CONTENTS

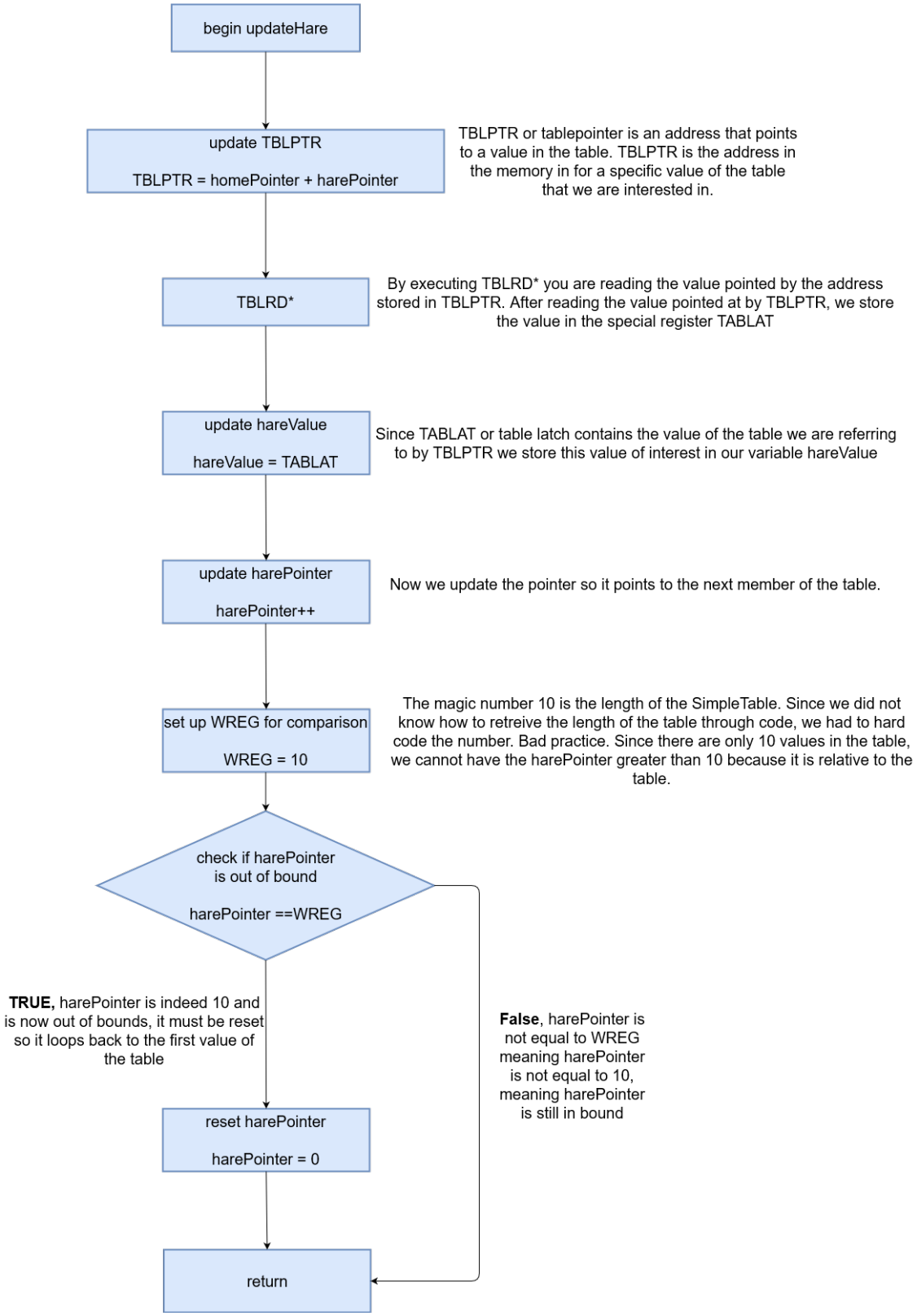

Fig. 1. Flowchart describing the subroutine for updateHare

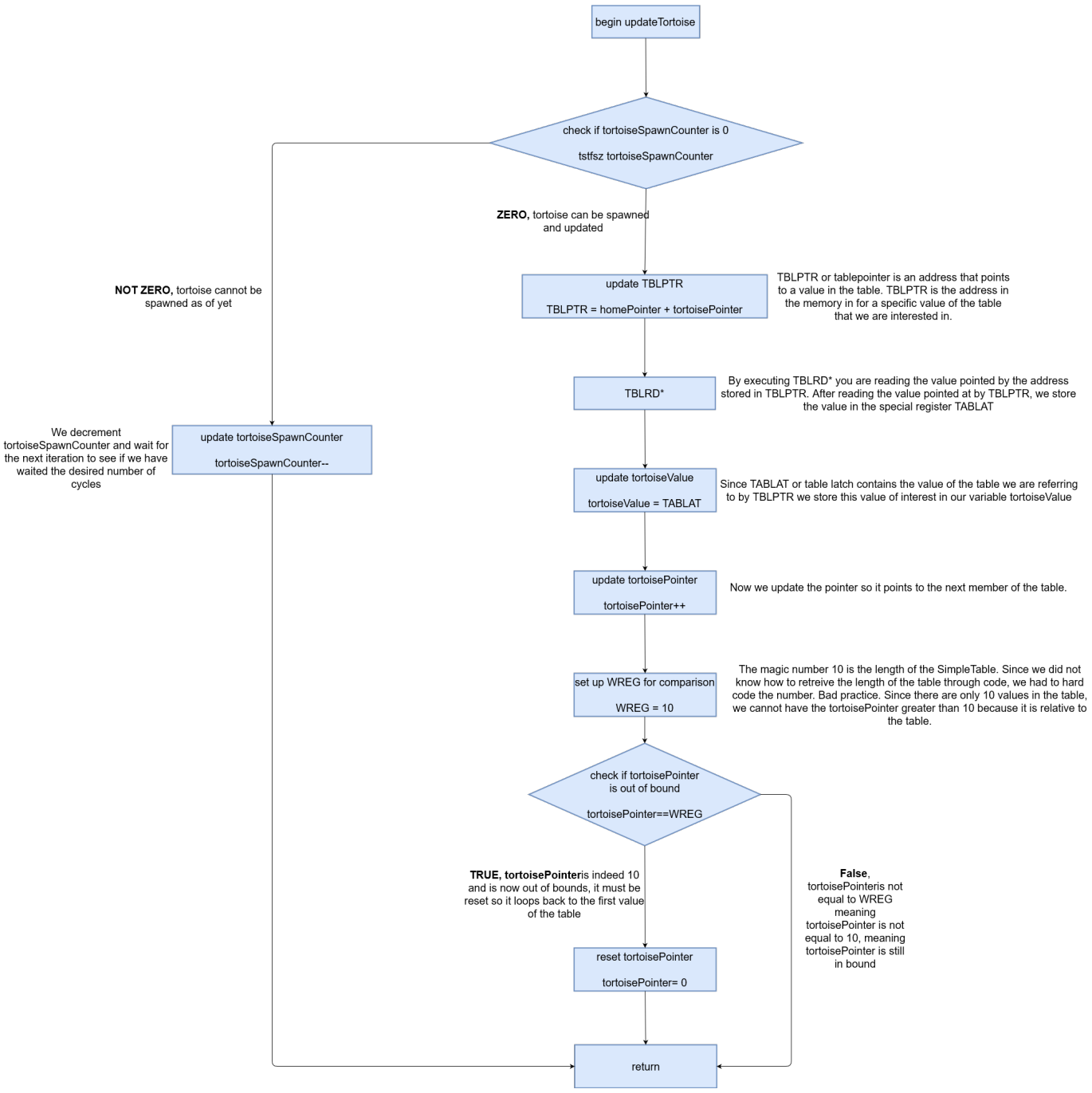

Fig. 2. Flowchart describing the subroutine for updateTortoise

Exp 5 - Discrete-Time Series Averaging Filter

Anas Ashraf

I. Problem Outline

We are given an array, x, of arbitrary length. Let us define the number of elements in this array to be N and place the condition on it that $N \in \mathbb{N}$. Let n be a number defined $n \in [0, N]$. Using the notation $x[n]$ we are referring to the nth element of the array. Let k be an arbitrary constant defined $k \in \mathbb{N}$. We wish to find the resulting array $y[n]$ for particular values of k when:

$$y[n] = \frac{1}{2}x[n] + \frac{1}{2}x[n-k] \quad \forall n \tag{1}$$

Thus we wish to find all the values of $y[n]$ at a given value of k for a given array $x[n]$. Further, in this problem we add the constraint that $y[n]$ must have an infinite length. If we wish to compute $y[\gamma]$ such that $\gamma > N, \gamma \in \mathbb{N}$, then we do so as such:

$$y[\gamma] = \frac{1}{2}x[\gamma \ mod \ N] + \frac{1}{2}x[(\gamma \ mod \ N) - k] \tag{2}$$

Thus, we see that the sizes of arrays x and y are not comparable.

II. Important Considerations

A. On the magnitude of of the offset

The offset is defined as the constant k found in the second term in equation 1. Let k = 3 and let the array we are considering have one hundred elements or $N = 100$. We will notice that if we wish to compute $y[n]$ for a few n then:

$$\begin{aligned} y[5] &= \frac{1}{2}x[5] + \frac{1}{2}x[2] \\ y[6] &= \frac{1}{2}x[6] + \frac{1}{2}x[3] \\ y[7] &= \frac{1}{2}x[7] + \frac{1}{2}x[4] \\ y[8] &= \frac{1}{2}x[8] + \frac{1}{2}x[5] \\ &\dots \end{aligned}$$

You will notice for $y[5]$ we needed $x[5]$ and we will need $x[5]$ again for $y[8]$. Thus we must hold onto $x[5]$ for atleast three cycles. Further, you will notice that we must also hold $x[3]$, $x[4]$, and $x[5]$ while we are computing $y[5]$ so we should not discard them. Thus we need to have a memory buffer of at least four bytes or $k + 1$. Now what happens if we let k = 1,000. You will notice we need to be holding at least 1001 bytes of information. Thus, given the constraint of only having to work with two terms, it would be a far smarter idea to not hold all the values in RAM but to only hold pointers that point to the actual terms in RAM. This makes makes the cost of storage for any k be $\mathcal{O}(1)$ instead of $\mathcal{O}(k)$. Thus, instead of working with the actual values of the array, that is $x[n]$ we will work with the index of the array and use the index to point to the actual value of the array.

B. Reducing the number of operations

Division is a costly operation. Thus we will factor the equation shown in equation 1 so that division is only performed once. This will result in the equation:

$$y[n] = \frac{1}{2}(x[n] + x[n-k]) \tag{3}$$

Further, you should notice that since we are only dividing by two we can compute this with a simple right logical shift. Here we are assuming that the sum of the two values is an even number so we do not have to worry about a floating point error.

```
187  SimpleTable
188  db 0,50,100,150,200,250,200,150,100,50
```

Fig. 3. The values of the array $x[n]$ being loaded into memory

```
107          MOVLF upper SimpleTable,TBLPTRU
108          MOVLF high  SimpleTable,TBLPTRH
109          MOVLF low   SimpleTable,TBLPTRL
```

Fig. 4. Initializing the values for TBLPTR, TBLPTRH, TBLPTRL, and TBLPTRU

C. Reading the array efficently

The array of values that we are working with, $x[n]$, is defined for our particular problems as:

$$x[n] = [0, 50, 100, 150, 200, 250, 200, 150, 100, 50] \tag{4}$$

Notice in equation 4 we have an array of 10 entries and it is (almost) symmetric about the value 250. We load this array into our PIC microcontroller as shown in figure 3.

Further, in figure 4, we initialize the values of the the pointers TBLPTR, TBLPTRH, TBLPTRL, and TBLPTRU which stand for table pointer, table pointer high, table pointer low, and table pointer upper respectively. The most important special register is TBLPTR because it points to the beginning of the array defined in figure 3.

Reading of the table is an expensive operation and should be avoided as much as possible. This is why, as you will later see, our solution has removed the code shown in figure 4 from the mainline. The reading of the table is only done once in the initialization subroutine and never again during runtime. Doing it this way is more efficient. If we had an array or table with one million values we cannot afford to be reading it every time we get to the end of the millionth entry.

D. On the need for two separate/independent pointers

The consequence of equation 2 is that we cannot have the same pointer variable for the first term $x[n]$ and the second term $x[n-k]$. The two terms must have their own special variable pointers and here is why. Consider equation 4, whereby, after reading the table using the code in figure 4 we get that the TBLPTR has the value ϵ. This means that the address that the first value in the array $x[n]$ is stored in the memory address of value ϵ. If we use the same pointer for both $x[n]$ and $x[n-k]$, we will notice that near the end values of $x[n]$, before we loop around $x[n]$, we compute the following operations:

$$\begin{aligned}
&\vdots \\
y[\epsilon+9] &= \frac{1}{2}(x[(\epsilon+9) \textit{ modulo } N] + x[(\epsilon+9) \textit{ modulo } N-1]) = \frac{1}{2}(x[\epsilon+9] + x[\epsilon+9-1]) \\
y[\epsilon+10] &= \frac{1}{2}(x[(\epsilon+10) \textit{ modulo } N] + x[(\epsilon+10) \textit{ modulo } N-1]) = \frac{1}{2}(x[\epsilon] + x[\epsilon-1]) \\
&\vdots
\end{aligned}$$

You will notice, since the second term points relative to the first term, we have the second term point below ϵ wherein ϵ is the address of the first entry in $y[\epsilon+10]$. Thus the second term is out of bound and points outside of the array, returning some random value in the memory. To mitigate this problem, we must have two separately defined pointers for both the terms that are mutually exclusive to the other. Our solution is to have the pointer for each term read the table independently and only delay the creation of the second pointer by k cycles. This allows for multithreading and running the code more efficiently if more cores/threads are available. Thus, we kill two birds with one stone, we stop the out of bound error, and in doing so we also make the code more parallelizeable where we can create a parallel instance of the code on a different thread that scales linearly with the number of terms in equation 1. If and only if the number of terms in equation 1 is less than the number of threads available, we will have a computation cost of $\mathcal{O}(1)$ instead of $\mathcal{O}(\mu)$ where μ is the number of terms modulo with the number of threads available.

III. Solution Implementation

A. The Mainline Program

We can get a rough understanding of the solution by looking at the mainline program shown in figure 5. We first begin by executing *MOVLF 5, offset*. This instruction moves the value 5 into the offset variable. The offset variable is the value of

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

Mainline
        MOVLF 5, offset     ;HUMAN DEFINED, value of the offset or the k in x[n-k]
                            ;change as needed by the problem

        rcall  Initial          ;Initialize everything
        mainLoop
            rcall updateHare
            rcall updateTortoise
            rcall updateAnswer
        bra mainLoop
```

Fig. 5. Mainline program of our final solution for a general $y[n]$ as per equation 1

the k in the second term in equation 1. So in this instance k or offset has been set to 5. This can be easily changed before execution of the program. Then we call the initial subroutine and then jump into our mainLoop. The mainLoop only has three major subroutines to execute: updateHare, updateTortoise, and updateAnswer. Each subroutine will be explained later on. However, updateHare, updates or fetches the value for the first term in equation 1. So updateHare computes the value of $x[n]$ for a given n. updateTortoise updates the value of the second term in equation 1. So it updates $x[n-k]$ for a particular n and the k is defined by offset. Finally, the updateAnswer subroutine computes the sum of the two terms and then divides them by 2 through a logical shift as per equation 3.

B. The Initial Subroutine

The initial subroutine is shown in figure 6. There is however a section of code the initializes PORTA, PORTB, and other special registers but that has been omitted as it is not important to our solution. We begin the important part of the initial subroutine by reading the table and initializing the value of table pointers. Notice, the first part has also been shown in figure 4. Note that we only read the table once in the initial subroutine and don't read it again during the entirety of the mainline program. Our next instruction is *copyRegister homePointer, TBLPTR*. We set up a constant called homePointer which points to the beginning of the SimpleTable. TBLPTR points to the beginning of SimpleTable after we initialize the table and load it into memory. We will never change homePointer, thus it is a constant, but we will change TBLPTR as needed to point to different values over time.

Further, in the mainline we defined the constant offset. By executing the instruction *copyRegister tortoiseSpawnCounter, offset*. This copies the value of the constant offset into our variable tortoiseSpawnCounter which will be changed as needed. It will countdown to 0 after which we will spawn the tortoise or compute the second term in equation 1. Lines 123-127 are simply setting the other variables than we use in our other subroutines to 0. It is standard practice. Lastly, notice that copyRegister is our custom defined macro that is shown in figure 7.

We can think of the macro in figure 7 as assignment. We assign the value of B to A. We do this by first moving B to WREG and then moving WREG to A. Quite a simple macro.

C. The updateHare subroutine

The pseudocode representation for reading an array infinitely is shown in listing 1. We begin by first loading the values of the array into memory in line 3 and then we load the address of the first element into some constant integer called arrayBeginningPointer. This constant is not permutable and will serve as a reference address to which we can read the array. Next we declare a variable integer called arrayRelativePointer which cycles through the entries in the array. We declare arrayPointer to be the sum of arrayBeginningPointer and arrayRelativePointer so that we can use the memory address to cycle through the array's values. We then read the desired value stored in the address in arrayPointer and store the value stored in the address into a variable called arrayValue. Then we increment arrayRelativePointer. Finally, we compute the modulo of arrayRelativePointer to make sure it doesn't point out of bounds. By computing the modulo of arrayRelativePointer we ensure that we loop back around in reading the values of the array.

We can further breakdown the code shown in listing 1 into a general flowchart from which we will obtain our eventual assembler code. To understand the updateHare subroutine code shown in figure 8 we must look at the complementary flowhchart shown in figure 1. In the flowchart shown in figure 1 we will notice that we begin by updating the TBLPTR register. We

```
104          ;My initializations
105
106          ;Think of this as initializing the table values and loading them into memory
107          MOVLF upper SimpleTable,TBLPTRU
108          MOVLF high  SimpleTable,TBLPTRH
109          MOVLF low   SimpleTable,TBLPTRL
110
111          ;This declares constant homePointer which is the address of the first
112          ;entry in the SimpleTabl, in this specific example we have
113          ;homePointer = 190 but this solution is more robust by not defining it
114          ;explicitly
115          copyRegister homePointer, TBLPTR    ;;homePointer = TBLPTR
116
117          ;Here we define the variable tortoiseSpawnCounter based on our offset or the
118          ;k in x[n-k] which is defined in the mainline program
119          copyRegister tortoiseSpawnCounter, offset   ;variable tortoiseSpawnCounter
120          ;The tortoiseSpawnCounter counts down time until we can spawn and use
121          ;the second counter for the offset term x[n-k]
122
123          MOVLF 0, harePointer
124          MOVLF 0, hareValue
125          MOVLF 0, tortoiseValue
126          MOVLF 0, tortoisePointer
127          MOVLF 0, answer
128    return
```

Fig. 6. Initial subroutine

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

Fig. 7. Macro definition for copyRegister

```
// This pseudo-code is for reading an array and storing the values in some arbitrary variable

array = [0,50,100,150,200,250,200,150,100,50]   //arbitrary array
const int  arrayBeginningPointer = address(array[0])      //stores the address of the 0th element
int arrayRelativePointer = 0;

int arrayPointer = arrayBeginningPointer + arrayRelativePointer;
read[arrayPointer], store[arrayValue];  //reads memory address arrayPointer and stores value in arrayValue
arrayRelativePointer++; //increment arrayRelativePointer
arrayRelativePointer = arrayRelativePointer modulo len(array);//loop to keep reading the array
```

Listing 1: Pseudocode for reading an array infinitely

update the TBLPTR register by setting it to the sum of homePointer (which points to the first value in SimpleTable) and the harePointer (which points to a particular entry in the SimpleTable). Notice, harePointer should never exceed the length of the SimpleTable which in our particular instance is 9. homePointer is a constant and is never permuted. After updating our TBLPTR we read the table using the address stored in TBLPTR. When we read the table using *TBLRD** we store the value in the address of TBLPTR into the special register TABLAT. We then move this value from TABLAT into our hareValue

```
132  ;;;;;;;;;;updateHare subroutine;;;;;;;;;;;;;;;;;;
133  updateHare
134      addTwoRegisters TBLPTR, homePointer, harePointer;TBLPTR = homePointer + harePointer
135      TBLRD*                                          ;read value in address TBLPTR and store value in TABLAT
136      copyRegister hareValue, TABLAT                  ;set hareValue = TABLAT
137      incf harePointer, F                             ;increment harePointer and store value in harePointer
138
139      MOVLF 10, WREG                                  ;This magic number 10 is the length of SimpleTable
140      cpfseq harePointer                              ;checks if harePointer points outside of array
141      return                                          ;not equal, harePointer is valid, return
142      MOVLF 0, harePointer                            ;resets harePointer to 0 so it points inside array
143      return
```

Fig. 8. The updateHare subroutine

variable. This is because TABLAT is updated everytime you read the table and since we will need to read the table again for the second term in equation 1 we need to move the value out of TABLAT as it will be written to again. After we update hareValue we update harePointer to see if our addition has caused it to be greater than 9. We check if harePointer == 10. This magic number, 10, is the length of the SimpleTable array. The code solution would be more robust if we could somehow automatically retrieve the size of the SimpleTable array and not hard code it but since we do not know how to implicitly retrieve the length of SimpleTable, we must hardcode a solution. If our harePointer is indeed equal to 10, this means it now points outside of the SimpleTable and must be reset. We reset it back to 0 so the sum of homePointer and harePointer points to the beginning of the array of SimpleTable. Thus, after harePointer is detected out of bounds, we make it loop back to the beginning. If however, we did not find harePointer to be out of bounds after the increment instruction, we simply return.

D. The updateTortoise subroutine

The updateTortoise subroutine is a superset of the updateHare subroutine. Thus, if you understand the updateHare subroutine, you can easily understand the updateTortoise subroutine by a mere extension of the logic. Both the subroutines are extremely similar, and if we had dynamic variable assignment, we could condense both the subroutines into a single subroutine, however, since we do not have dynamic variable assignment we must use two separate subroutines. Further, as we have shown previously, we need two separate pointer variables to be mutually independent of the other thus it necessitates the use of dynamic variable assignment (or simply two separate subroutines).

The updateTortoise subroutine shown in figure 9 can be better understood by looking at the flowchart in figure 2. We begin the flowchart by first checking if tortoiseSpawnCounter is 0 yet. If it is not zero, we decrement the tortoiseSpawnCounter. Recall that offset, updateTortoiseCounter, and k in the second term of equation 1 are the same thing. The tortoiseSpawnCunter simply counts down the number of cycles until we have waited for k cycles. After the k cycles have elapsed we compare to see if tortoiseSpawnCounter is 0 and this indeed returns true, meaning after k cycles tortoiseSpawnCounter is 0. We then proceed to update tortoiseValue the same way as we would for hareValue. The only thing different is the name of the variables, the core logic is exactly identical.

We begin by updating TBLPTR by defining it to be the sum of the homePointer and the tortoisePointer so it points to a particular value in SimpleTable. Then using TBLPTR we read the table and store the read value in TABLAT. We move the value in TABLAT into tortoiseValue and increment tortoisePointer. We then check to see if our incrementation of tortoisePointe has put it out of bounds. If tortoisePOinter is 10 due to increment (magic number coming from the length of the SimpleTable array) then we reset tortoisePointer back to 0 before returning. Otherwise we simply return without alterting the value of tortoisePointer.

E. The updateAnswer subroutine

Finally, the updateAnswer subroutine computes the $y[n]$ in equation 1 for an offset or k defined in the mainline. We begin by executing the instruction *addTwoRegisters answer, tortoiseValue, hareValue* which is the same as adding $x[n-k]$ and $x[n]$ respectively and storing it in the answer variable. Then we do a right rotation or right logical shift with a carry bit. In our instance since our largest addition for any k will never be greater than 511, we will never have an overflow. However, this is good practice so we will include the carry bit in our rotation.

IV. Solutions for a set of particular k

The code output for a set of k such that $k \in \{1, 2, 3, 4, 5\}$ is shown in table I. The k here is the same k as in the equation 1. The complementary graph is shown in figure 11. We can see in figure 11 that as we increase k, the output waveform becomes

```
147  ;;;;;;;;;updatetortoise subroutine;;;;;;;;;;;;;;;;;;;;;;;;
148
149  updateTortoise
150      tstfsz tortoiseSpawnCounter         ;checks if time to spawn and update tortoise has arrived
151      bra updateTortoiseSpawnCount        ;not zero, not yet time to spawn tortoise
152
153      ;Here tortoiseSpawnCounter is indeed zero, time to spawn tortoise and update value
154      addTwoRegisters TBLPTR, homePointer, tortoisePointer ;TBLPTR = homePointer + tortoisePointer
155      TBLRD*
156      copyRegister tortoiseValue, TABLAT
157      incf tortoisePointer, F
158
159      MOVLF 10, WREG
160      cpfseq tortoisePointer
161      return
162      MOVLF 0, tortoisePointer
163      return
164
165
166
167      updateTortoiseSpawnCount
168          decf tortoiseSpawnCounter, F
169          return
```

Fig. 9. The updateTortoise subroutine code

```
173  ;;;;;;;;;;;;;update Answer;;;;;;;;;;;;;;;;;;;;;;;;;;;;
174
175  updateAnswer
176       addTwoRegisters answer, tortoiseValue, hareValue
177       rrcf answer, F           ;divide answer by 2 and update answer register
178       return
```

Fig. 10. The updateAnswer subroutine

more and more flatter. At $k = 5$ the output waveform eventually becomes a straight line after the transient phase has subsided. The transient phase subsides at 6 time units in figure 11.

TABLE I
COMPARISON OF $x[n]$ AND $y[n]$ FOR VARIOUS K WHERE $y[n]$ IS DEFINED IN EQUATION 1

$x[n]$	$y[n]$ @k=1	$y[n]$ @k=2	$y[n]$ @k=3	$y[n]$ @k=4	$y[n]$ @k=5
0	0	0	0	0	0
50	25	25	25	25	25
100	75	50	50	50	50
150	125	100	75	75	75
200	175	150	125	100	100
250	225	200	175	150	125
200	225	200	175	150	125
150	175	200	175	150	125
100	125	150	175	150	125
50	75	100	125	150	125
0	25	50	75	100	125
50	25	50	75	100	125
100	75	50	75	100	125
150	125	100	75	100	125
200	175	150	125	100	125
250	225	200	175	150	125
200	225	200	175	150	125
150	175	200	175	150	125
100	125	150	175	150	125
50	75	100	125	150	125
0	25	50	75	100	125
50	25	50	75	100	125
100	75	50	75	100	125
150	125	100	75	100	125
200	175	150	125	100	125
250	225	200	175	150	125
200	225	200	175	150	125
150	175	200	175	150	125
100	125	150	175	150	125
50	75	100	125	150	125
0	25	50	75	100	125

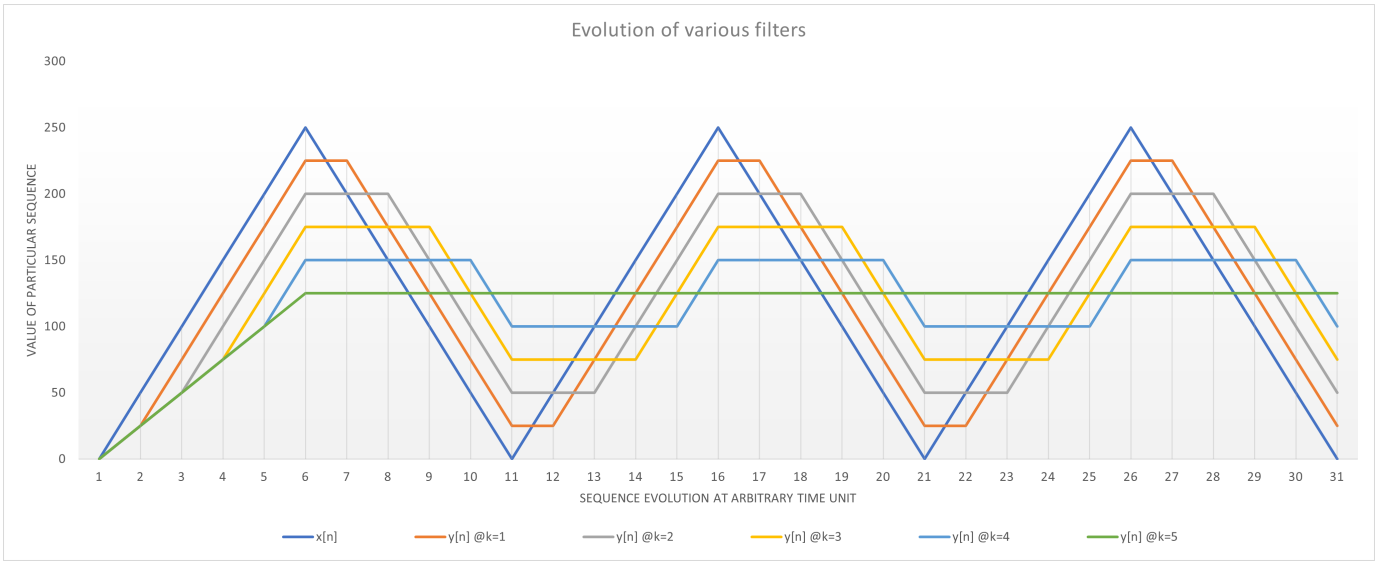

Fig. 11. Graphical representation of table I