Integrated Computer Science
Trinity College Dublin, the University of Dublin

CSU 11022 Final Assignment

Pavel Petrukhin, Student ID: 19328624

Magic squares & Chess clock

Dublin, Ireland - 4th May 2020

1 Approach (Magic Squares)

1.1 Definition

A magic square is an $n * n$ matrix, where each element is an integer from 1 to n^2, every number is unique and the sum of elements in every column, row or main diagonal is the same. This number is known to be a magic constant [1]:

$$M_2(n) = \frac{1}{n} \sum_{k=1}^{n^2} k = \frac{1}{2} n(n^2 + 1) \tag{1}$$

1.2 Examples

The ARM assembly file contains 10 different test matrices, so that one can see how the program handles edge cases. There are 3 magic squares available and 7 matrices that fail to be a magic square for various reasons. I do not think I should provide examples here, since the algorithms are quite straightforward.

1.3 Steps

- check that every number in the matrix is unique and is in range $(1, ..., n^2)$
- calculate magic constant
- check whether any column, row, and main diagonal sum is equal to magic constant

1.4 Subroutine checkValid

The approach is to have an array of length n^2, where each element represents the number of times the respective number (index) occurred:

$$[a_0, a_1, a_2, a_3, ..., a_{n^2-1}] \tag{2}$$

The $k-th$ element gives us the number of times $k+1$ occurred in the matrix. For example, element a_2 represents the number of times 3 occurred in the matrix.

- if at some point the number of occurrences for a given number turns out to be greater than one, there is no need to continue checking
- if at some point the number is outside the range of $1, ..., n^2$ there is also no need to continue checking

Here you can see the C# code of the equivalent subroutine:

```
static bool checkValid(int[,] matrix, int n)
{
    int[] counters = new int[n * n];

    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            if (matrix[i, j] < 1||matrix[i,j]>n*n)
            {
                return false;
            }

            counters[matrix[i, j] - 1]++;

            if(counters[matrix[i,j]-1]>1)
            {
                return false;
            }
        }
    }
    return true;
}
```

1.5 Subroutine isMagic

- call the "checkValid" subroutine to determine whether the matrix is valid
- if the matrix is not valid, return false
- calculate magic constant
- initialize the diagonal sum variables to zero
- calculate the row and column sum at the same time by using nested for loops
- add elements to the appropriate diagonal sum after checking that they lie on one of the diagonals
- if at some point the sum (row/column) is not equal to magic constant, return false
- check if the diagonal sums are equal to magic constant
- if at least one of the diagonal sums is not equal to magic constant, return false
- return true

Here you can see the C# code of the equivalent subroutine:

```
static bool isMagic(int [,] matrix, int n)
{
    if(!checkValid(matrix,n))
    {
        return false;
    }

    int magicConstant = (n * n + 1) * n / 2;

    int mainDiagonalSum1 = 0;
    int mainDiagonalSum2 = 0;

    for (int i = 0; i < n; i++)
    {
        int columnSum = 0;
        int rowSum = 0;
        for (int j = 0; j < n; j++)
        {
            rowSum += matrix[i, j];
            columnSum += matrix[j, i];
            if (i == j)
            {
                mainDiagonalSum1 += matrix[i, j];
            }

            if (i == n - j - 1)
            {
                mainDiagonalSum2 += matrix[i, j];
            }
        }

        if (rowSum != magicConstant)
        {
            return false;
        }

        if(columnSum != magicConstant)
        {
            return false;
        }
    }
```

```
    if(mainDiagonalSum1 != magicConstant)
    {
        return false;
    }

    if(mainDiagonalSum2 != magicConstant)
    {
        return false;
    }

    return true;
}
```

1.6 Notes

The subroutines above illustrate algorithms, which are implemented in assembly language. However, the order of operators and statements is a bit different in the .s implementations (without impacting the algorithm and logic, ofcourse), since sometimes it allows us to execute less instructions, which is especially important in the context of Load/Store instructions.

There is no pseudo code provided in .s file, however there is an appropriate amount of comments, which make it clear what is done at each point in the program.

2 Efficiency

It is important to note that we have to make at least n^2 accesses to memory to read all the elements, which means that the complexity cannot be better than $O(n^2)$, where n is the size of square matrix.

The program specified above has two blocks of two nested for loops going one after another, which means that asymptotically time required to run the program will be proportional to $2n^2$. Therefore, the time complexity is $O(n^2)$.

As for the memory, an array of n^2 words is created in "checkValid" subroutine, so the memory complexity is also $O(n^2)$

There was a way to use just one block of two nested for loops. The program could calculate the sums at the same time with checking that the numbers are unique and are in the appropriate range. However, I decided not to implement this, because it would make the program very difficult to understand as well as the asymptotic complexity would still remain $O(n^2)$.

I could also decrease the number of registers I use by reusing them more, however I think that it is detrimental to the readability of code.

Finally, there is always a way to decrease the number of arithmetic and Load/Store instruc-

tions executed by looking at the code more carefully. I intend to do that in the summer break.

3 Approach (Chess clock)

3.1 Interpretation of the chess clock[2]

I consider the chess clock to be a device which has two timers. Initially the device is in reset state, which allows the user to specify the timer duration by pressing player 1 and player 2 buttons. When the reset button is pressed the device is transferred to a game state. Now it is waiting for one of the players to start the game. When one player presses the button their timer stops and the other player's timer starts. If the reset button is pressed the device is transferred to the reset state. When one of the timers reaches the threshold the game stops and if the user presses the reset button, the device is transfered back to a reset state. Here is a state diagram illustrating the above description:

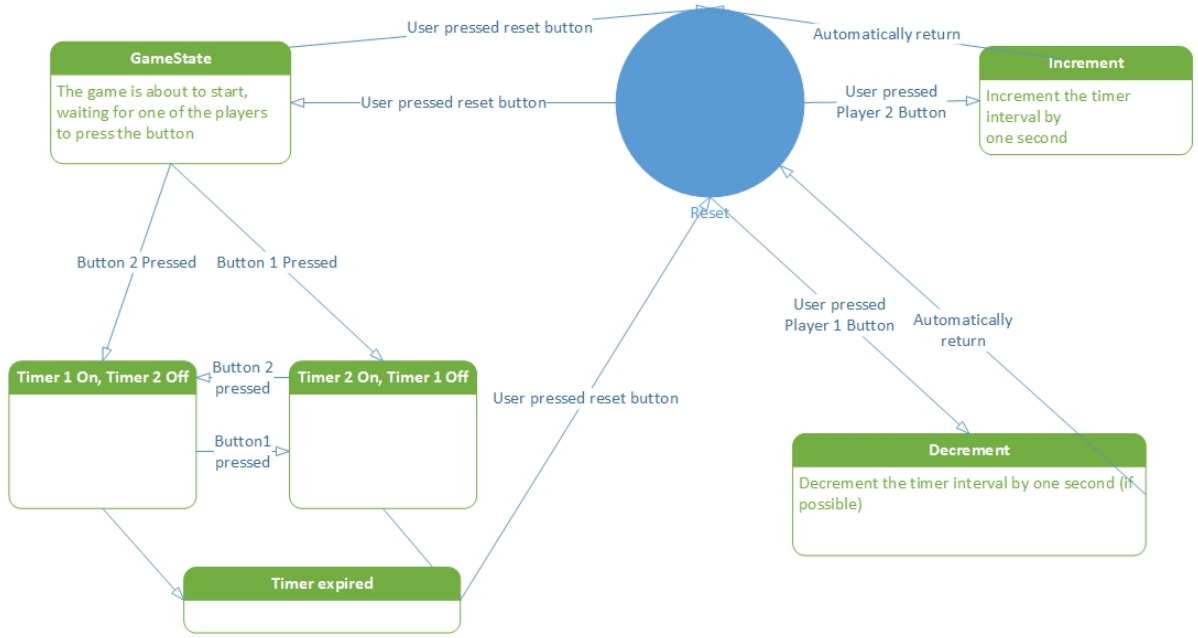

Figure 1: State diagram of the chess clock

I believe that such design is reasonable, because it allows the chess clock program to be used for various games, which can obviously have different time limits for the players. It also enables the user to start new games without running the program again (rebooting the device).

3.2 ARM Assembly implementation

I used the techniques for handling IRQ exceptions discussed in the class. More specifically, I set up P2.10, P2.11 and P2.12 for GPIO. P2.10, P2.11 and P2.12 are assigned to EINT0, EINT1 and EINT2 respectively. I decided to interpret P2.10 as a player 1 button, P2.11 as a reset button and P2.12 as a player 2 button. I configured the interrupt handling process by using VIC. As for the timers, I configured Timer0 and Timer1 to work with IRQs using the timer interrupt example. For each interrupt I wrote a subroutine handler which is called from the top level exception handler.

I implemented a simple error handling scheme by enabling and disabling interrupt channels. I stored variables which describe the state in memory, so that they are not corrupted and can be accessed in every mode.

3.3 State variables

3.3.1 Reset

This is a boolean variable. If it is true, the user has an opportunity to set the new timer interval. However, if it is false, pressing one of the player buttons will start the game.

3.3.2 Timer interval

This is an integer value, which represents the duration of the game. For instance, in blitz chess this is 5 minutes.

3.3.3 Player who has run out of time

An integer value, which is -1 when no player ran out of time, 1 when first player ran out of time and 2 when second player ran out of time.

3.4 Subroutines

In the context of exception handling, pseudo code is not a good way to present the subroutines. Therefore, I decided to list what is done in every subroutine step by step.

3.4.1 Reset handler

- Initialize stack pointers
- Configure Timer0 and Timer1 for IRQs using VIC
- Configure P2.10, P2.11, P2.12 for external interrupts using VIC
- Timer interval is set to 30 seconds, the reset flag is set to 1 and the "run out of time player" indicator is set to -1 (no player ran out of time at this point)

3.4.2 IRQ handler

- branch to the appropriate interrupt handler by getting the address from VIC

3.4.3 Timer 1

- reset interrupt
- since timer 1 is already stopped, just stop timer 2
- set the "run out of time player" to 1
- disable button 1 and button 2 interrupt handlers
- clear source of the interrupt

3.4.4 Timer 2

- reset interrupt
- since timer 2 is already stopped, just stop timer 1
- set the "run out of time player" to 2
- disable button 1 and button 2 interrupt handlers
- clear source of the interrupt

3.4.5 Button1

- reset interrupts (EINT0 and EINT2)
- if reset flag is set
  - decrease the timer interval by one second (if possible)
- if reset flag is not set
  - stop timer 1 and start timer 2
  - disable button 1 and enable button2
- clear source of the interrupt

3.4.6 Button2

- reset interrupts (EINT0 and EINT2)
- if reset flag is set
  - increase the timer interval by one second
- if reset flag is not set
  - stop timer 2 and start timer 1
  - disable button 2 and enable button 1
- clear source of the interrupt

3.4.7 Reset

- reset interrupt
- if reset flag is set
  - update the match control registers with a new time interval
  - clear the reset flag
  - enable button 1 and button 2 interrupts
- if reset flag is not set
  - stop and reset both timers
  - set the "run out of time player" to -1, since no player ran out of time at this moment
  - set the reset flag
- clear source of the interrupt

References

[1] E. W. Weisstein, "Magic square." https://mathworld.wolfram.com/MagicSquare.html

[2] Wikipedia, "Chess clock." https://en.wikipedia.org/wiki/Chess_clock