CONTENTS

List of Flowcharts

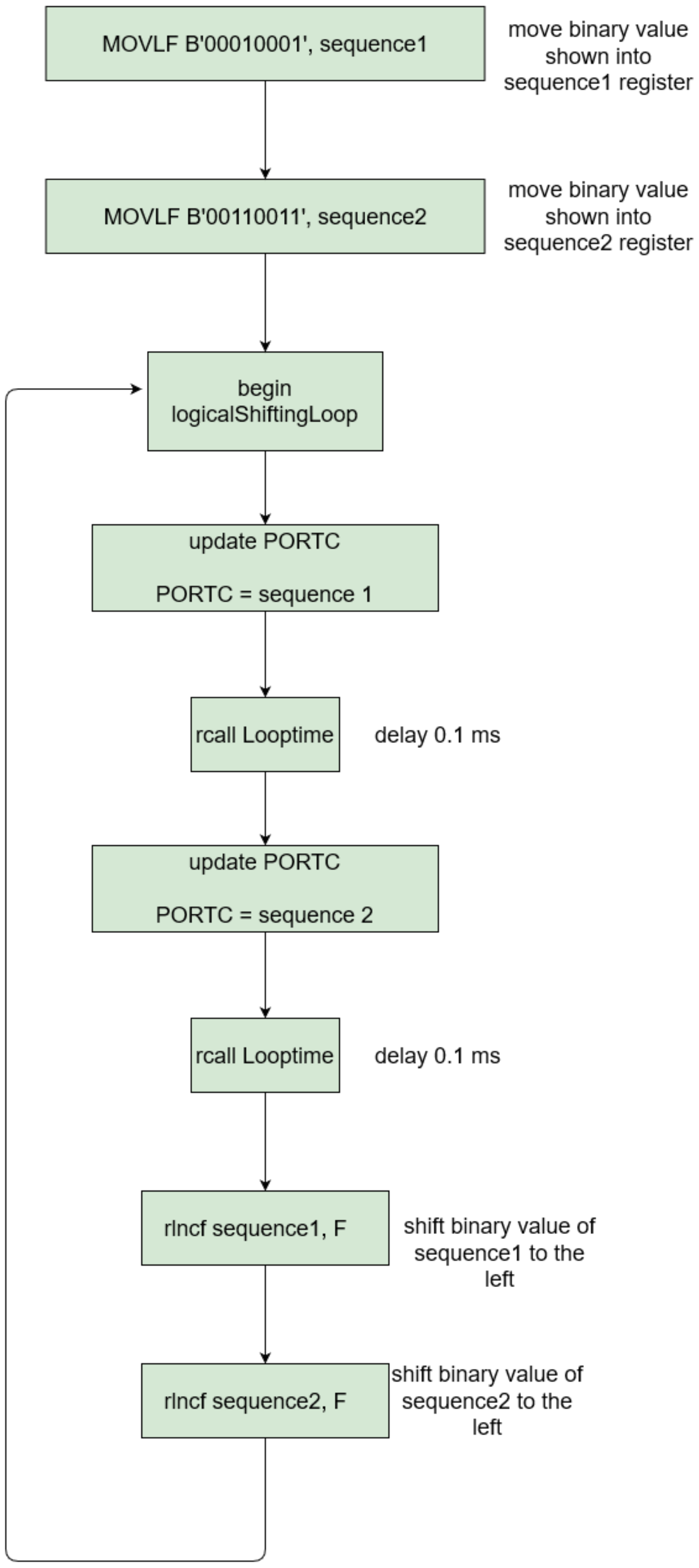

MOVLF B'00010001', sequence1
move binary value shown into sequence1 register
MOVLF B'00110011', sequence2
move binary value shown into sequence2 register
begin logicalShiftingLoop
update PORTC
PORTC = sequence 1
rcall Looptime
delay 0.1 ms
update PORTC
PORTC = sequence 2
rcall Looptime
delay 0.1 ms
rlncf sequence1, F
shift binary value of sequence1 to the left
rlncf sequence2, F
shift binary value of sequence2 to the left

Fig. 1. Mainline program for problem 3

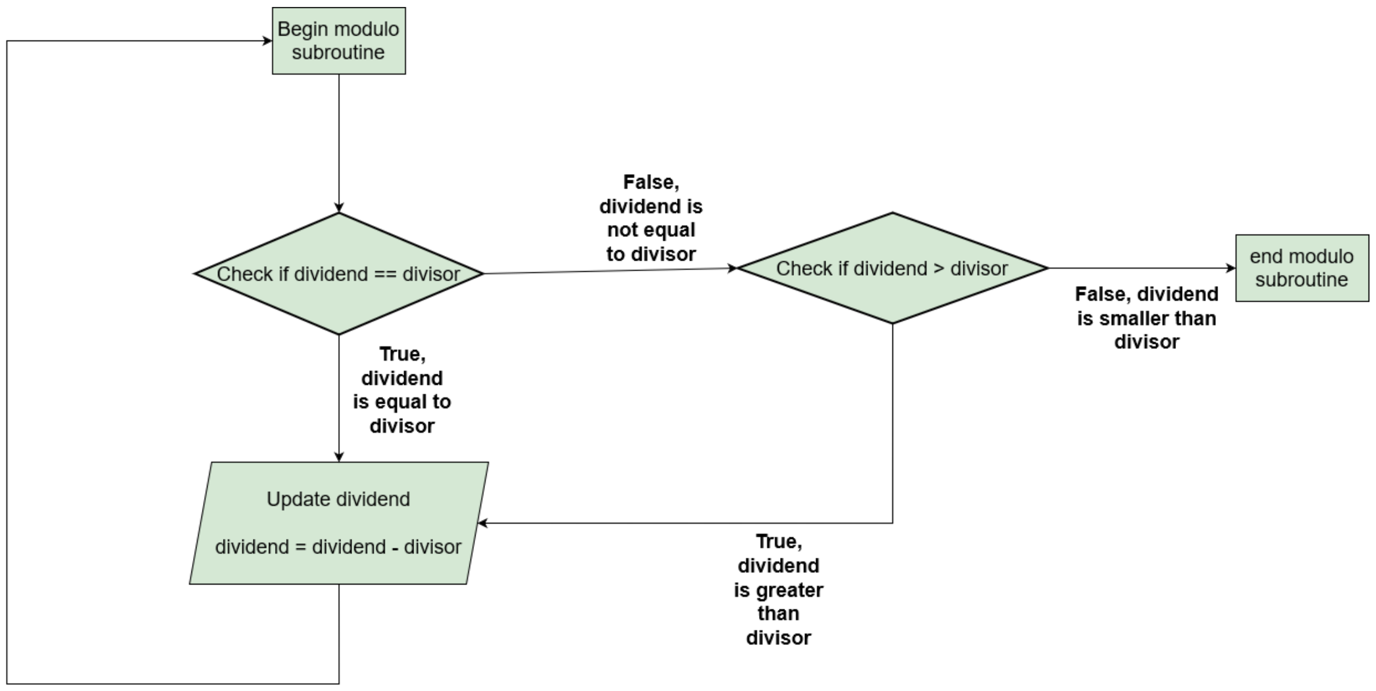

Begin modulo subroutine
Check if dividend == divisor
False, dividend is not equal to divisor
Check if dividend > divisor
end modulo subroutine
False, dividend is smaller than divisor
True, dividend is equal to divisor
Update dividend
dividend = dividend - divisor
True, dividend is greater than divisor

Fig. 2. Modulo subroutine flowchart

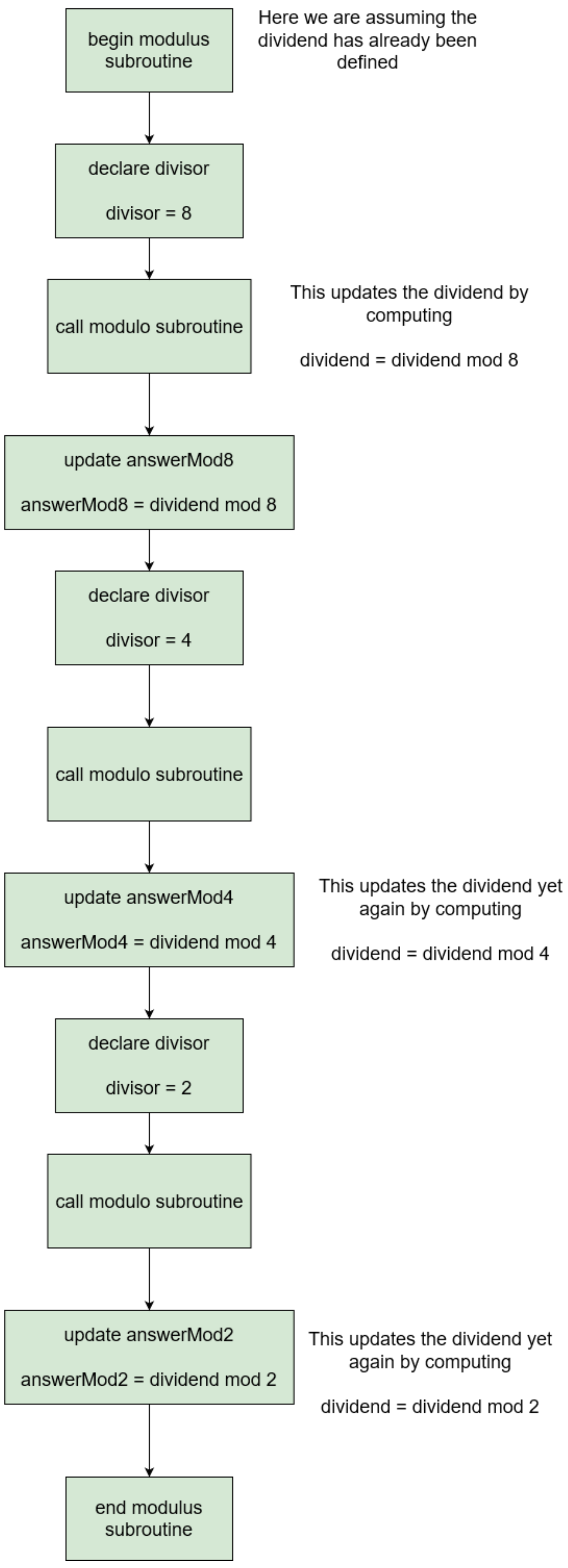

begin modulus subroutine
Here we are assuming the dividend has already been defined
declare divisor
divisor = 8
call modulo subroutine
This updates the dividend by computing
dividend = dividend mod 8
update answerMod8
answerMod8 = dividend mod 8
declare divisor
divisor = 4
call modulo subroutine
update answerMod4
answerMod4 = dividend mod 4
This updates the dividend yet again by computing
dividend = dividend mod 4
declare divisor
divisor = 2
call modulo subroutine
update answerMod2
answerMod2 = dividend mod 2
This updates the dividend yet again by computing
dividend = dividend mod 2
end modulus subroutine

Fig. 3. Modulus subroutine (implemented) without dynamic register assignment

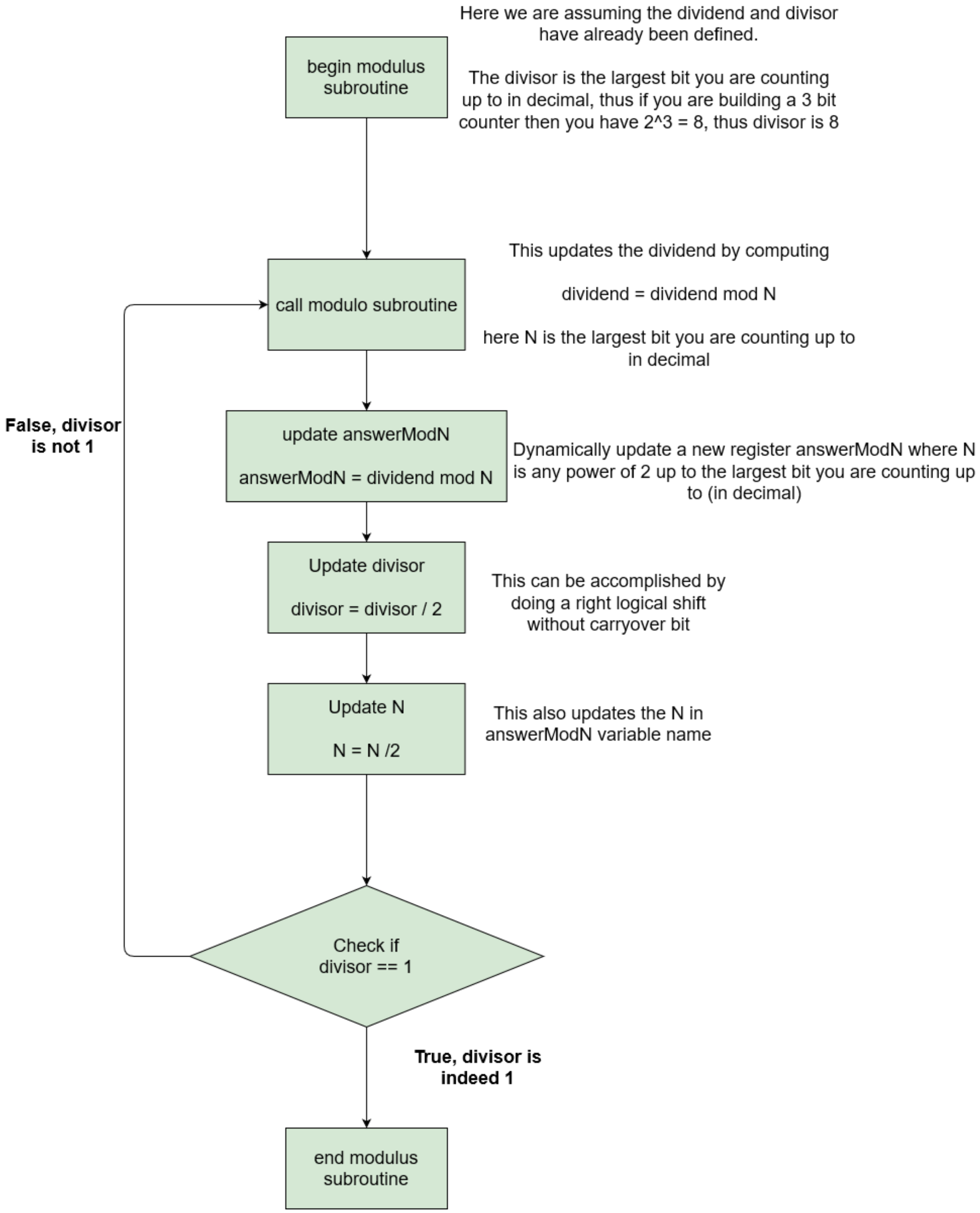

Here we are assuming the dividend and divisor have already been defined.
The divisor is the largest bit you are counting up to in decimal, thus if you are building a 3 bit counter then you have 2^3 = 8, thus divisor is 8
begin modulus subroutine
call modulo subroutine
This updates the dividend by computing
dividend = dividend mod N
here N is the largest bit you are counting up to in decimal
update answerModN
answerModN = dividend mod N
Dynamically update a new register answerModN where N is any power of 2 up to the largest bit you are counting up to (in decimal)
Update divisor
divisor = divisor / 2
This can be accomplished by doing a right logical shift without carryover bit
Update N
N = N /2
This also updates the N in answerModN variable name
Check if divisor == 1
False, divisor is not 1
True, divisor is indeed 1
end modulus subroutine

Fig. 4. Modulus subroutine (unable to implement) with dynamic register assignment

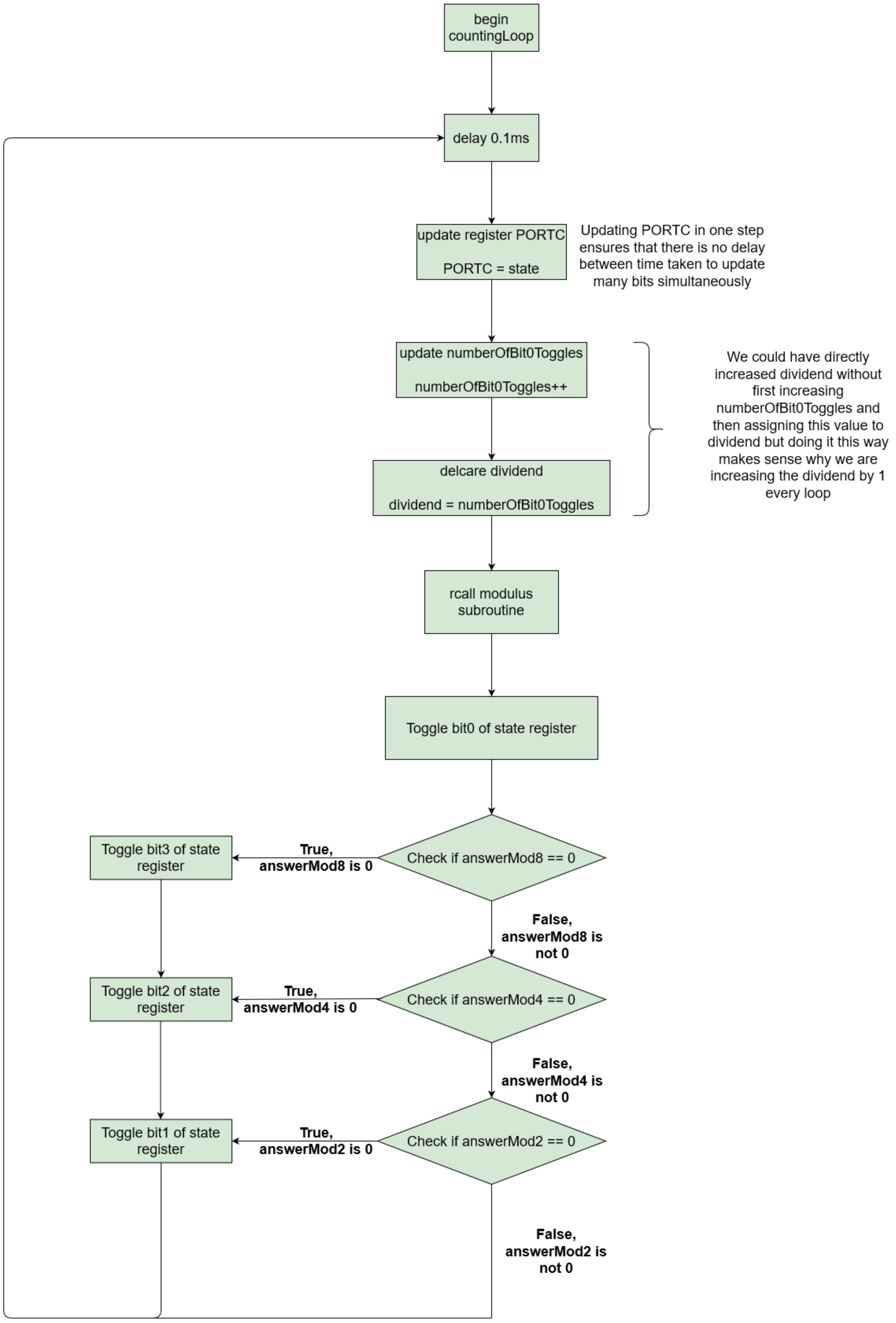

begin countingLoop
delay 0.1ms
update register PORTC
PORTC = state
Updating PORTC in one step ensures that there is no delay between time taken to update many bits simultaneously
update numberOfBit0Toggles
numberOfBit0Toggles++
delcare dividend
dividend = numberOfBit0Toggles
We could have directly increased dividend without first increasing numberOfBit0Toggles and then assigning this value to dividend but doing it this way makes sense why we are increasing the dividend by 1 every loop
rcall modulus subroutine
Toggle bit0 of state register
Check if answerMod8 == 0
True, answerMod8 is 0
Toggle bit3 of state register
False, answerMod8 is not 0
Check if answerMod4 == 0
True, answerMod4 is 0
Toggle bit2 of state register
False, answerMod4 is not 0
Check if answerMod2 == 0
True, answerMod2 is 0
Toggle bit1 of state register
False, answerMod2 is not 0

Fig. 5. countingLoop to count from 0 to 15

Exp 4 - Pulse Train Synchronization

Anas Ashraf

I. Problem 1: Logical Shifts

We are prompted to produce the sequence shown in table I using register PORTC and the bits RC0, RC1, RC2, and RC3. We see that this is a simple shift register and this can be solved using a left logical shift of the PORTC register when we define the first four and the last four bits of PORTC to be 0001 and then shift them to the left for every step.

A. Problem Outline

TABLE I
Desired sequence to be reproduced using bits RC0 to RC3.

Ch. 4	Ch. 3	Ch. 2	Ch. 1
0	0	0	1
0	0	1	0
0	1	0	0
1	0	0	0

B. Solution

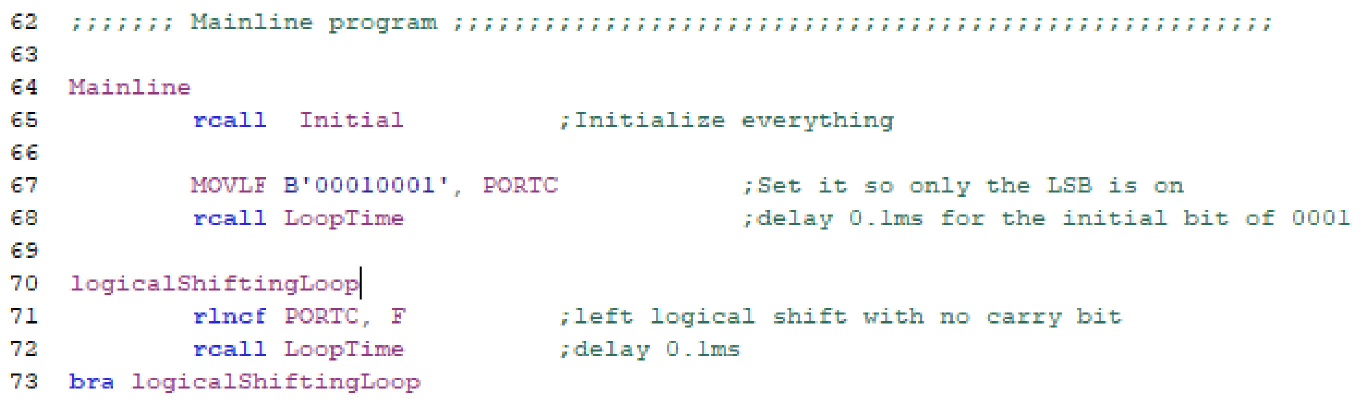
The code solution for this problem is shown in figure 6. As you can see we begin by executing $\boxed{\textit{MOVLF B'00010001', PORTC}}$ and notice that this is step 1 of the desired sequence shown in table I. Also note that the first four bits of PORTC are the same as the last four, this is because we are only dealing with reproducing a 4 bit pattern thus they first four bits must be identical to the last four bits. Every instance of $\boxed{\textit{rcall LoopTime}}$ only causes a 0.1ms delay so after setting PORTC to display 0001 as per step 1 of the desired sequence we delay 0.1ms before moving on to the next label which is $\boxed{\textit{logicalShiftingLoop}}$ wherein we will remain in this loop for the rest of the runtime.

The first step in this loop is to execute $\boxed{\textit{rlncf PORTC, F}}$ which stands for rotate left with no carry bit the register PORTC and store the rotated value back into PORTC. By rotate we mean a shifting of the registers. Thus it shifts the binary value in PORTC from 0001 $\underline{0001} \rightarrow$ 0010 $\underline{0010}$ where the underlined numbers are what we see in the bits RC3, RC2, RC1, and RC0 respectively. Then we delay by 0.1 ms.

In the next iteration of the logicalShiftingLoop we shift the values for PORTC again going from 0010 $\underline{0010} \rightarrow$ 0100 $\underline{0100}$ and then delaying for 0.1ms again.

In the last unique iteration of the logicalShiftingLoop we shift the values for PORTC from 0100 $\underline{0100} \rightarrow$ 1000 $\underline{1000}$ and then delay 0.1ms again.

After this we loop back to step one as PORTC goes from 1000 $\underline{1000} \rightarrow$ 0001 $\underline{0001}$ and the loop continues infinitely. We get an output waveform shown in figure 7.



```
;;;;;;; Mainline program ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

Mainline
        rcall  Initial          ;Initialize everything

        MOVLF B'00010001', PORTC            ;Set it so only the LSB is on
        rcall LoopTime                      ;delay 0.1ms for the initial bit of 0001

logicalShiftingLoop
        rlncf PORTC, F          ;left logical shift with no carry bit
        rcall LoopTime          ;delay 0.1ms
bra logicalShiftingLoop
```

Fig. 6. Mainline program for left logical shifting of 0001 sequence

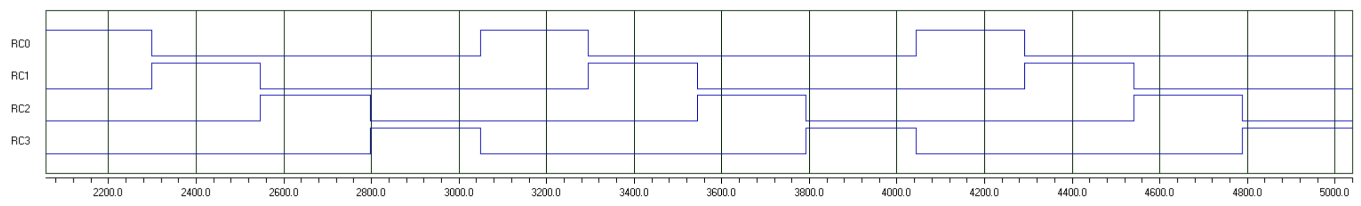


Fig. 7. Output waveform for PORTC for mainline in figure 6

II. Problem 2: Logical Shifts with Different Initial Conditions

A. Problem Outline

The sequence we are asked to produce in this problem is similar to that of section I but the only difference is that now we have a different sequence of bits to shift. The core logic to produce the shifting still stays the same, only the initial condition changes. The desired sequence is shown in table II and notice we can simply solve this by modifying the mainline program in figure 6 to start with a different initial condition when defining PORTC in line 67.

TABLE II
Desired sequence to be reproduced for problem 2 using bits RC0 to RC3 of register PORTC.

Ch. 4	**Ch. 3**	**Ch. 2**	**Ch. 1**
0	0	1	1
0	1	1	0
1	1	0	0
1	0	0	1

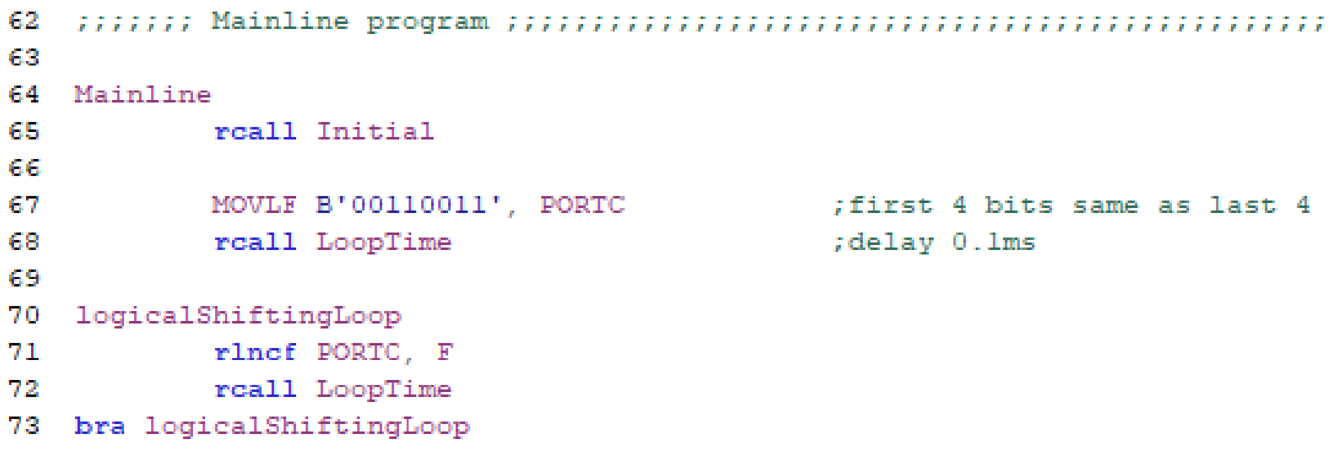
B. Solution

The code implementation of this problem is shown in figure 8. As you can see the only change is in line 67 where we change the code from $\boxed{\textit{MOVLF B'00010001', PORTC}}$ $\rightarrow$ $\boxed{\textit{MOVLF B'00110011', PORTC}}$. All the other code is completely identical. After initializing everything in line 65 in figure 8 and defining PORTC we delay 0.1ms so that the logical analyzer can display the states for PORTC 0011 $\underline{0011}$ for 0.1ms, where the underline numbers once again represent the bits RC3, RC2, RC1, RC0 respectively.

As we go through our first iteration of the logicalShiftingLoop we go from 0011 $\underline{0011}$ $\rightarrow$ 0110 $\underline{0110}$ and then delay 0.1ms.

The next iteration of logicalShiftingLoop has us go from 0110 $\underline{0110}$ $\rightarrow$ 1100 $\underline{1100}$ and then delay 0.1ms.

The last unique loop has us go from 1100 $\underline{1100}$ $\rightarrow$ 1001 $\underline{1001}$ and then delay 0.1ms. After this we loop back to previously visited steps since we go from 1001 $\underline{1001}$ $\rightarrow$ 0011 $\underline{0011}$ and repeat infinitely. The waveform is shown in figure 9.



```
62  ;;;;;;; Mainline program ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
63
64  Mainline
65          rcall Initial
66
67          MOVLF B'00110011', PORTC          ;first 4 bits same as last 4
68          rcall LoopTime                    ;delay 0.1ms
69
70  logicalShiftingLoop
71          rlncf PORTC, F
72          rcall LoopTime
73  bra logicalShiftingLoop
```

Fig. 8. Mainline program for problem 2

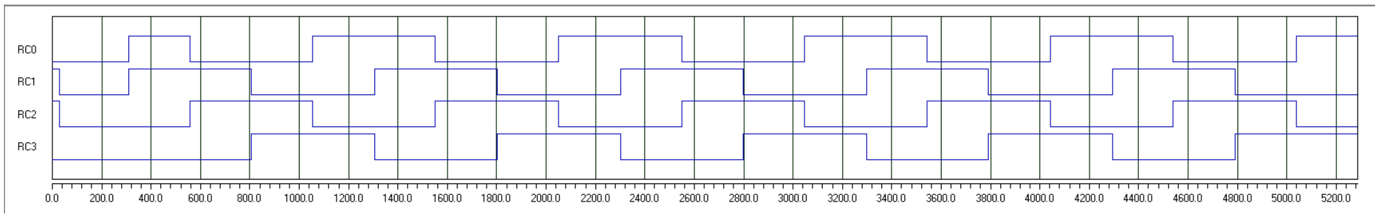


Fig. 9. Logical analyzer waveform for bits RC3-RC0 of PORTC when executing mainline in figure 8

III. Problem 3: Logical Shifts and Proxy Registers

A. Problem Outline

The sequence we wish to reproduce for problem 3 is shown in table III which, if you pay close attention, you will notice is actually a combination of the sequence in section I (highlighted in table III) and the sequence in section II (non-highlighted rows in table III). Thus, the solution is to simply add the two mainline codes of figures 6 and 8.

TABLE III
Desired sequence to be reproduced for problem 3 using bits RC0 to RC3 of register PORTC.

Ch. 4	Ch. 3	Ch. 2	Ch. 1
0	0	0	1
0	0	1	1
0	0	1	0
0	1	1	0
0	1	0	0
1	1	0	0
1	0	0	0
1	0	0	1

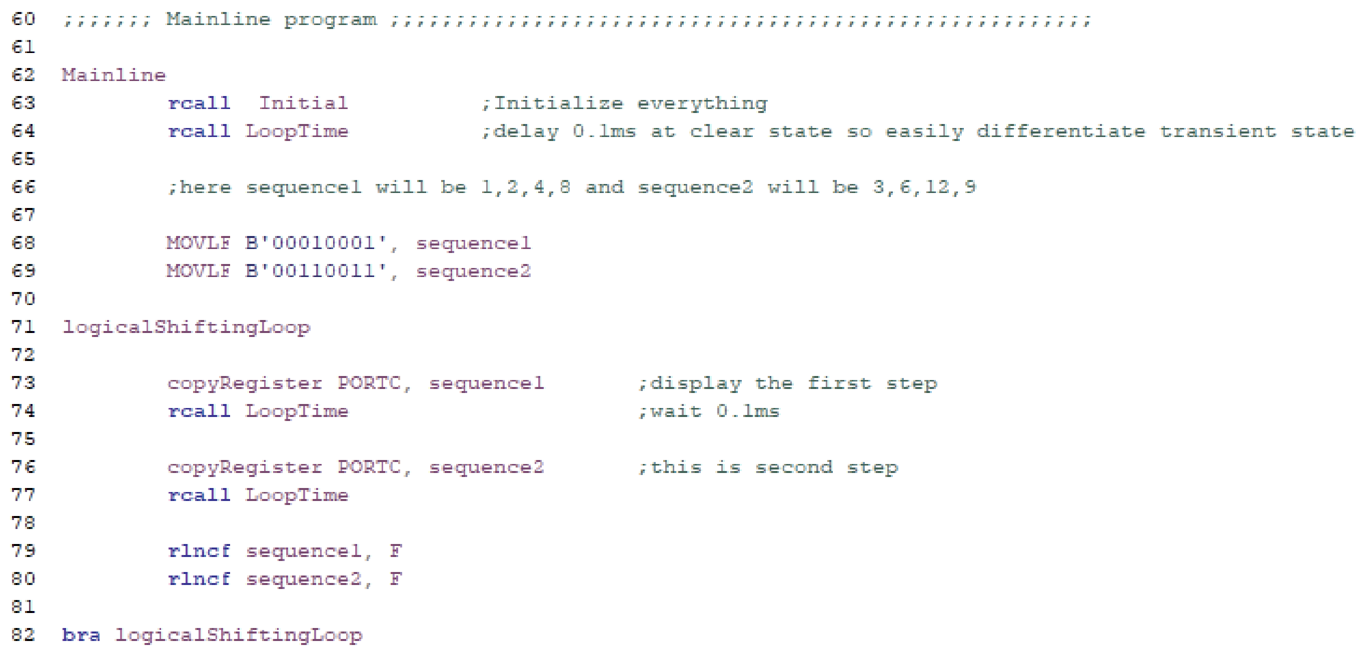
B. Solution

To combine the two mainline programs covered previously in figures 6 and 8 we will use two proxy registers from which we will update PORTC periodically. Let us define a variable called sequence1 which will hold the initial condition for the highlighted rows in table III and define another variable, sequence2, which will hold the initial condition for the non-highlighted rows in table III. Now we will update PORTC once every loop and at the end of the loop we will shift the proxy registers sequence1 and sequence2 to the left once.

Now let us see how we would implement the idea presented above in a flowchart so we can begin coding. Refer the flowchart in figure 1. We begin by executing $\boxed{\textit{MOVLF B'00010001', sequence1}}$ which moves the inital condition for the highlighted rows in table III into the proxy register sequence1. Then we do the same for sequence2. After the initial conditions have been established into each of the proxy registers we are ready to begin our infinite loop to produce the values in table III.

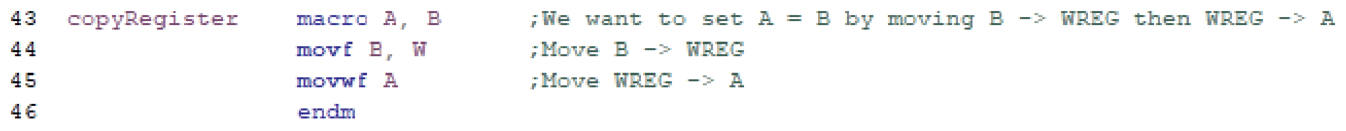
We first begin by updating PORTC so as to display the first step in table III which is 0001 $\underline{0001}$ where the underlined numbers are what we see in bits RC3, RC2, RC1, RC0 respectively. we delay for 0.1ms by calling LoopTime. Now we go to our next row of values which is the first non-highlighted row by moving the binary value in the proxy register sequence2 into PORTC, essentially having PORTC display 0011 $\underline{0011}$. Now we have finished going through the first 2 steps of table III. To prepare for the next steps we shift the binary values of sequence1 to the left so that sequence1 goes from 0001 $\underline{0001}$ $\rightarrow$ 0010 $\underline{0010}$ and then we shift the values of sequence2 to the left so that sequence2 goes from 0011 $\underline{0011}$ $\rightarrow$ 0110 $\underline{0110}$. Then we loop back to our logicalShiftingLoop. This will loop back with the values for sequence1 and sequence2 changing like we have described in sections I and II respectively. We will first update PORTC with sequence1 so that the second highlighted row in table III is displayed and then update PORTC with sequence2 so that the second non-highlighted row in table III is displayed. This will keep looping until we have displayed all the values of the rows in table III.

The code implementation of this is shown in figure 10. As you can see we initially execute $\boxed{\textit{rcall Initial}}$ followed by $\boxed{\textit{rcall LoopTime}}$ so we can easily differentiate the transient values from the actual program as there will be a big gap between them. Then we go on and define sequence1 such that it will display the values 1,2,4,8 when it is shifted to the left and define sequence2 to display the values 3,6,12,9 when it is shifted to the left. The command copyRegister is not a native command but is instead defined by me to move the value of the variable sequence1 to PORTC. The macro definition for copyRegister is shown in figure 11. As you can see we firstly move the value in register B to WREG and then move the value from WREG to A. Thus this is an assignment macro where you can think of this as executing $A = B$. The steps in lines 79 and 80 of figure 10 exist to shift the proxy registers to the left just like we showed in the flowchart in figure 1. The output waveform generated by the mainline is shown in figure 12.



```
;;;;;;; Mainline program ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

Mainline
        rcall  Initial          ;Initialize everything
        rcall LoopTime          ;delay 0.1ms at clear state so easily differentiate transient state

        ;here sequence1 will be 1,2,4,8 and sequence2 will be 3,6,12,9

        MOVLF B'00010001', sequence1
        MOVLF B'00110011', sequence2

logicalShiftingLoop

        copyRegister PORTC, sequence1       ;display the first step
        rcall LoopTime                      ;wait 0.1ms

        copyRegister PORTC, sequence2       ;this is second step
        rcall LoopTime

        rlncf sequence1, F
        rlncf sequence2, F

bra logicalShiftingLoop
```

Fig. 10. Mainline program for the flowchart in figure 1



```
copyRegister    macro A, B      ;We want to set A = B by moving B -> WREG then WREG -> A
                movf B, W       ;Move B -> WREG
                movwf A         ;Move WREG -> A
                endm
```

Fig. 11. Macro definition for the copyRegister in problem 3

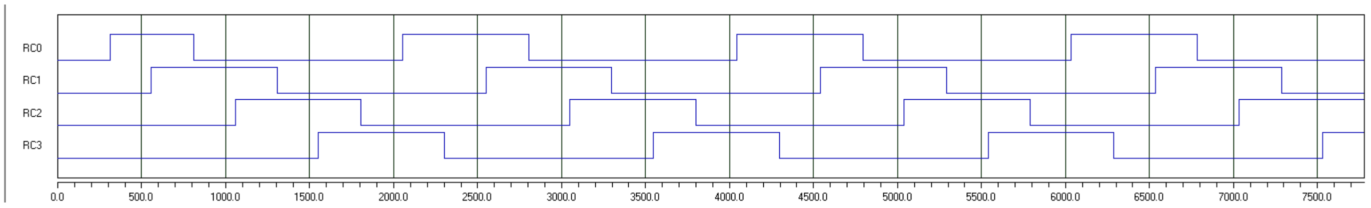


Fig. 12. Waveform generated by the mainline program in figure 10

IV. Problem 4: Counting from 0 to 15

A. Problem Outline

We wish to count from 0 to 15 using the bits RC0, RC1, RC2, and RC3 of register PORTC in the manner shown in table IV. This is a very simple problem that can be solved with a simple implementation of the modulo operator. So let us begin by first defining the modulo operator in assembler and then moving on with solving the problem .

B. The modulo operator

The modulo operator can be thought of as an operator that returns the remainder when a number (dividend) is divided by another number (divisor). For our intents we do not care about preserving the original dividend and later on you will see, it is actually slower to work with the divisor itself rather than the modulus of a divisor by some power of two. Thus, given all of this, our goal is to create a subroutine that takes accepts a dividend and divisor and returns the remainder. Since implementing

TABLE IV
DESIRED SEQUENCE TO BE REPRODUCED FOR PROBLEM 4 USING BITS RC0 TO RC3 OF REGISTER PORTC.

Ch. 4	Ch. 3	Ch. 2	Ch. 1
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1

division with a remainder is not a native function to the PIC microcontroller we will define division with a remainder by calling subtraction in a loop. The pseudo-code representation of this function is shown in listing 1.

```
while((dividend >= divisor)):
    dividend = dividend - divisor;           #this will store remainder in dividend
else:
    return dividend;                         #return only dividend (which stores remainder)
```

Listing 1. Pseudo-code for modulo operator

Thus, we essentially have to keep checking in every iteration of the modulo subroutine whether or not dividend is greater than or equal to the divisor, and if it is, then update the dividend by subtracting the divisor from it and storing the new value in the dividend, and when the dividend is finally less than the divisor, it becomes the remainder of the division, return the dividend. Now let us implement a more digestible version of the pseudo-code by breaking it down into a flowchart as is seen in figure 2. Notice when we begin the modulo subroutine we first check if the dividend is equal to divisor, and if it is, then update the value of the dividend by subtracting the divisor from it. Let us run through an example of 17 mod 8.

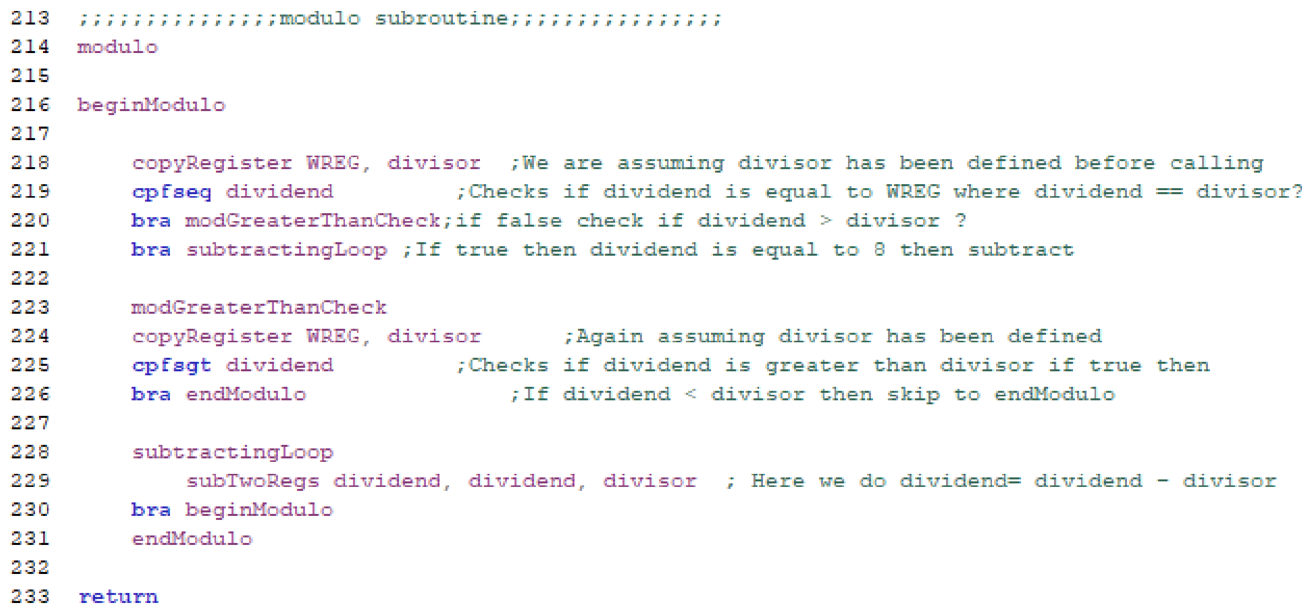
The dividend is 17 and the divisor is 8. We check if the dividend is equal to the divisor, that is checking to see if $17 == 8$. The check returns false, thus we check to see if the dividend is greater than the divisor, that is if $17 > 8$. This check returns true and thus we set dividend = dividend - divisor, meaning dividend = 17 - 8 which means dividend = 9. Now we check to see if dividend is equal to divisor or $9 == 8$. This returns false and thus we now check if $9 > 8$ which returns true thus we update dividend again, dividend = 9-8 or dividend = 1. Now we check if $1 == 8$ which returns false and then we check if $1 > 8$ which also returns false and thus we exit the subroutine.

The implementation of the flowchart into code is shown in figure 13. We begin by copying the divisor (that has been declared before the subroutine) into WREG. The instruction *cpfseq dividend* allows us to compare the dividend (which has also been declared prior) with the WREG, essentially comparing dividend with the divisor to see if they are equal. If they are NOT equal then we execute the next line which is *bra modGreaterThanCheck*. If they are indeed equal then we skip line 220 and branch to the subtractingLoop.

If the comparison in line 219 was not equal then we branch to modGreaterThanCheck label which first moves divisor to WREG using the copyRegister macro defined in section III, and then executes *cpfsgt dividend* which compares dividend with WREG and if dividend is smaller than divisor then we branch to endModulo, however, if dividend is indeed greater than WREG (divisor) then we skip the instruction which makes us branch to endModulo and go into subtractingLoop label which has only one instruction which is *subTwoRegs dividend, dividend, divisor*. This instruction subtracts the third argument (divisor) from the second (dividend) and stores it in the first argument (dividend). After executing this instruction you branch back to beginModulo and repeat this loop again. Now we have defined our modulo operator.

C. Counting using modulo

Refer back to table IV and notice that Ch. 2 is only toggled when Ch. 1 has toggled in multiples of 2, and Ch. 3 only toggles when Ch. 1 is toggled in multiples of 4, and finally Ch. 4 only toggles when Ch. 1 toggles in multiples of 8. You could alternatively also look at it the following way. Ch. 3 only toggles when Ch. 2 has toggled in multiples of 2, and Ch. 4 only toggles when Ch. 3 has toggled in multiples of 2. However, we found that the latter method uses more memory than the former method and thus we will use the viewpoint established by the former method in our program. This means we can count



```
;;;;;;;;;;;;;;modulo subroutine;;;;;;;;;;;;;;;
modulo

beginModulo

    copyRegister WREG, divisor  ;We are assuming divisor has been defined before calling
    cpfseq dividend         ;Checks if dividend is equal to WREG where dividend == divisor?
    bra modGreaterThanCheck;if false check if dividend > divisor ?
    bra subtractingLoop ;If true then dividend is equal to 8 then subtract

    modGreaterThanCheck
    copyRegister WREG, divisor      ;Again assuming divisor has been defined
    cpfsgt dividend         ;Checks if dividend is greater than divisor if true then
    bra endModulo               ;If dividend < divisor then skip to endModulo

    subtractingLoop
        subTwoRegs dividend, dividend, divisor  ; Here we do dividend= dividend - divisor
    bra beginModulo
    endModulo

return
```

Fig. 13. Code for the modulo subroutine

by keeping track of the number of toggles by ch. 1 (or numberOfBit0Toggles). If numberOfBit0Toggles mod 2 returns 0 then we must toggle Ch. 2 and if numberOfBit0Toggles mod 4 returns 0 then we must toggle Ch. 3 and if numberOfBit0Toggles mod 8 returns 0 then we toggle Ch. 4.

However, recall the properties of the modulus operator which is shown in equation 1.

$$\text{If } a \ mod \ b^n = 0, \ \textit{then } a \ mod \ b^m = 0 : n, m \in \mathbb{N}, m < n \tag{1}$$

Thus if numberOfBit0Toggles mod 8 returns 0 then it must be true (without checking we can state) that numberOfBit0Toggles mod 4 returns 0 and numberOfBit0Toggles mod 2 returns 0. Thus if Ch. 4 is toggled we must toggle both all other channels less than 4 (Ch.3, Ch. 2, Ch.1) and if Ch. 3 is toggled we must toggle all channels less than 3 and so on. This makes our counting algorithm more efficent without having to check if dividend mod 2 or mod 4 returns 0 when we already know dividend mod 8 returns 0. With this in mind we can design an n^{th} order counter using the modulus subroutine whose flowchart is shown in figure 4.

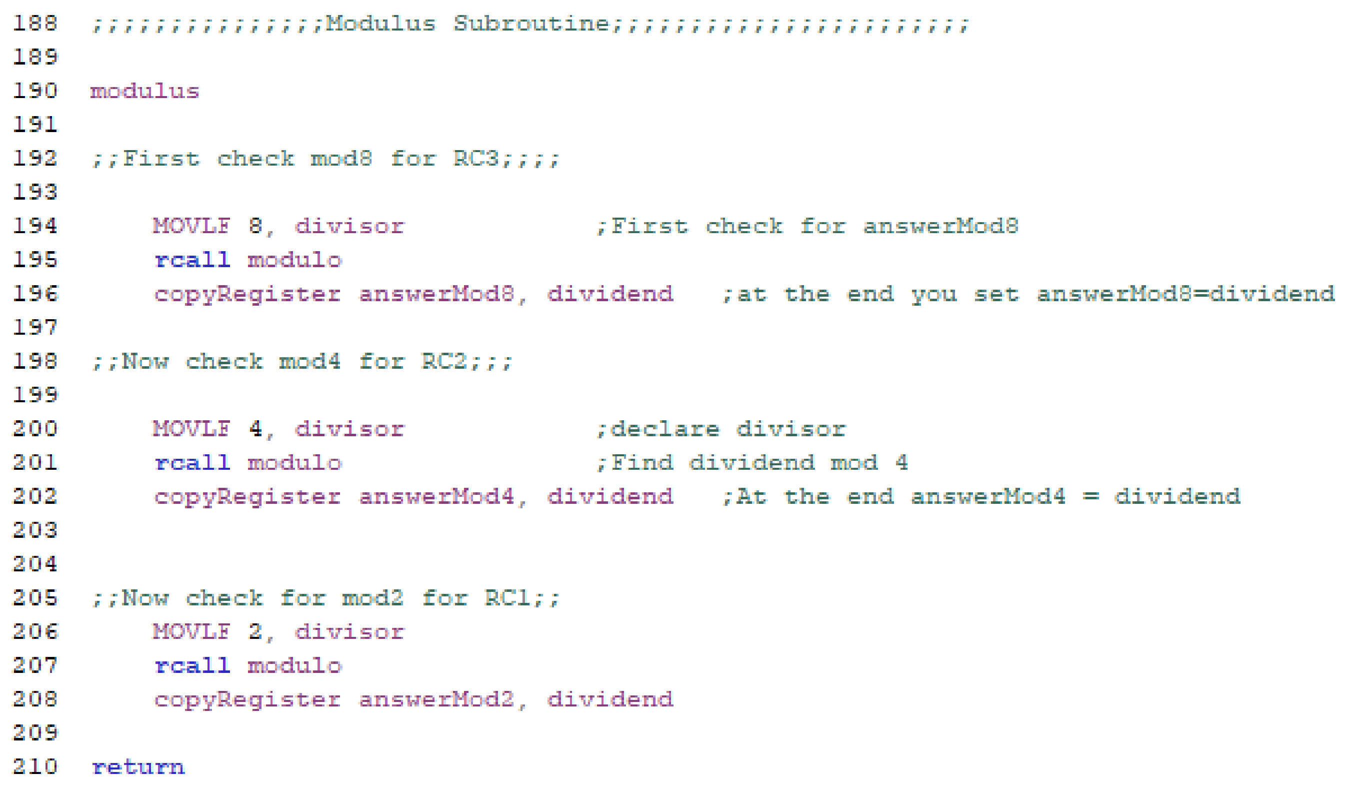
Here we are assuming that we have already defined the dividend (or recorded the numberOfBit0Toggles) and defined the order of bit size we are counting up to (or the largest divisor). We then call the modulo subroutine which returns the remainder in the dividend variable after which we update a variable called answerModN where N is the bit to which we are counting up to. So for instance let us assume the largest bit we are counting up to is the 1st bit (or in decimal 2), thus the value of N is 2. We update answerMod2 (replacing N with 2). Then we update the divisor to be half of the previous divisor (in this instance divisor will be 1) and notice we do the same to N (where N is also 1). Here we are assuming that updating N also updates the name of the variable answerModN. Then we check if divisor ==1 and since it is true we exit, if it were however false, for instance we are counting up to 32 for instance, then we go back to the beginning, compute answerMod32 and then update divisor and N to be 16 and keep repeating the loop until we hit 1.

However, it is extremely difficult to assign and create variables dynamically in the assembler language thus we boil down the flowchart in figure 4 into figure 3. Notice that the flowchart in figure 3 is a direct application of the flowchart in figure 4. You can consider the flowchart in figure 3 to be a brute-force implementation of the flowchart in figure 4.

We see in the flowchart 3 that we begin by declaring the divisor to be 8 and then calling the modulo subroutine after which we have updated dividend to be dividend = dividend mod 8. Then we update the divisor to be 4 and then call the modulo subroutine essentially computing dividend = dividend mod 8 mod 4 (however in reality we merely compute dividend mod 4), after which we declare the divisor to be 2 and essentially compute dividend = dividend mod 8 mod 4 mod 2 (however actually just computing dividend mod 2) before exiting the subroutine.

The implementation of the flowchart 3 is done in the code shown in figure 14.

We see, just like in the flowchart in figure 3 the code in figure 14 begins by declaring divisor to be 8, then calling modulo subroutine and then saving the current dividend in a variable called answerMod8, after which it declares divisor to be 4, calls



```
;;;;;;;;;;;;;;Modulus Subroutine;;;;;;;;;;;;;;;;;;;;;;

modulus

;;First check mod8 for RC3;;;;

        MOVLF 8, divisor                ;First check for answerMod8
        rcall modulo
        copyRegister answerMod8, dividend   ;at the end you set answerMod8=dividend

;;Now check mod4 for RC2;;;

        MOVLF 4, divisor                ;declare divisor
        rcall modulo                    ;Find dividend mod 4
        copyRegister answerMod4, dividend   ;At the end answerMod4 = dividend


;;Now check for mod2 for RC1;;
        MOVLF 2, divisor
        rcall modulo
        copyRegister answerMod2, dividend

return
```

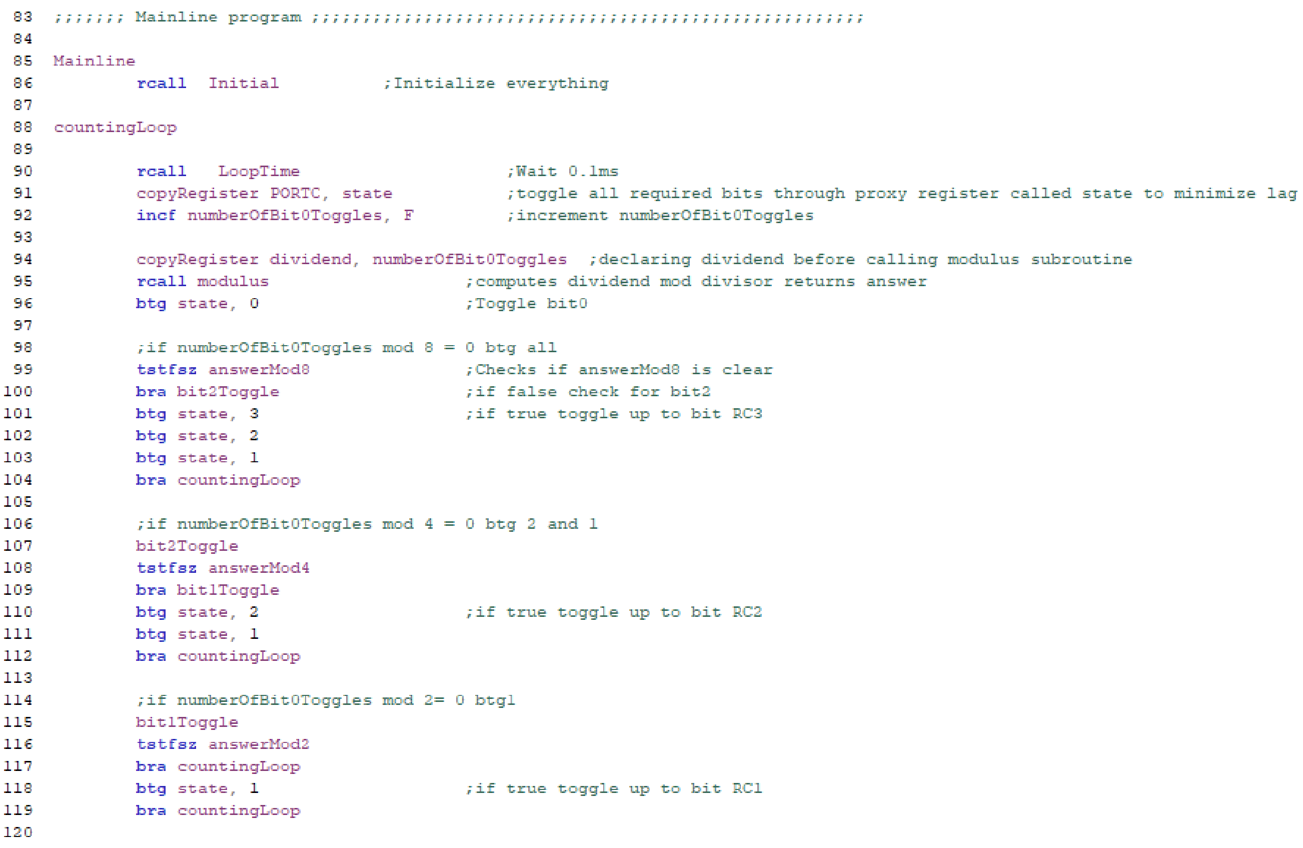
Fig. 14. Modulus subroutine implementation shown in flowchart in figure 3

modulo subroutine and saves the current dividend in a variable called answerMod4, and declares the divisor to be 2, calls the modulo subroutine and saves the current dividend in answerMod2, and fianlly returns.

D. *Putting it all together*

To use the modulo and modulus subroutine to count we refer to flowchart in figure 5. We begin by delaying for 0.1ms by calling *rcall LoopTime* however it is more imporant to think of LoopTime as a 0.1ms delay, it is better to visualize the solution that way. Then you will notice we update PORTC with the proxy register called state, this is done so that all of the bits RC0, RC1, RC2, and RC3 are all updated at once without having a delay between the time taken for RC0 and RC3 to be updated (which is considerable without the proxy register). Then we update the numberOfBit0Toggles for this loop and assign dividend to be equal to the numberOfBit0Toggles. We could have removed either of the two variables, as for all intents and purposes, numberOfBit0Toggles and dividend variable are exactly the same, however, treating them separately helps us think of the solution in a more clear manner. We know how and why the divdend is changing. Then we call the modulus subroutine which computes dividend mod 8, dividend mod 8 mod 4 (essentially not literally), and dividend mod 8 mod 4 mod 2 (essentially not literally). Then we begin toggling our state registers. We first toggle bit0 of state before checking if answerMod8 is 0. If answerMod8 is indeed 0 we toggle bit3, bit2, and bit1 of state before looping back. If hoever answerMod8 is not 0 we check if answerMod4 is 0, and if it is true we toggle bit2 and bit1 before looping back. If answerMod4 is not 0 then we check if answerMod2 is 0, if answerMod 2 is indeed 0 then we toggle bit1 and then loop back, if answerMod2 is not 0 then we toggle nothing and loop back. In the next loop (after a 0.1ms delay) we update PORTC by copying the binary value in the state register into PORTC and repeat infinitely.

The code for the flowchart in figure 5 is shown in figure 15. Notice the only difference between the flowchart and the code is that we have replaced the delay for 0.1ms block with *rcall LoopTime* in line 90, however, they are both identical. Further, notice we have used a brute-force method to toggle the bits of state register in lines 101-103, 110-11, and 118. Since, the number of bits we are dealing with are so small it is easier to brute-force a solution rather than make another subroutine. The output waveform for mainline program in 15 is shown in figure 16.



```
;;;;;;; Mainline program ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

Mainline
        rcall  Initial          ;Initialize everything

countingLoop

        rcall   LoopTime                    ;Wait 0.1ms
        copyRegister PORTC, state           ;toggle all required bits through proxy register called state to minimize lag
        incf numberOfBit0Toggles, F         ;increment numberOfBit0Toggles

        copyRegister dividend, numberOfBit0Toggles  ;declaring dividend before calling modulus subroutine
        rcall modulus                   ;computes dividend mod divisor returns answer
        btg state, 0                    ;Toggle bit0

        ;if numberOfBit0Toggles mod 8 = 0 btg all
        tstfsz answerMod8               ;Checks if answerMod8 is clear
        bra bit2Toggle                  ;if false check for bit2
        btg state, 3                    ;if true toggle up to bit RC3
        btg state, 2
        btg state, 1
        bra countingLoop

        ;if numberOfBit0Toggles mod 4 = 0 btg 2 and 1
        bit2Toggle
        tstfsz answerMod4
        bra bit1Toggle
        btg state, 2                    ;if true toggle up to bit RC2
        btg state, 1
        bra countingLoop

        ;if numberOfBit0Toggles mod 2= 0 btg1
        bit1Toggle
        tstfsz answerMod2
        bra countingLoop
        btg state, 1                    ;if true toggle up to bit RC1
        bra countingLoop
```

Fig. 15. Mainline program for problem 4

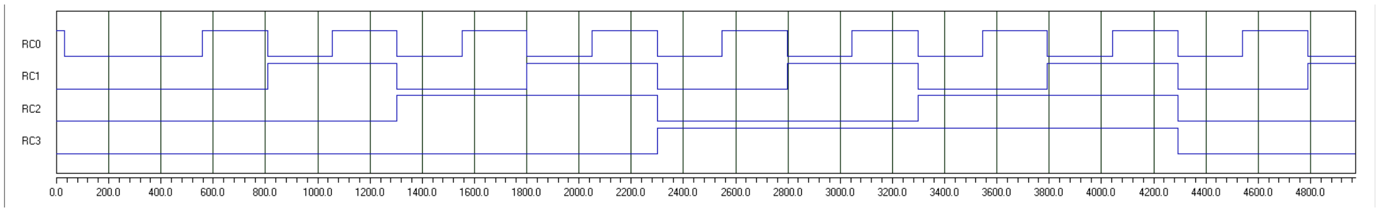


Fig. 16. Waveform for counting from 0 to 15 as per mainline in figure 15