ENCM 369 PIC Activity 3

ENCM 369 ILS Winter 2021 Version 1.0

Activity Objectives

- Design and build a 6-bit LED counter circuit
- Produce a design flowchart from which you will write your code
- Introduce the ENCM 369 coding conventions
- Code the 6-bit counter per the coding conventions
- First code review of a peer

Deliverables

- One image file that includes a picture or screenshot of your circuit diagram beside the picture of your program flow chart (this will be part of your Git commit)
- Check-in of you code to Github
- Completed rubric review of someone's code in your learning community

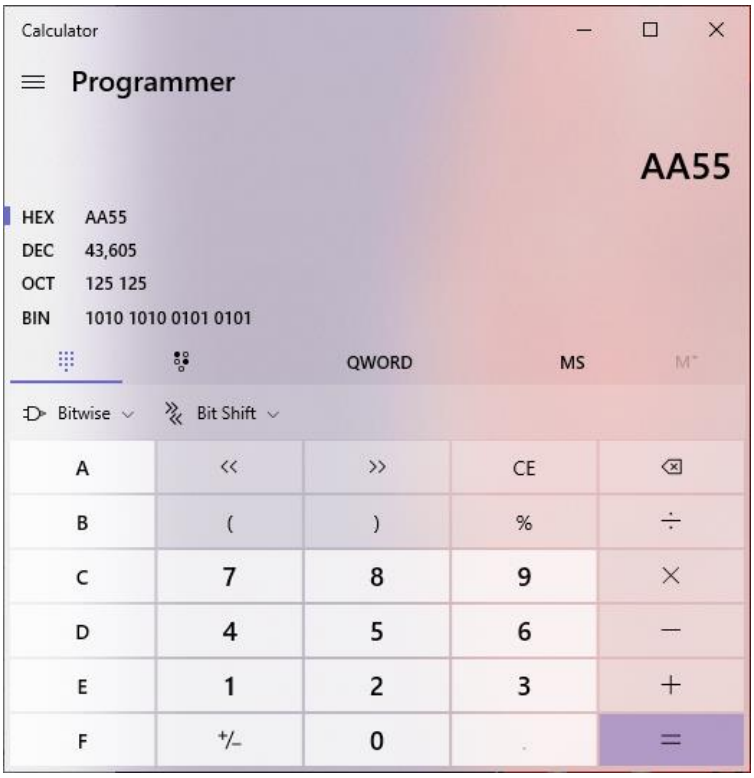
Background Information

When doing "low level" programming with a microcontroller to configure registers and interact with the processor's hardware, embedded developers make extensive use of binary and hexadecimal number representations. This is because these number systems so intuitively and directly map to bit positions in registers and are essential when considering the many bitwise operations that are required to manipulate data at this level.

Even if you have very little experience with base 2 and base 16 number systems, the nice part is that you really only need to be proficient at 16 numbers (0 to 15) since every hex number is just combinations of 4-bit values. The majority of problems you'll have to solve is simply knowing what bits are set or clear as indicated by a hex value. For example, if you need to set bits 0, 3 and 7 in an 8-bit register, the thought process you need is:

Bit's 0, 3, 7: b'1000 1001'
Hex equivalent: 0x89

If you ever need to convert hex to decimal or the other way around, or do any mathematical operations, just use a calculator! Windows Calculator has a "Programmer" mode that you can use to do anything that you can't do in your head, so keep it handy. In the example below, the calculator is in Hex mode, and you can see it shows the decimal, octal, and binary equivalents for you. Don't worry about octal values. Do you notice anything significant about the values 0xA and 0x5? Why do you think that 0xAA and 0x55 are very often used as test values?



For a bit of practice, fill out the table below as fast as you can.

Decimal	Hex	Binary (4 bit)
0	0x00	0000
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		
15		

The other skill you'll need extensively for embedded development is bitwise operations. Unlike the logical operators && (AND) and || (OR) that compare whole values, bitwise operators work bit-by-bit between two numbers to produce a result.

The bitwise operators (also called unary operators) are:

Bitwise OR: |
Bitwise AND: &
Bitwise XOR: ^

The truth tables for these operators are shown here, though you need to fill in the Z column:

OR		
A	B	Z
0	0	
0	1	
1	0	
1	1	

AND		
A	B	Z
0	0	
0	1	
1	0	
1	1	

XOR		
A	B	Z
0	0	
0	1	
1	0	
1	1	

As a quick test, write down the result of comparing the two values below using the three bitwise operators. Write the result in binary and hex.

```
b'01010101'
b'10001111'

OR: _____________________________________
AND: _____________________________________
XOR: _____________________________________
```

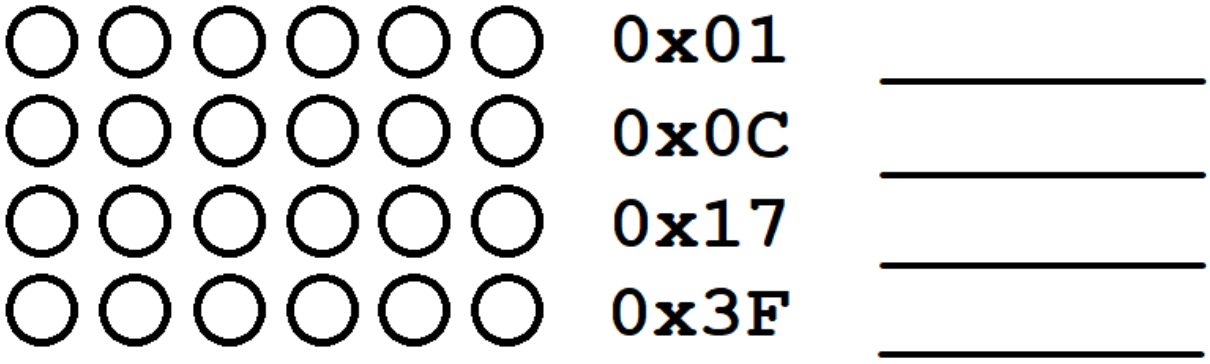
Just to be complete, here is a short code snippet from PIC Activity 2 for toggling RA0 using the bitwise XOR:

```
LATA = LATA ^ 0x01;  (can also be written LATA ^= 0x01;)
```

If you have any trouble with these concepts, there are infinite resources online to assist. It is fully expected that you have mastery of this concept going forward in this course.

6-bit Counter

Armed with this knowledge, you should be able to very quickly imagine what a 6-bit binary counter would look like. On the image below, fill in the circles (that represent LEDs) for the hex values provided. Remember each hex digit is 4-bits. LSB (bit 0) is always on the right. The MSB for a 6-bit counter is the 6th bit (bit 5).

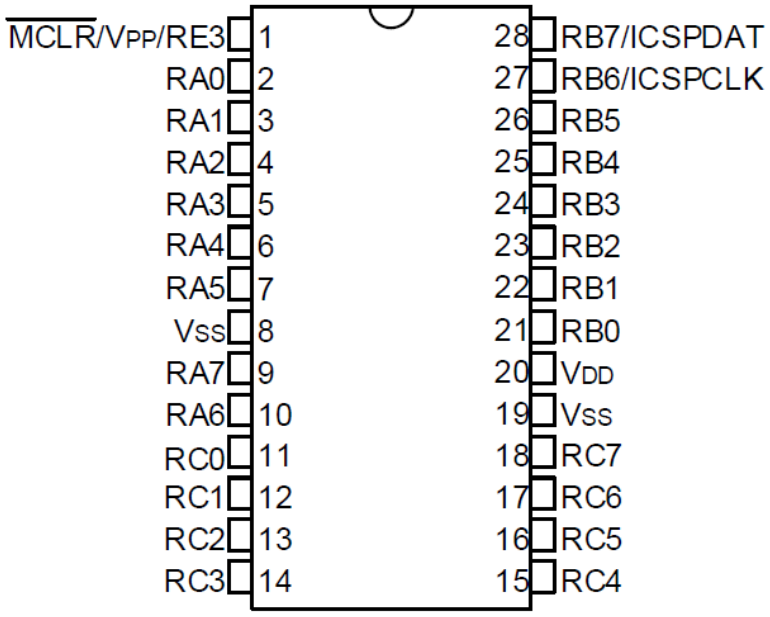


Think about what the LEDs would look like as the counter goes from 0 to 16.

Now, build this circuit on your breadboard. Use RA0 for bit 0, RA1 for bit 1, …, RA5 for bit 5. You've already done RA0 in PIC Activity 2. The only trick is figuring out how to physically get everything on your breadboard that will cause the least amount of trouble for wiring. Make sure, whatever you do, that the LEDs are lined up beside each other so the counter will look decent.

Add one more LED on RA7. This LED should be initialized ON when you set up the registers in your code and should never turn off as your counter runs. This is your indicator that you are writing only the counter bits in the output register.

Draw in the circuit components on the image below to create your circuit schematic. **This image is one of the deliverables for this activity**, so be neat. Label the parts with values so someone looking at your schematic could build the circuit. Don't forget power and ground connections, and PIC Kit connections (just indicate "to PIC Kit").



Program Design

It is extremely important to design programs before you ever start typing. Complicated problems must be broken down into manageable steps and functions. You can start to practice by documenting even simple problems like coding a 6-bit counter.

There is no wrong way to capture a design, though perhaps some better ways depending on exactly what you're doing. Flow charts are great for simple problems, pseudo code is good, too. Or some combination. What is important is that your design and your code are obviously representative of each other. This is particularly helpful when you're new to coding. Your design should indicate what you think you want to do, so when someone is debugging your code they can see what is supposed to happen vs. what you are actually doing.

As an example, let's write the pseudo code for PIC Activity 2:

```
Initialization (this runs only once when the code starts)
   -  Configure TRISA, ANSELA, and LATA (or PORTA) registers

while(1)
{
   -  Toggle RA0
   -  Wait 250ms
}
```

Draw or write out a program design for this activity. The counter should start at 0 and count to the maximum value, then rollover back to 0. The delay between counts should be about 250ms.

This design is the second deliverable for this activity.

Coding Conventions

Writing neat, clean code is a valuable skill. Well-written, commented, organized code is not only easier to read and understand, but significantly easier to debug. While individual programmers will always have their own style, standardizing to best-practices is an essential part for developing in industry.

Most companies will have a coding standard that defines the rules and expectations for code written by employees. The standard may cover anything from cosmetic requirements, all the way to restricting things like pointer use, or certain coding practices.

For this course, we will use a coding standard from a local company. The document is called "Engenuics Coding Standard.pdf" and is posted at the top of the PIC Activity section in D2L. The standard mostly covers cosmetic and organizational aspects of the code and references an industry standard called MISRA C for a lot of other rules. You do NOT need to look at the MISRA C standard. For this activity, pages 4-6 are relevant and should be read. The grading

rubric for this activity will be checking conformance to the standard. Even if you don't agree with the standard, you must follow it because it is the standard in this course.

Also be sure that your code is cleaned up, commented, and organized when you commit it for assessment. Delete any code that you commented out during development that you didn't end up using. Make sure that any cut and pasting operations didn't break the formatting rules. Delete big blocks of whitespace. Neatness counts, and it is frankly irritating to read code that looks like garbage. The quality of your code is just one of the many attributes on which you will be judged when working in industry or when be considered for a job, so practicing now will help you in the future.

Programming

Be sure to use a new branch in Github called PIC_Activity3 for this firmware. This branch should come from the Master branch, which should have the course's template project files in it. Once you do the branch in Github Desktop, create the MPLAB project and copy the project files into the Git repo folder. Open the project in MPLAB and confirm it builds. There is a video on D2L that shows you every step.

Next, implement your design from the previous section making sure you are following the coding conventions. You will again use GpioSetup() for the register initialization and this is the only place in your code that you are allowed to turn on RA7. Use UserAppRun() for the rest of the code including the delay. The delay should be about 250ms +/- 250ms (don't waste time scoping this out to get it exactly 250ms).

Use the debugger and/or simulator to SINGLE STEP through the code you wrote to ensure that what you have written is actually doing what you expect. When stepping through your delay loop, use a breakpoint after the delay loop so you can run through this at full speed. You will not be granted an extension for this activity if you complain that it took too long to single step through a million loop instructions.

Hints:

- Don't worry that this is running on a microcontroller. If you were asked in your C programming class to write a counting loop, you could do it in about 5 seconds.
- When you update the counter to the LEDs, make sure you only update bits RA0-RA5 in LATA. There are a few ways to do this, including some ways that have not been discussed or that you would not yet know so it is not expected that you search for these methods. You have everything you need to be able to do this.
- If the light on RA7 goes off, you did something wrong.