Student Name: Prathamesh Sai
Student ID: 19314123

CSU11022 Introduction to Computing II

End-of-Semester Assignment

Question 1:

Design and write an ARM Assembly Language subroutine, isMagic(...), that will determine whether a square two-dimensional array in memory is a Magic Square.

(i) Detailed documentation, including your interpretation of the problem and an explanation of the approach you have used. If you have decomposed the problem into supporting subroutines, you should explain this. Your documentation should be supported by pseudocode, examples and diagrams.
[20 marks, max: 1,000 words]

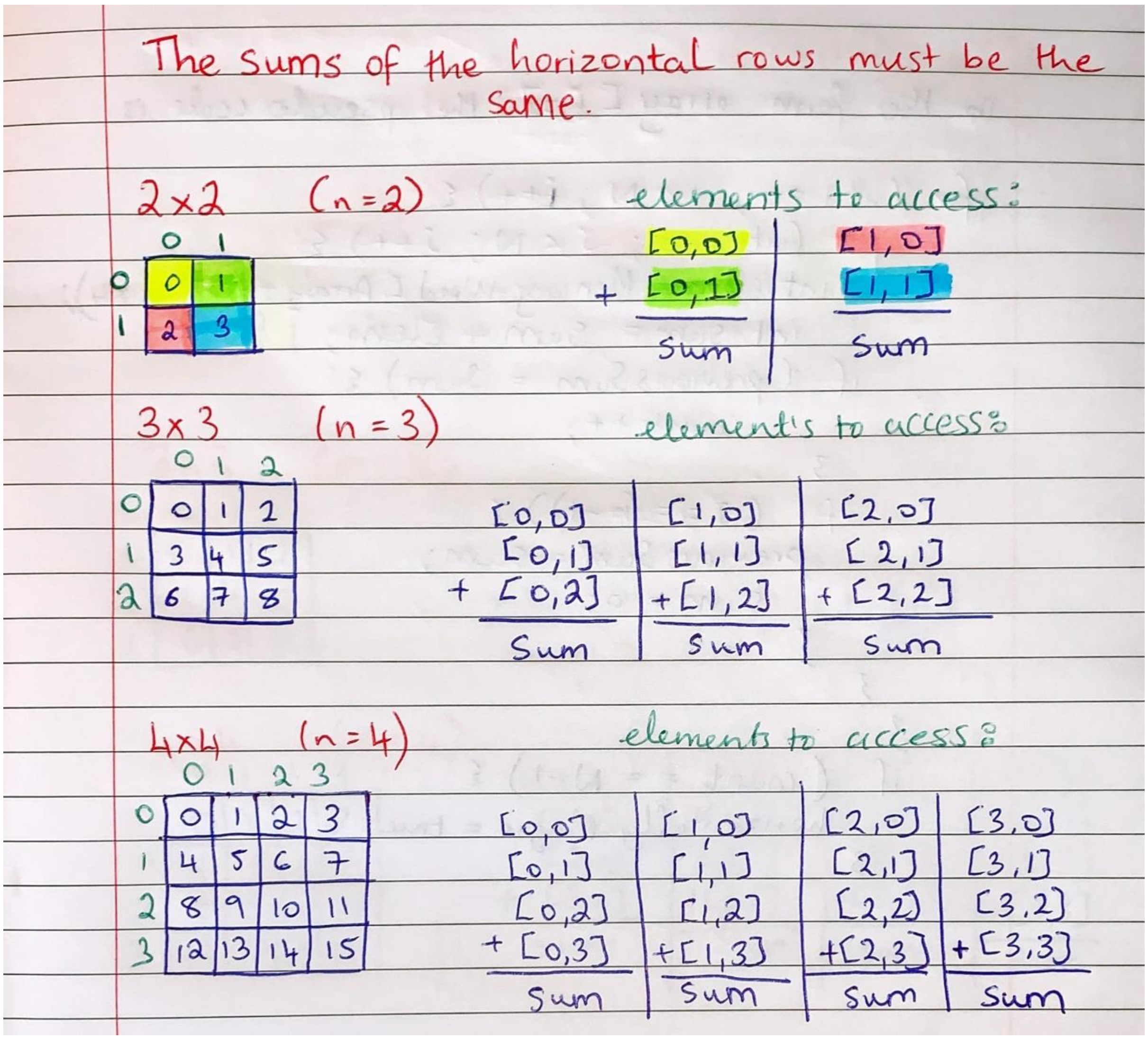
I figured out a few steps to determine if an N x N array is a magic square or not. I came up with these 3 steps:

1. The sums of the horizontal rows must be equal.
2. The sums of the vertical columns must be equal.
3. The sums of the elements across the main diagonal must be equal to the sum of elements across the off-diagonal.

I thought it would be better to draw out diagrams of the concepts before getting straight into µVision as it would lay the foundation for my pseudo-code afterwards.

Student Name: Prathamesh Sai
Student ID: 19314123

Step 1: Check Horizontally (diagram)



The sums of the horizontal rows must be the same.

2x2 (n=2)

$$\begin{array}{c|cc} & 0 & 1 \\ \hline 0 & 0 & 1 \\ 1 & 2 & 3 \end{array}$$

elements to access:

[0,0]	[1,0]
+ [0,1]	[1,1]
sum	sum

3x3 (n=3)

$$\begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 0 & 1 & 2 \\ 1 & 3 & 4 & 5 \\ 2 & 6 & 7 & 8 \end{array}$$

element's to access:

[0,0]	[1,0]	[2,0]
[0,1]	[1,1]	[2,1]
+ [0,2]	+ [1,2]	+ [2,2]
sum	sum	sum

4x4 (n=4)

$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 2 & 3 \\ 1 & 4 & 5 & 6 & 7 \\ 2 & 8 & 9 & 10 & 11 \\ 3 & 12 & 13 & 14 & 15 \end{array}$$

elements to access:

[0,0]	[1,0]	[2,0]	[3,0]
[0,1]	[1,1]	[2,1]	[3,1]
[0,2]	[1,2]	[2,2]	[3,2]
+ [0,3]	+ [1,3]	+ [2,3]	+ [3,3]
sum	sum	sum	sum

I wrote down the elements to be accessed in the 2x2, 3x3, and 4x4 arrays to determine if the sums of the horizontal rows are equal (as highlighted in the 2x2 diagram). This let me figure out a pattern which I could use to my advantage in making the pseudo-code. I noticed that the first element would always be [0,0] in the form [i, j] where i indicated the index of the rows, and j indicated the index of the columns. We would access [i, j] in this case, then the index of the column would be incremented. This continued until the column was equal to (n-1) (the end of the row). After this, the index of the rows would be incremented. This pattern happened (n-1) times. I realised that these series of operations were similar to the format of two for-loops.

Student Name: Prathamesh Sai
Student ID: 19314123

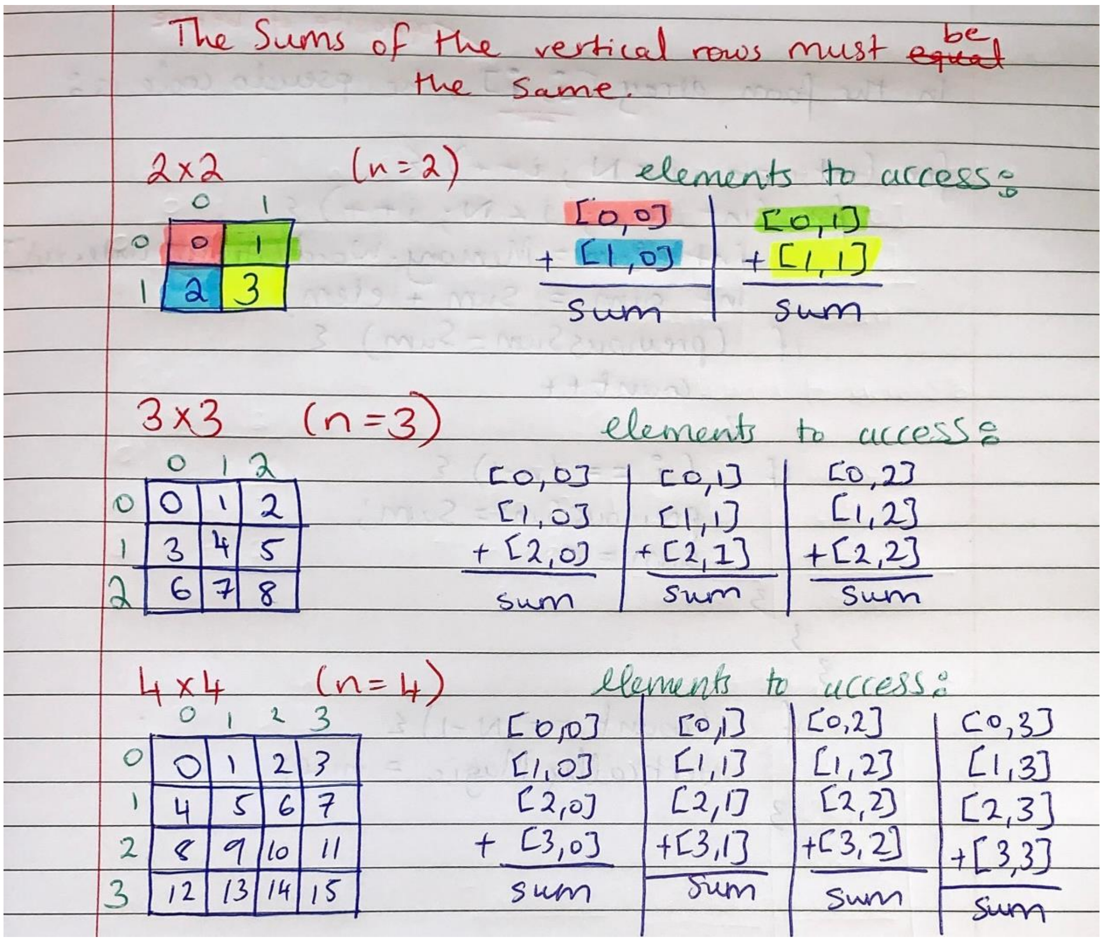
Step 1: Check Horizontally (pseudo-code)

//In the form array[i, j] in an N x N array, the pseudo-code to check horizontally is:

```
public Boolean isHorizontallyMagic(N, array[]) {
boolean horizontallyMagic = false;
int sum = 0;
int count = 0;
int previousSum = 0;

        for (int i = 0; i < N; i ++) {
                for (int j = 0; j < N; j++) {
                        int elem = Memory.Word[Array + (index * 4)]; // get element
                        sum = sum + elem;

                        if (previousSum == sum) {
                                count ++;
                        }

                        if (j == (N - 1)) {
                                previousSum = sum;
                                sum = 0;
                        }
                }
        }

        if (count == (N - 1)) {
                horizontallyMagic = true;
        }
return horizontallyMagic;
}
```

The pseudo-code was easy to figure out with reference to the diagram I made earlier. I figured out that you need two for-loops, one for incrementing the row, and another for incrementing the column. The inner-nested for-loop would loop through the columns and the outer for-loop would loop through the rows. After each row, we needed to store the summation of that row into a register to compare it later. Then it would move on the next row. At the end of a row, it will compare that row's summation to the previous summation (if there exists one). If they are equal, we can increment a counter by one. It will continue to do so until both the loops finish. If the summation of each row is the same, the counter must end up being equal to (n-1) since the rows will be compared (n-1) times; the previous sum must be equal to the current sum for both the rows in question to be "magic".

Step 2: Check vertically (diagram)



Similarly, to check vertically, I did the same thing to allow me to figure out a pattern which I could use when making the pseudo code. I could see that the first element would always be [0,0] in the form [j, i] where i indicated the index of the columns, and j indicated the index of the rows. Interestingly, I noted from the diagrams that this method was very similar to checking horizontally, however now we want to access array [j, i] instead of array [i, j]. Subsequently, we would access the element, which is array [j, i] in this case, and the index of the row would be incremented. This happened until the row was equal to (n-1) (the end of the column). After this, the index of the column would be incremented. This pattern would happen (n-1) times. I quickly realised this pattern was similar to the format of two for-loops too.

Student Name: Prathamesh Sai
Student ID: 19314123

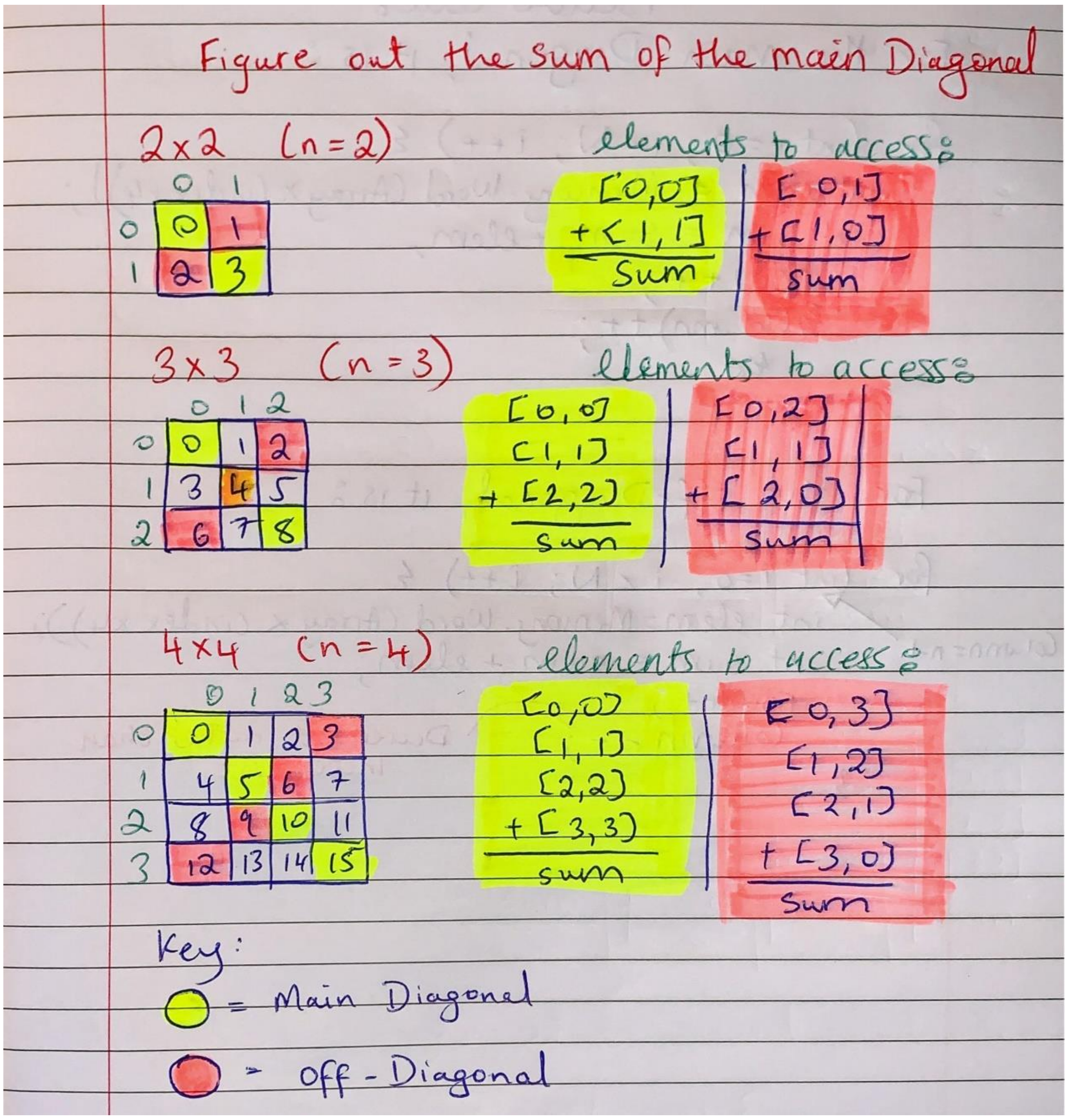
Step 2: Check vertically (pseudo-code)

//In the form array[j, i] (instead of [i, j]) in an N x N array, the pseudo-code to check vertically is:

```
public Boolean isVerticallyMagic(N, array[]) {
boolean verticallyMagic = false;
int sum = 0;
int count = 0;
int previousSum = 0;

       for (int i = 0; i < N; i ++) {
              for (int j = 0; j < N; j++) {
                     int elem = Memory.Word[Array + (index * 4)]; // get element
                     sum = sum + elem;

                     if (previousSum == sum) {
                            count ++;
                     }

                     if (j == (N - 1)) {
                            previousSum = sum;
                            sum = 0;
                     }
              }
       }

       if (count == (N - 1)) {
              verticallyMagic = true;
       }

return verticallyMagic;

}
```

Once again, the pseudo code was straightforward to figure out with reference to the diagram I made earlier. I knew that I needed two for-loops; one for incrementing the row, and another for incrementing the column. The inner-nested for-loop would loop through the rows and the outer for-loop would loop through the columns. After each column, we needed to store the summation of that column into a register to compare it later. Then it would move on the next column. When it reaches the end of a column, it will compare that column's summation to the previous summation (if there exists one). If they are equal, we can increment a counter by one. It will continue until both the loops finish. If the summation of each column is the same, the counter must end up being equal to (n-1) since the columns will be compared (n-1) times.

Step 3: Check Diagonally (diagram)



Similarly, I wrote down all the elements that needed to be accessed in a 2x2, 3x3, and 4x4 array to check if the summation of the elements along the main diagonal (yellow) is equal to the summation of the elements along the off-diagonal (red). I found a pattern for the main diagonal, where the element starts off at [0,0] in the form array [i, j] and then both i and j are incremented after each element is accessed. On the other-hand for the off-diagonal, it was similar, but we needed to start off at [0, (n-1)] and then increment i while decrementing j after each element was accessed.

Student Name: Prathamesh Sai
Student ID: 19314123

Step 3: Check Diagonally (pseudo-code)

//In the form array[i, i], where i represents the rows and the columns,
//the pseudo code for the **main diagonal** is:

```
public int isMainDiagonallyMagic(N, array[]) {
int sum = 0;
int previousSum = 0;

      for (int i = 0; i < N; i ++) {
            int elem = Memory.Word[Array + (index * 4)]; // get element
            sum = sum + elem;
            row++;       // i++;
            column++;    // i++;
            }
return sum;
}
```

//In the form array[i, j], where i represents the row and j represents the column,
//the pseudo code for the **off-diagonal** is:

```
public int isOffDiagonallyMagic(N, array[]) {
int sum = 0;
int previousSum = 0;
int j = N-1 // Starting element needs J to be equal to N-1.

      for (int i = 0; i < N; i ++) {
            int elem = Memory.Word[Array + (index * 4)]; // get element
            sum = sum + elem;
            row++;       // i++;
            column--;    // j--;
            }
return sum;
}
```

Unlike the other steps, I realised that only one for-loop was needed for each diagonal. Inside each of the for-loops, we would access the element required depending on whether we want the elements across the main diagonal or the off-diagonal. We calculate the summation of the diagonal and then we increment the row, and either increment or decrement the column based on whether we are accessing the elements across the main diagonal or the off-diagonal. **These subroutines are designed to output the summation of the two diagonals. Then, in the isMagic subroutine, I can compare the output of both inner-subroutines and determine if the summation of both the diagonals are equal**. I decided to do this so I could keep the code clean and distinct so it would improve understandability rather than cramming the code for both the main diagonal, off diagonal, and checking if they're equal into one subroutine.

Student Name: Prathamesh Sai
Student ID: 19314123

(ii) Your ARM Assembly Language subroutine and a program that tests your subroutine using a number of different Magic Squares. **[25 marks]**

- My ARM subroutine is attached to the assignment question on blackboard as MagicSquare.s as requested.
- At the top of the program, I have included 4 tests to test 4 arrays, two of which are valid magic squares and the remaining two are invalid. In a test scenario, you would test each array one at a time, but I have included the code to test for each of the 4 arrays one-after-each-other for convenience. Hence, when testing one would test each array one at a time (and check if it is a valid/invalid array by checking R0.).
- As specified in the program, R0 will equal 1 if the array being tested is a magic square, and R0 will equal 0 if the array being tested is NOT a magic square.

(iii) A discussion of the performance (or efficiency) of your approach and how it might be improved. **[5 marks, max: 250 words]**

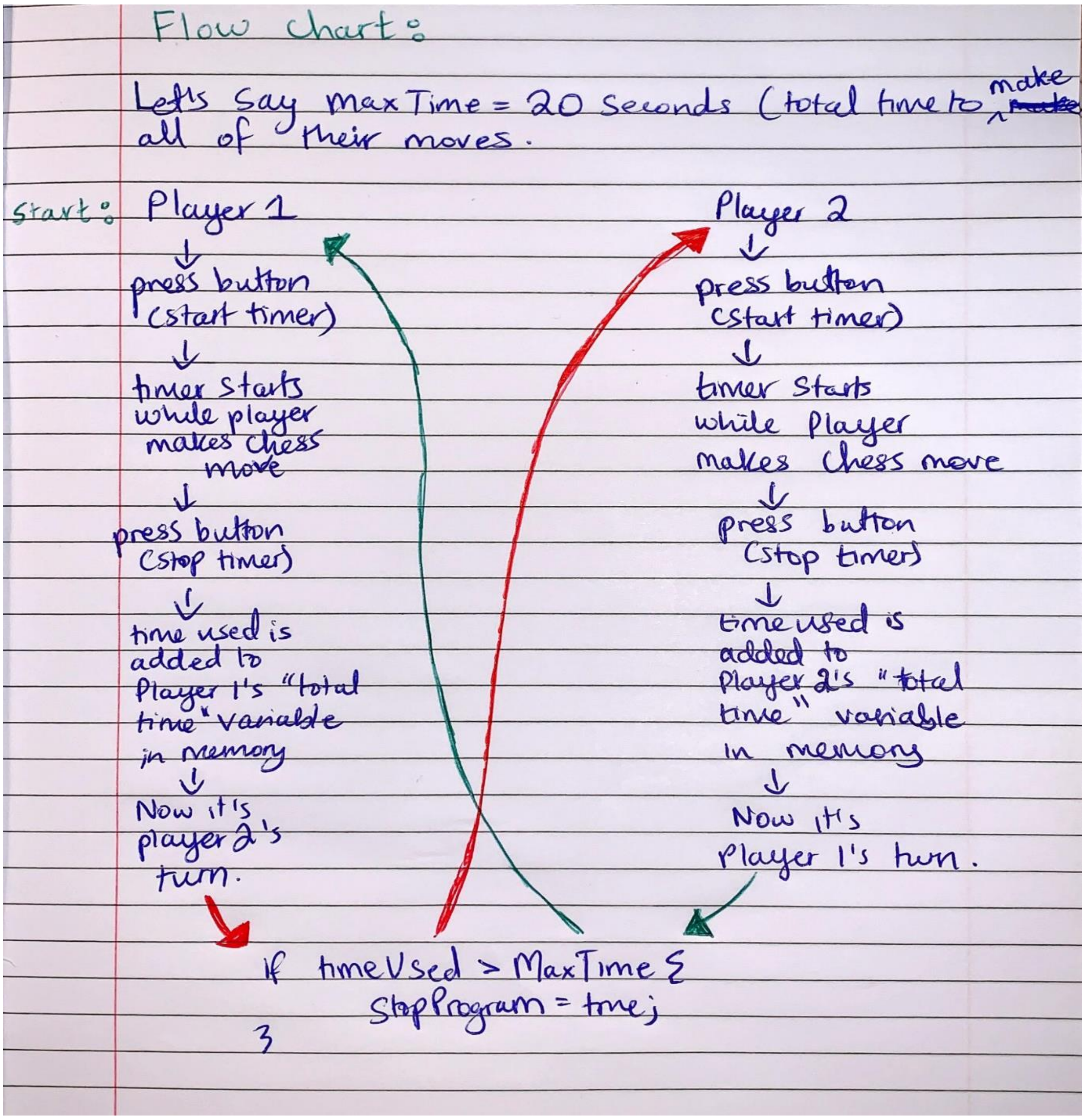
I wanted to put emphasis on the understandability of my program, so I divided the isMagic subroutine into smaller subroutines. The amalgamation of these mini subroutines contributes to the output of the isMagic subroutine. In terms of performance, I stuck to all the necessary rules when using the system stack. For example, I always made sure to pop off everything that I pushed on to the stack; this helped me avoid confusion in the long run. Also, I used the registers suggested by the content in CSU11022 in each subroutine; for example, I used R0-R3 for passing functions into subroutines, and R4-R12 as local variables for each subroutine. Using this structure in designing my subroutines made my approach more efficient and less confusing. In terms of how it might be improved, I could critique my program and say that I could have combined two subroutines for each diagonal in the array into one subroutine to make the code smaller. In retrospect, it would be possible, but I decided to make it into two different subroutines for understandability and ease of use. However, to make it more efficient, I would have combined the two subroutines for the diagonals into one subroutine and not have to push/pop registers off the stack twice.

Question 2:

<u>Design and write an ARM Assembly Language program that will implement a Chess Clock using an LPC2468 microcontroller. Your implementation must make use of GPIO and interrupts.</u>

(i) Detailed documentation, including your interpretation of the problem and an explanation of the approach you have used. Your documentation should be supported by pseudo-code, examples and diagrams. **[20 marks, max: 1,000 words]**

Since a chess clock is a sequential type of hardware, I decided to make a flow chart to illustrate my idea for the program. I wanted to make it as simple as possible so it could be as straightforward and efficient as possible, and this **flow diagram** helped immensely:


Flow chart:
Let's say maxTime = 20 seconds (total time to make ~~make~~ all of their moves.
start: Player 1
press button (start timer)
timer starts while player makes chess move
press button (stop timer)
time used is added to Player 1's "total time" variable in memory
Now it's player 2's turn.
Player 2
press button (start timer)
timer starts while player makes chess move
press button (stop timer)
time used is added to Player 2's "total time" variable in memory
Now it's Player 1's turn.
if timeUsed > MaxTime {
StopProgram = true;
}

Explanation of the flow diagram

As you can see above, the program starts with Player 1 in the chess game. Player 1 presses the button down to start the timer and Player 1’s timer will start. Now, Player 1 has time to make their move. Once they are done, they can release the button. When the button is released, the code in the button handler is used. Inside the button handler, the timer stops.

Now, the program must figure out which player stopped the timer. Once that is calculated, we must get the time that has elapsed between the previous two consecutive button pressed (essentially, get the time that Player 1 used to make a move) and add this value to Player 1’s total time. Player 1’s total time can be saved in memory for ease of use. Once all of this is done, we must do a final check; we check if Player 1’s total time is bigger than the maximum time we have allocated when the program begins (In my examples, I used 20 seconds). If Player 1’s total time is bigger than the maximum time, then Player 1 has used too much time overall and has been delaying the game. In our program, we can use some sort of visual indicator to the user(s) to show that this has happened. Otherwise, the program will continue as normal and the same process occurs, but for Player 2.

I realised from this that there was no time when 2 timers would be on at the same time, so I decided to use one timer instead so the program would be the same in terms of the user’s point of view but be more efficient. This was fine but that imposed a new question, how can the program find out who pushed the button previously with one timer?

Variables and how they are used in my program

At the end of my program, I decided to have some test data which essentially act as variables. These included things such as the maximum time each player can use (in my examples, I use 20 seconds for ease of use). Also, I had the total time used by Player 1 and Player 2. However, as I mentioned in the previous paragraph, I needed something to allow the program to calculate which player pressed the button before. This piece of information is needed so the correct player's total time can be added to (for example, if Player 1 just spent 5 seconds on a move, it wouldn't make sense to add that time to Player 2's total time).

To combat this issue, I included a counter which was stored in memory. I figured out that if I increment the count after each player presses the button, the count will always be odd when it's Player 1's turn, and count will be even when it's Player 2's turn. In high-level languages like Java, I could use count%2 (count modulus 2) to find if count is even or not. But µVision did not accept a MOD instruction, so I improvised by using an 'AND' instruction and #1 (which is essentially n-1). By using 'AND' and #1 with count, the answer (i.e. the remainder on division) should equal 0 if it is even and it was Player 2 who pressed the button last. Otherwise, it was Player 1 who pressed the button last.

With this, the program can successfully add the time to the correct Player's total time and the cycle in the flow chart will keep continuing until the game is over when the user presses the stop button or if a Player uses too much time. With this information, I was confident in making pseudo-code.

Student Name: Prathamesh Sai
Student ID: 19314123

Chess Clock (pseudo code)

```
boolean startProgram = true;
boolean buttonPressed = true;
boolean startTimer = true;
int count = 1;
int whichPlayerPlayed = 0;
boolean playerTwoPlayed = false;
boolean playerOnePlayed = false;
double playerOneTotalTimeUsed = 0; // will be stored in memory in my ARM program.
double playerTwoTotalTimeUsed = 0; // will be stored in memory in my ARM program.
double timeUsed = 0;
double maxTime = 20000000;         // 20 seconds

// Code below is done by the IRQ handler
while (startProgram) {
        if (buttonPressed == true) {
                startTimer = true;  //start the timer
                buttonPressed = false;
        }
// Now the player is given time to make a move..
// Below is the code for the button handler
        if(buttonPressed == true) {
                 startTimer = false;// stop the timer
                 timeUsed = getTimeThatHasPassed(); //in ARM, use T0TC.
                 buttonPressed = false;
                  whichPlayerPlayed = (count % 2); //Use AND instruction with 1
                 // NOTE: How the code above works is :
                 // When count is odd, it's player One's turn in the game
                 // but when count is even, it's player Two's turn in the game.
                 // So we use Modulo 2 to find if it's even or odd.
                 // µVision doesn't seem accept the MOD instruction, so I can use
                 // AND with (n-1) the number I want the modulus by.
                 // In this case, it would be AND with 1 for modulo 2.
                        if (whichPlayerPlayed == 0) {
                                playerTwoPlayed = true;
                        }
                        else {
                                playerOnePlayed = true;
                        }
                        if (playerOnePlayed == true) {
                //add new time to summation of time
                playerOneTotalTimeUsed = playerOneTotalTimeUsed + timeUsed;
                playerOnePlayed = false;
                        }
                        if(playerTwoPlayed == true) {
                //add new time to summation of time
                playerTwoTotalTimeUsed = playerTwoTotalTimeUsed + timeUsed;
                playerTwoPlayed = false;
                        }
        if (playerOneTotalTimeUsed > maxTime || playerTwoTotalTimeUsed > maxTime) {
        // i.e if someone used too much time to make a move
                        startTimer = false; // stop the timer.
                        startProgram = false;
                        maxTime = 99999999;
                        // The 999999999 is a visual indicator to the user(s)
                        // that someone's total time overflowed the max time.
                        }
                }
        }
```

How I set up the program at the start

The pseudo code above focuses on the general concepts, but I had to do a few more particular things in the reset handler. Firstly, I enabled P2.10 for EINT0 by clearing bits 20 and 21, and setting them to 01, and set edge-sensitive mode for EINT0 by loading #1 into EXTMODE. Then, I set rising edge mode for EINT0 by loading #1 into EXTPOLAR. After this was done, I had to reset EINT0 by loading #1 into EXTINT. From here, I had to prepare the timer, so I had to reset TIMER using the timer control register (TCR) by setting bit 0 to 0 and bit 1 to 1 in T0TCR and I had to reset the TC by loading 0 into T0TC.

In terms of the timer, I cleared any previous TIMER0 interrupts by writing 0xFF to the TIMER0. Penultimately, I set the match register and the IRQ on match using the match control register by loading 0x03 into T0MCR. Finally, I configured VIC for EINT0 interrupts. The final process with programming the VIC was complicated but was essential for the button handler.

Inside of the button handler

The code inside the button handler was arguably the hardest to program. It consisted of initially resetting the EINT0 interrupt. Since the button handler is invoked when the button is pressed, the timer had to be stopped inside of the button handler. Using T0TC, I found out the value from the timer. This was used appropriately to add to the correct Player using the method I mentioned above. Count was then incremented. If someone took too much time, I stopped TIMER0 and then I had to reset it. After resetting the TC, I set the maximum time to 999999999 to indicate that someone took too much time. Finally, I cleared any previous TIMER0 interrupt by writing 0xFF to TIMER0 and cleared the source of the interrupt.

(ii) Your ARM Assembly Language program, including any interrupt handlers that you implement.

[25 marks]

- My ARM program is attached to the assignment question on blackboard as ChessClock.s as requested.
- The program includes interrupt handlers and makes use of GPIO as asked in the question.

(iii) A short video demonstrating your Chess Clock. The video should be no longer than 60 seconds. You may either submit the video as a file attachment or as a link to a service such as YouTube.

[5 marks]

- I have attached an edited 60 second video demonstrating my Chess Clock to the assignment question on blackboard labelled chessClockDemo.MOV.