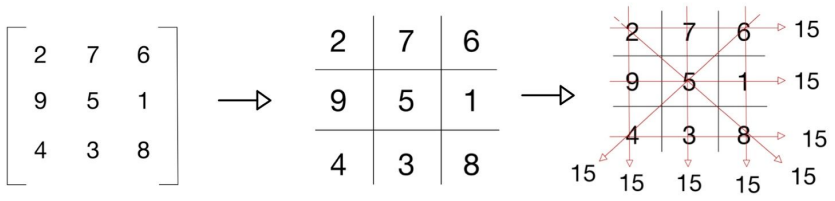
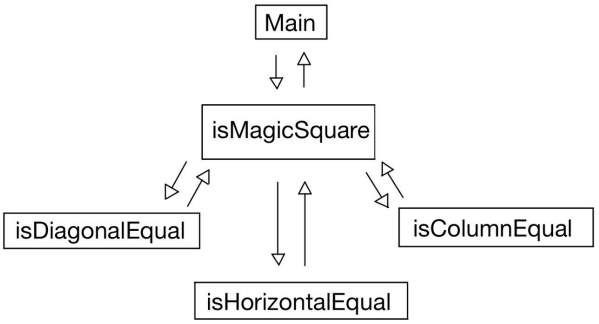
1.i. My interpretation of the magic square problem was that given a two dimensional matrix, it should be thought of in a grid-like format and each row, column, and diagonal should produce the matching sums.



When creating the main subroutine isMagicSquare, I thought that it should have a boolean return value. For parameters, I gave the address of the start of the array, so the array would be accessible and also the size of the array so I would be able to configure columns and make sure not to go beyond where the array is stored.

Within the main subroutine isMagicSquare, I thought it should first check whether it was a one by one matrix because it is known that a one by one is a magic square, so it is unnecessary to try adding its rows and columns the same way one would for a larger matrix. After that, the main isMagicSquare subroutine checks whether the diagonals, rows, and columns have matching sums by calling on three separate subroutines.



I broke it into further subroutines to try and maximize the efficiency of finding the rows, columns, and diagonals individually. In addition, breaking it down further allows for it to be more easily understood, tracked, and tested. Each of the subroutines it calls is uniform in its parameters and return values, which match that of the main subroutine as well. However, the further subroutines also return the

matching sum it has found to make sure that every direction produces the same result. If after calling a subroutine, it returns false, the main subroutine branches to the end of the subroutine to return the value without making further calls to the other subroutines. However if the first subroutine called returns true the matching sum is stored and compared to later subroutines that return true. If the two sums do not match then it immediately returns false.

```
public static boolean isMagicSquare(int[][]array, int size){
    if(size == 1)
        return true;
    int diagonalSum = isDiagonalEqual(array, size);
    if(diagonalSum == null)
        return false;
    int horizontalSum = isHorizontalEqual(array, size);
    if(horizontalSum == null || diagonalSum != horizontalSum)
        return false;
    int columnSum = isColumnEqual(array, size);
    if(columnSum == null || diagonalSum != columnSum)
        return false;
    return true;
}
```

Within the main subroutine, the first subroutine called is isDiagonalEqual. No matter how large the array is, a square can only ever have two diagonals, so particularly for the larger arrays that are not magic squares this could prove more efficient because it will not have to make continuous loops to find each individual sum. In particular, when I created the subroutine I came to the realization that I could get both diagonal sums at the same time if I were to go through the rows once and kept track of both column indexes. Because the top left and bottom right diagonal always have the same index for both the column and the row, I only used two registers to keep track of both of them. After going through and adding each element to the corresponding diagonal sum, the sums were compared and returned whether they matched.

```
public static int isDiagonalEqual(int[][] array, int size){
    int firstDiagonalSum = 0;
    int secondDiagonalSum = 0;

    for(int firstDiagonal = 0, int secondDiagonal = 0;
        firstDiagonal < size; firstDiagonal++, secondDiagonal++){
        firstDiagonalSum += array[firstDiagonal][firstDiagonal];
        secondDiagonalSum += array [firstDiagonal][secondDiagonal];
    }

    if(firstDiagonalSum!= secondDiagonalSum)
        return null;

    return firstDiagonalSum;
}
```

The second subroutine called within the main isMagic subroutine is isHorizontalEqual. This checks whether all the horizontal sums or rows have matching sums. It begins by adding and storing the first two values in the row as it begins to find the current row's sum (it should be noted this is an acceptable assumption because 1x1 arrays do not go into any of the further subroutines). This adds some more code that could be removed if it were to go straight to storing values singly and combining it with the total. However, in execution, being able to store the two values at the same time ends up being more efficient. Particularly due to the fact it initializes the register with the first sum instead of having to initialize it with zero. The address is updated each time and the current element is added to the current sum, while also keeping track of the current 'column' it would correspond to. The first sum of the first row is then saved in a register that later rows will compare their sum to. If they do not match they immediately branch to the end of the subroutine, returning false. If it goes through all the elements with matching sums, it returns true and sets the matching value as the first row sum.

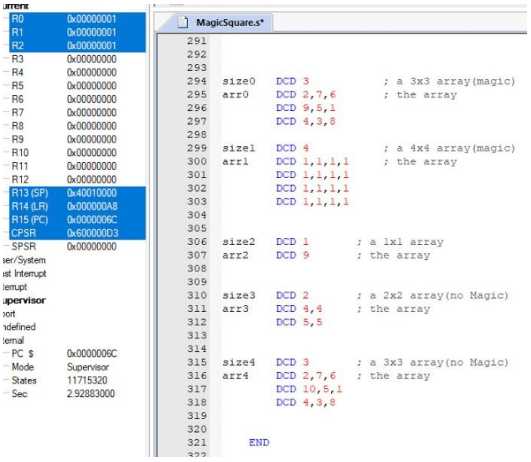
```
public static int isHorizontalEqual(address of array, int size){
    int rowSum = 0;
    addressOfArray = address of array;
    for(int row = 0; row < size; row++){

        int arrayElement = Memory.byte[addressOfArray++];
        int secondArrayElement = Memory.byte[addressOfArray++];

        currentSum = arrayElement + secondArrayElement;

        for(int col = size - 2; col > 0; col--){
            int arrayElement = Memory.byte[addressOfArray++];
            currentSum += arrayElement;
        }
        if (row == 1)
            rowSum = currentSum;
        else if(rowSum != currentSum)
            return null;
    }
    return rowSum;
}
```

The final subroutine called is isColumnEqual. This subroutine keeps track of the current column and current row and finds the address of the element by multiplying the row by the row size and then adding the column index. It iterates through until it has gone through each element in the column, adding it to its column total. After storing the first column total, the later column totals are compared to the first and branches to the end of the subroutine if they do not match and returns false. If after running through they have all matched, it returns

true and returns the first column's sum. It is important to note that because of the multiplication and addition necessary to find the address, this subroutine will be a bit longer than the isHorizontalEqual subroutine. Assuming that the two dimensional array is not a magic square, this is put last because if the array has a multitude of different directional sums, it may be slightly more efficient to have used one of the previous first. However, it should be noted that this may not hold true for all cases because isColumnEqual would be faster than isHorizontalEqual if the first two columns have different sums, but the last two rows are what have different sums.

```
public static int isColumnEqual(int [][] array, int size){
    int columnSum = 0;

    for(int col = 0; col < size;){
        int currentSum = 0;

        for(int row = 0; row < size; row++){
            currentSum += [row][col];
        }
        col++;
        if(col != 1)
            if(currentSum != columnSum)
                return null;
        else
            columnSum = currentSum;
    }
    return columnSum;
}
```

When testing the isMagic subroutine, I initially tested each of the subroutines it called to make sure they worked properly. When testing the isMagicSquare subroutineit I used different sized arrays as well as a variation of arrays that did and did not have magic. The arrays' results were pushed onto a stack in such a way that when popped off at the end the array number's results were in the next to the corresponding register number.



1.iii. My program is made efficient by the order in which the main subroutine makes calls to the other subroutines. In particular checking whether it is a 1x1 matrix first, which then allows the further subroutines to be optimized for larger matrices. The way each subroutine has the same return value as the main subroutine also allows for it to be more efficient when going to return the result. In addition to that, each time one sum is found to not match up with another, the program immediately branches out and returns false, which makes it quite efficient.

For my program to be improved, it might make more sense to combine the subroutine that checks for diagonals and the program that checks the columns to be combined. Another possibility would be to put them all into one subroutine. It would be a long subroutine and might be quite difficult to follow, however, it could consistently use the same registers and instead of having to branch out of two subroutines, it would increase efficiency to only have to branch out of one. In addition, it wouldn't need to move the address and size values before calling one of the subroutines, which would also improve its efficiency. It would also decrease the length of my program because if it were false there would only have to be one line instead of various individual lines within subroutines setting it to false.

2.i. My approach to the chess clock began with a bit of research to find out what a chess clock was. I found that it was typically made up of two timers that counted down from 20 minutes, typically counting down by seconds. So, I set out to make a usable chess clock like the ones I had found. While I initially planned to make use of the timers in ARM as my two chess clocks, I found that they only timed for 4 minutes (not typically that short, also they seemed to be more easily adjustable), they only counted up (not typical for chess timers), and they counted in hexadecimal (which is also not typical), so I decided that I would have to go about it a bit differently in order to produce what I wanted to.

I decided to put the two clocks in memory, and, because I wanted them to look like real clocks (that have a colon), I viewed them in ASCII. That meant that when I set up the reset handler I would store the ascii number values individually bit by bit. I decided to put the address for clock one and clock two at the last bit of the five bit timer. I decided to put it at the end because it would make subtracting easier, particularly because most times it just needs the last seconds digit to go down by one. I separated the two timers by 3 bits, mostly just for visual separation between the two of them to make it look a bit nicer. Between the two, I put the address for the currentTimer, which helped track which clock was the one that should be updated. Additionally in the reset, the timer is configured and the VIC for timer0 is set. The button is also set up, which means that the P2.10 for EINT0 was configured as was the VIC for EINT0 interrupts. Most of these configuring came from the button-int and timer-int examples covered previously.

```
char zero = '0'
char colon = ':'
char two = '2'

clockOne = address of last bit in clock one
Memory.byte[clockOne--] = zero
Memory.byte[clockOne--] = zero
Memory.byte[clockOne--] = colon
Memory.byte[clockOne--] = zero
Memory.byte[clockOne] = two

clockTwo = address of last bit in clock two
Memory.byte[clockTwo--] = zero
Memory.byte[clockTwo--] = zero
Memory.byte[clockTwo--] = colon
Memory.byte[clockTwo--] = zero
Memory.byte[clockTwo] = two

currentTimer = address of bit determines current clock
int currentClock = 0
Memory.byte[currentTimer] = currentClock
```

The timer0 match register was configured to be set to 1 second. Because most chess clocks seemed to change by seconds, I figured that every second I would update the clock so that it had that same look to it. So for the timer interrupt, which occurs every second, the timer first resets and stops. Using the currentTimer address, the current chess clock is then accessed and stored as an address register. The program enters a loop then where the bit number is tracked. The current bit digit is then taken and compared to a colon, if it is the colon it moves onto the next bit. If it is not a colon it is then compared to zero. If the digit is in fact zero then it sets the current digit to 9 (unless it is the fifth digit, which would be the

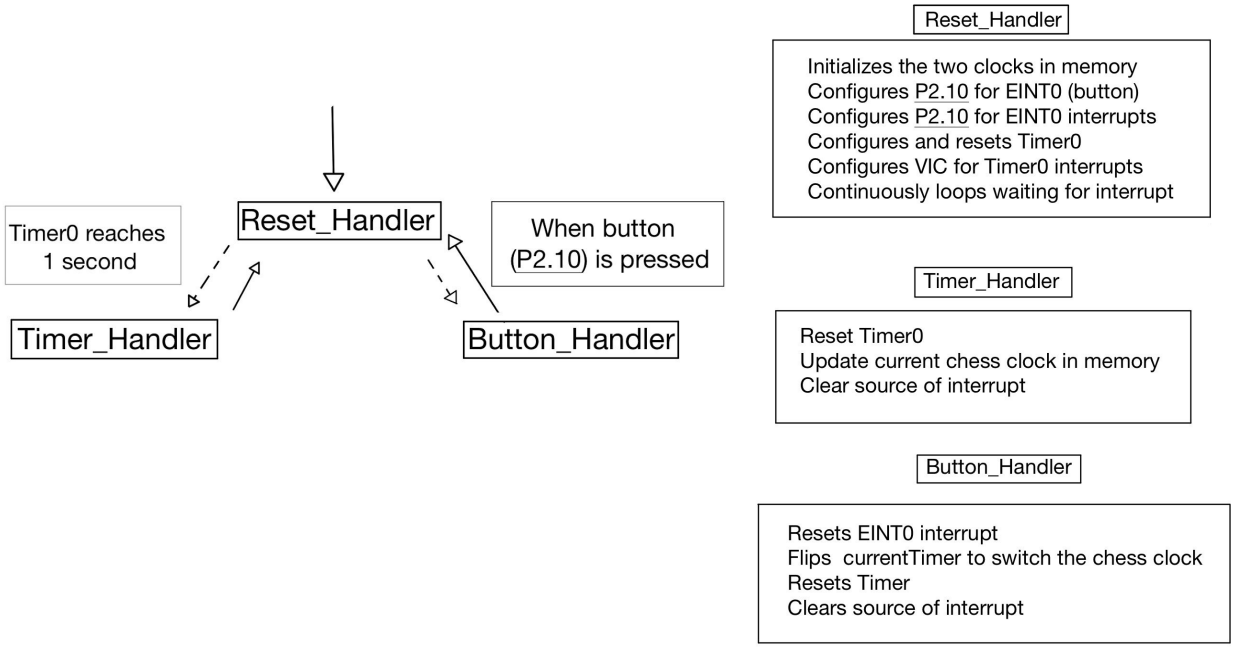
```
currentTimerAddress = adress of current timer
currentTimer = Memory.byte[currentTimerAddress]

if(currentTimer == 1)
    clockAddress = address of last bit in firstClock
else
    clockAddress = address of last bit in secondClock

bitNumber = 1

while(bitNumber <= 5){
    bitNumber++

    char currentDigit = Memory.byte[clockAddress]
    if (currentDigit != ':'){
        if(currentDigit == '0'){
            if(bitNumber == 5)
                break

            boolean isEnoughTime = false;
            for(int tempBit = bitNumber+1, tempAddress = clockAddress -1 ;
                bitNumber <= 5; bitNumber++, tempAddress--){
                char tempDigit = Memory.byte[clockAddress]
                if(tempDigit != ':'){
                    if(tempDigit != '0'){
                        isEnoughTime = true
                        break
                    }
                }
            }
```

start of the timer, or the second digit which goes to 5 because there are 60 seconds not 90). It then goes to the next digit and goes through the loop again after subtracting one from the address register to get the next bit. If the digit is not a colon or zero then it will branch out of the loop and subtracts one from the current digit and is then stored in the current digits address. After that, the clock will be updated in memory and the source of the timer interrupt will be cleared.

When P2.10 is pressed, the button handler initially resets the EINT0 interrupt. Then, the bit in the address of currentTimer is taken and flipped so that now when the timer interrupt is called the other clock will now be counting down. The timer is then reset and then the source of the interrupt is cleared.

```
currentTimer = address of bit determines current clock
currentClock = Memory.byte[currentTimer]
currentClock = ~currentClock
Memory.byte[currentTimer] = currentClock
```

The interrupts make this program a little different from the previous program's flow, but the sequence of events are not too difficult to follow. The program first runs through the reset handler and sets everything up. Then, it loops through the stop branch until there is an interrupt from either the timer or the button. It then branches to the corresponding button handler or timer handler. After it finishes in the handler, it goes back to the reset where it keeps looping through the stop.



The above flow chart and diagram help to simplify and explain the interrupts and the natural flow of my chess clock.