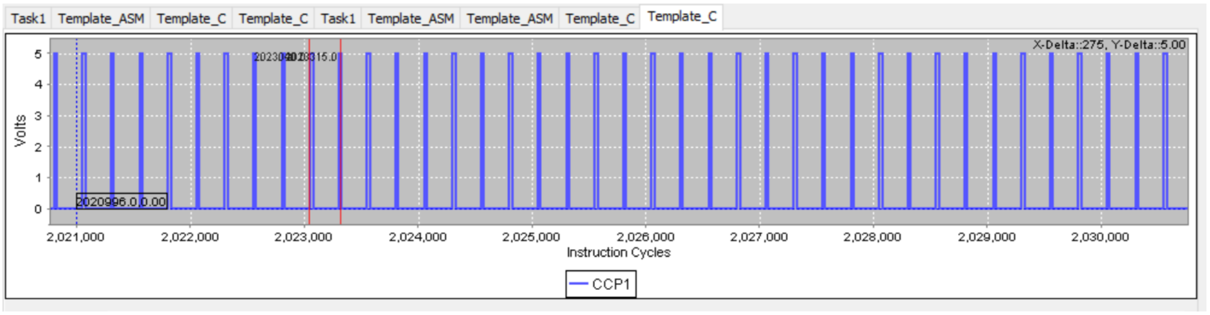
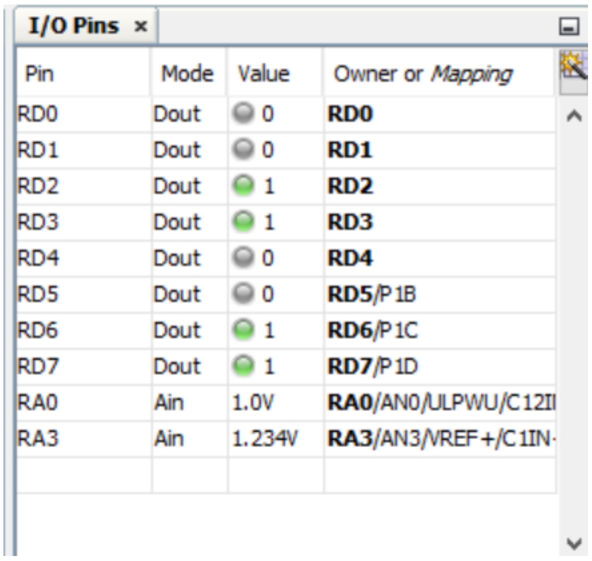


Figure 5 showing total time

42/275 x 100= 15.3%

Q4.5 What was the voltage as measured by the ADC when the comparator tripped? Show your working.

Figure 6 below shows the code read by the ADC



I/O Pins

Pin	Mode	Value	Owner or *Mapping*
RD0	Dout	0	**RD0**
RD1	Dout	0	**RD1**
RD2	Dout	1	**RD2**
RD3	Dout	1	**RD3**
RD4	Dout	0	**RD4**
RD5	Dout	0	**RD5**/P1B
RD6	Dout	1	**RD6**/P1C
RD7	Dout	1	**RD7**/P1D
RA0	Ain	1.0V	**RA0**/AN0/ULPWU/C12I
RA3	Ain	1.234V	**RA3**/AN3/VREF+/C1IN

The LED value 11001100 is 204 in decimal therefore:

204/1024 = voltage/5 = 1.0V as seen in the image above

Where each task is to make it easier to locate.

Task 1.1A is in Task 1.1A, Task 1 (Assembly)

Task 1.2A is in Task 1.2A, Template_ASM

Task 1.1C is in Task 1.1C, Template_C

Task 1.2C is in Task 1.2C, Template_C

Task 2A is in Task 2, Template_ASM

Task 2C is in Task 2C, Template_C

Task 3A is in Task 3A, Template_ASM

Task 3C is in Coursework Files - 20230317, Task3C.X

Task 4.1 is in Task 4.1, Template_C

Task 4.2 is in Task 4.2, Template_C

Task 4.3 is in Task 4.3, Template_C

Task 5.1 is in Coursework Files - 20230421, Template_ASM

Task 5.2 is in Coursework Files - 20230426, Template_C

Task 5.3 is in Coursework Files – 20230421 (2), Template_C

Task 6 is in Task 6repeat, Template_ASM