ISTANBUL TECHNICAL UNIVERSITY

FACULTY OF ELECTRIC–ELECTRONICS

DEPARTMENT ELECTRONICS & COMMUNICATION ENGINEERING

BLG212E Microprocessor Systems

Term Project Report

Match Puzzle Game

Fatih Sari

040190221

“Microp”

INTRODUCTION

In this project, we designed a simple card matching game using the Arm Assembly programming language and the Keil Microvision IDE. Our goal in the game is to guess the cards that are the same as each other and to ensure that all the cards are revealed. At the beginning, all the cards are in the closed position and we can reach the position of the card we want to open with the arrow keys and open the card we want by pressing the B button. If we guess the correct cards, the cards that are face up will remain face up until the end of the game. If different cards are opened, both cards will return to the face down position. In terms of design, we made a study inspired by the old legend Yu-Gi-Oh cards.

TEAM INFO

Team Name : Microp

Fatih Sari - 040190221

Pressing B button to open or close a card. Implementation and development of closed card, frame or open card suppression algorithm

Nurgül Gürler - 040190225

Counting consecutively opened card number and if this number is 2, then deciding last opened two cards are same or different, For the result of the above sentence, updating borders for printing the result image.

Hatice Nur Andı - 040200203

Pressing direction buttons to move, Controlling the currently printed card is open or close,

IMPLEMENTATION

<u>Registers used for specific purposes:</u>

R8 : contains the location column number and row number of cursor
R9 : contains the location of open cards
R12: contains the minimum column number of currently printed card
R11: contains the minimum row number of currently printed card
R7 : move on the picture bits in main program
R10: contains the locations of the consecutively opened cards.(max 2)

```
__main          PROC
                ldr     r0, =0x00001e00
                mov     r8, r0
                movs    r0, #0
                mov     r9, r0
                b       prt_scr
                ENDP
```

First of all, in the main part of our program, our program starts by assigning the r8 register, the first value of our registir containing the starting position of our cursor, and assigning 0 to r9 since all cards are closed at the beginning.

```
prt_scr         PROC
                ldr     r0, =0x40010000
                movs    r6, #0
                mov     r12, r6
                movs    r6, #30
                mov     r11, r6
                movs    r7, #0
                mov     r2, r11
                mov     r3, r12
                str     r2, [r0]
                str     r3, [r0, #0x4]
paint           str     r6, [r0, #0x8]
                b       openCardCheck
openCardOut     mov     r1, r8
                lsrs    r1, r1, #20
                cmp     r1, r12
                bne     notFramed
                mov     r1, r8
                lsrs    r1, r1, #8
                ldr     r4, =0x00000FFF
                ands    r1, r1, r4
                ldr     r4, =yugioh_arka_scaled_wframe
                cmp     r1, r11
                beq     Framed
notFramed       ldr     r4, =yugioh_arka_scaled
Framed          ldr     r6, [r4, r7]
```

At the end of the main part, we switch to the **prt_scr** function, which is in a continuous loop, where we do our printing to the screen. Here, the initial row and column information of the card region that we will print on the screen is assigned to the r11 and r12 registers. In addition, r0 register is assigned to the required address in order to operate on the LCD. The loop starts from the **paint** label. Here, it first jumps to the **openCardCheck** function to decide whether the card in the region it is in is open or closed. If the card is not open, it means that we will either press the cursor to the screen or we will press the down card to the screen. In order to decide on this, we compare the row and colum information of the cursor we have placed in the r8 register with the starting points we have determined for the current r11 and r12 printing process. If the region is the cursor region, we load the address information of the framed background into the r4 register and keep the system running. If the match does not occur, this time we continue the game by synchronizing the address of the normal background.

Here, we also want to touch on the working principle of the r8 register. In order to use the registers efficiently, we have stored both column and row data in the r8 register. For this, we used the last 12 bits of the register to hold the column information, and the next 12 bits to hold the row data.

r8 = 0xCCCRRR00

Here, C represents the column data region and R represents the row data region. We didn't use the top 8 bits for any purpose.

```
openCardCheck   PROC
                push    {r0-r3}
                movs    r0, #1
                mov     r1, r12
cmpColumnC      cmp     r1, #0
                beq     countedColumnC
                adds    r0, r0, #1
                subs    r1, r1, #40
                b       cmpColumnC
countedColumnC  mov     r1, r11
cmpRowC         cmp     r1, #30
                beq     countedRowC
                adds    r0, r0, #8
                subs    r1, r1, #70
                b       cmpRowC
countedRowC     movs    r2, #1
                lsls    r2, r2, r0
                mov     r5, r0
                mov     r1, r9
                ands    r1, r1, r2
                cmp     r1, #0
                pop     {r0-r3}
                beq     openCardOut
                ldr     r4, =colorTable
                lsls    r5, r5, #2
                ldr     r6, [r4, r5]
                b       openCardActive
                ENDP
```

Inside the **openCardCheck** function, it is determined which card is the card by designing a kind of counter to determine which card region we are currently printing. After this process, we compare the number of open cards we have stored in the r9 register with this data, and if the region we are in has been opened before, we return to the main loop by pulling the required RGB assignment from the address where our color table is located. Otherwise, the system will continue its operation from where it was.

```
openCardActive  movs    r5, #255
                ands    r5, r5, r6
                lsls    r5, r5, #16
                movs    r1, #255
                lsls    r1, r1, #16
                ands    r1, r1, r6
                lsrs    r1, r1, #16
                adds    r5, r5, r1
                ldr     r1, =0xff000000
                adds    r1, r1, r5
                movs    r5, #255
                lsls    r5, r5, #8
                ands    r6, r6, r5
                adds    r6, r6 ,r1

                adds    r7, r7, #4
                movs    r5, #40
                add     r5, r5, r12
                adds    r3, r3, #1
                cmp     r3, r5
                bne     nc
                mov     r3, r12
                movs    r5, #70
                add     r5, r5, r11
                adds    r2, r2, #1
                cmp     r2, r5
                bne     nr
```

If it is detected that there is an open card, a direct jump to the **openCardActive** label is made with a labl in order to prevent the background addresses from being loaded into the r4 register again. Due to the file converter we used in this label, we had to swap R and B values. Therefore, we obtained the correct RGB values by updating the r6 register with a simple algorithm and masking operations. After this process, we increased the r7 register by 4 each time in order to load the new RGB values into the r6 register and make progress in the background addresses. The reason we increase it by 4 is that every time we load with the ldr instruction, it pulls 4 data from the file. After this process, we check whether the pixel we are processing in the card region we are in has reached the limits and update the borders according to the situation.

```
                ldr     r5, =0x00000118
                movs    r7, #0
                cmp     r5, r12
                bne     border_update
                ldr     r5, =0x000000aa
                cmp     r5, r11
                beq     refs

                b       border_update
afterUpd        mov     r2, r11
                mov     r3, r12

                ldr     r0, =0x40010000
nr              str     r2, [r0]
nc              str     r3, [r0, #0x4]
                b       paint

refs            movs    r7, #1
                str     r7, [r0, #0xC]
                b       checkTruth
not2CardOpen    movs    r7, #0
                b       border_update
                ENDP

```

After all pixels in a card area are scanned and filled with the required RGB values, the values of registers r11 and r12 need to be updated. This process is implemented by switching to the **border_update** function. It is important to note that here. The r2 and r3 registers hold the positions of the pixels, while the r11 and r12 registers hold the initial values of the card region. After all cards are printed on the screen, the entire screen is refreshed at once. This refresh is done by jumping to the **refs** label. The **checkTruth** function here is the function that allows us not to open more than 2 cards instantly and to check whether the opened cards are the same cards. Its description will be given below.

```
checkTruth        PROC
                  push    {r0-r4}
                  movs    r4, #25
                  movs    r3, #0
                  mov     r0, r10
compareAgain      movs    r2, #1
                  ands    r2, r2, r0
                  cmp     r4, #0
                  beq     compareFinished
                  cmp     r2, #1
                  beq     yesOne
                  lsrs    r0, r0, #1
                  subs    r4, r4, #1
                  b       compareAgain
yesOne            adds    r3, r3, #1
                  lsrs    r0, r0, #1
                  subs    r4, r4, #1
                  b       compareAgain
compareFinished   cmp     r3, #2
                  pop     {r0-r4}
                  bne     not2CardOpen
                  push    {r0-r4}
                  ldr     r0, =0x00000FFF
                  mov     r1, r10
                  ands    r0, r0, r1
                  lsrs    r1, r1, #12
                  cmp     r0, r1
                  bne     clsUnmatchC
                  movs    r0, #0
                  mov     r10, r0
                  pop     {r0-r4}
                  b       not2CardOpen
clsUnmatchC       mov     r0, r10
                  mov     r1, r9
                  mvns    r0, r0
                  ands    r1, r1, r0
                  mov     r9, r1
                  movs    r0, #0
                  mov     r10, r0
                  pop     {r0-r4}
                  b       not2CardOpen
                  ENDP

```

In this region, first counting how many 1 bits are in the r10 register and if 2 cards are not opened, the function is exited and the normal cycle continues. If 2 cards are face up and the distance between them is a predetermined amount, it is determined that the same two cards have been turned up. In this case, the r9 register and r10 registers are updated and the function is exited. Otherwise, the 2 cards that were opened last are closed back from the r9 register, and the main loop is continued.

```
border_update   PROC
                ldr     r1, =0x00000118
                mov     r0, r12
                cmp     r12, r1
                beq     columnEnd
                adds    r0, r0, #40
                mov     r12, r0
                b       afterUpd
columnEnd       movs    r0, #0
                mov     r12, r0
                ldr     r1, =0x000000aa
                mov     r0, r11
                cmp     r11, r1
                beq     rowEnd
                adds    r0, r0, #70
                mov     r11, r0
                b       afterUpd
rowEnd          movs    r0, #30
                mov     r11, r0
                b       afterUpd
                ENDP
                END
```

Here, as the name of the function indicates, we are updating the boundaries of the starting points of the card region we are in. Thanks to this operation, navigation is performed in all 24 card regions. The subject that needs to be mentioned in detail here is the column value we want to reach is 280 instead of 320. The reason for this is to save the number of registers. Before each limit control operation, the limit is determined and controlled by adding 40 to the r12 register. The same is true for rows.

```
Button_Handler  PROC
                EXPORT  Button_Handler
                ldr     r0, =0x40010010
                ldr     r1, [r0]
                ldr     r2, =0x000000FF
                ands    r1, r1, r2
                movs    r2, #128
                cmp     r1, r2
                beq     cursorR
backR           lsrs    r2, r2, #1
                cmp     r1, r2
                beq     cursorL
backL           lsrs    r2, r2, #1
                cmp     r1, r2
                beq     cursorD
backD           lsrs    r2, r2, #1
                cmp     r1, r2
                beq     cursorU
backU           lsrs    r2, r2, #1
                cmp     r1, r2
                beq     pressB
pressBBack      ldr     r1, =0x80000000
                ldr     r0, =0x40010010
                str     r1, [r0]
                bx      lr
                ENDP
```

One of the most critical points of this project was working with interrupts. Here we see the inside of the button interrupt. When the LCD screen is opened and interrupts are activated (tick the INT button), if any button is pressed, **Button_Handler** will be entered. Here, as the first operation, we take the necessary data from the address we assigned to r0, which stores the data of which button was pressed, and assign it to the r1 register. After this process, a simple masking process is performed to get rid of unnecessary data and jumping to the required function is performed according to which button is pressed.

```
cursorR         PROC
                push    {r4}
                ldr     r4, =0x00000118
                mov     r0, r8
                ldr     r3, =0xFFF00000
                ands    r0, r0, r3
                lsrs    r0, r0, #20
                cmp     r0, r4
                pop     {r4}
                beq     backR
                push    {r1}
                adds    r0, r0, #40
                lsls    r0, r0, #20
                mov     r3, r8
                ldr     r1, =0x000FFFFF
                ands    r1, r1, r3
                adds    r0, r0, r1
                mov     r8, r0
                pop     {r1}
                b       backR
                ENDP
```

In this part, if the arrow keys are pressed to the right, we need to move the cursor to the right. But we also need to check if it reaches the far right. Since we store the position of the cursor in the r8 register, we first pull the horizontal plane data from r8. With this data, necessary checks are made and if necessary, an update in r8 is made and the function is exited.

Since the other arrow keys are developed in a very similar way to the algorithm here, I do not include them all in the report. But as a working principle, I can say that it is one-to-one.

```
pressB          PROC
                push    {r0-r3}
                movs    r1, #1
                mov     r0, r8
                lsrs    r0, r0, #20
cmpColumn       cmp     r0, #0
                beq     countedColumn
                adds    r1, r1, #1
                subs    r0, r0, #40
                b       cmpColumn
countedColumn   mov     r0, r8
                lsrs    r0, r0, #8
                ldr     r2, =0x00000FFF
                ands    r0, r0, r2
cmpRow          cmp     r0, #30
                beq     countedRow
                adds    r1, r1, #8
                subs    r0, r0, #70
                b       cmpRow
countedRow      movs    r2, #1
                movs    r3, #1
                lsls    r2, r2, r1
                lsls    r3, r3, r1
                add     r10, r10, r2
                mov     r0, r9
                ands    r3, r3, r0
                cmp     r2, r3
                beq     alreadyOne
                add     r9, r9, r2
                pop     {r0-r3}
                b       pressBBack
alreadyOne      mvns    r0, r3
                mov     r1, r10
                subs    r1, r1, r2
                subs    r1, r1, r2
                mov     r10, r1
                mov     r3, r9
                ands    r0, r0, r3
                mov     r9, r0
                pop     {r0-r3}
                b       pressBBack
                ENDP
```

The B button is designed as a button that enables the face-down cards in the game to be opened or any open card to be closed again. We use the data of the Cursor position from the r8 register, as we want the operation to be performed at the position where the Cursor is located at the moment the B button is pressed. With these data, it is determined which card the Cursor is on. After this process, first the r10 register (to hide that 2 cards have been opened consecutively) and then the r9 register, the operation to be done is decided by looking at whether the card to be opened is open before. If the bit number of the card in the r9 register is 0, it is ensured that the card is closed before and that bit is set to 1 to open the card. If the current bit is already equal to 1 when the button is pressed, this time it sets the bit to 0 and closes the card back. **To be used in all these operations, the registers between r0-r3 are pushed to the stack and the lack of registers is solved.**

DISCUSSION

The developed game generally works in accordance with its purpose. Unfortunately, not much attention was paid to the performance criteria while the game was being developed. By making very simple improvements to the code, many unnecessary loops can be eliminated and the game can be run much faster. For example, **instead of pressing the screen all the time in the main function, it would be possible to make the whole system work every time a button is pressed, and otherwise the program would go to sleep mode**. Apart from this, although there was no performance improvement, in order to increase the playability, if the Cursor tried to progress after reaching the limits, it could be assigned to the initial state. In addition, although the LCD screen is 320x240, an area of 30 pixels is left at the top. We planned to **give the number of guesses in this field according to the wrong guess of the player and reduce it from here. (Like Mario's lifes in the OOP project), and we thought about adding a point counter to it**. However, as you may have guessed, we could not implement them due to a serious intensity of lectures.

Thanks to this project and the second assignment, I had the opportunity to learn and work with Arm Assembly language and Keil IDE for the first time. I had an interest in embedded systems before and I plan to develop my career in this field. I am developing small-scale projects with the STM32 Discovery development board. After that, I plan to make these experiments on a real card by trying improvements with arm assembly language. Thank you for all your hard work throughout the term.

Fatih Sari