VIETNAM NATIONAL UNIVERSITY - HO CHI MINH CITY

HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY

FACULTY OF COMPUTER SCIENCE AND ENGINEERING

COMPUTER ARCHITECTURE - CO2007

ASSIGNMENT

FOUR IN A ROW

Student name: Nguyen Huu Hao

Student ID: 2153327

Lecturer: Vice Prof. Pham Quoc Cuong

Lecturer: MSc. Bang Ngoc Bao Tam

Contents

CHAPTER 1

Introduction

The assignment requires us to build a game from scratch. Starting with a table of 7 columns and 6 rows, we are required to let 2 players competing with each other on the basis of determining which player will win. The game involves dropping down symbols representing the players' moves (with X and O) and concurrently, they also have 3 features: to undo move, to block opponent's move and to remove one piece of opponent.

With those requirements in mind, I have proceeded to sketch an idea involving 2 layers: memory management and game procedure.

1 Memory management

To efficiently keep track of the pieces on the board, I allocated a 42-byte-long array, with each byte can have 3 values:
0: empty space.
1: X piece.
3: O piece.

More than just that, I also allocated a 10-byte-long array to keep track of the status and abilities of each player. The values at indices 0 and 4 are used to keep track of the number of undo each player have left, 1 and 5 are for remove, 2 and 6 are for block, 3 and 7 are for the number of violations each player have violated, 8 and 9 are used to determined the blocked status, for player 1 and 2 respectively.

I also added miscellaneous arrays, such as a 52-byte-long to store the board and players' status in case of undo. 2 1-byte-long array with the first used to keep track of players' turn and the latter used to determine who is X and who is O. 2 10-byte-long to store each player's name.

2 Game procedure:

The overall idea is to divide the game into sections: initialization, first 2 moves, full game, endgame and miscellaneous/ utility functions. Each of those will have their own section in chapter 2, for now I will discuss the overall idea of implementing each.

1. Initialization: involves initializing the data laid out within the memory management section. I have done this in case of the players having ended the game decided to play on for another match without going through the trouble of restarting the game.

2. First 2 moves: the first 2 moves requires 2 players to place their pieces on column 4 so if they set their move to another column or if the column is invalid, their violation status will be incremented.

3. Full game: this section will involve the process of printing the status of each player, the game board, checking input, checking winning conditions and changing turns.

4. Endgame: will involve 3 outcome: winning by 4 pieces consecutively, winning by the other player's exceeding the number of violations allowed and tie.

5. Miscellaneous functions: to increase the reliability, shorten the code and the interchangeability of the program, I divided each process into many functions, with many main functions often sharing child functions.

a. Printing GUI: every time before proceeding to processing a player's move, we must print the GUI out to let the players know his and his opponent's status in order to make the best decision. Thus in this section, I will print the number of undo left and their violation number while dedicating the blocking and removing parts to the function processing the move.

b. Printing board: in this game, I have decided to use the Bitmap Display tool of the MARS to visualize the board. The concept is to print colors into pixels of the screen to dynamically visualize the board to the players instead of printing out strings of X and O and barriers.

c. Winning conditions checking: Within this large function is 4 small utility functions to check the winning conditions diagonally (both from left and right), horizontally and vertically. With diagonally from left starting from position (0, 0) (horizontal and vertical coordinates listing from up to down respectively) to (5, 4) and then proceeding down, with each iteration increase both coordinate by 1 (to (4, 3) is enough for checking winning condition but we have also consider the prospect of a chance of winning for blocking mechanics). Diagonally from right from (7, 0) to (3, 3). Horizontally from (0, 0) to (4, 0) and from (0, 0) to (4, 3). Vertically from (0, 0) to (0, 3) and from (0, 0) to (6, 3).

CHAPTER 2

Game details

In this chapter, I will report 8 main sections of my implementation of the game with 1 section for realizing the memory part I have previously discussed and 7 sections for 7 main functions.

The game starts in initialization function. The game then proceed to the first 2 moves function, who, along with full game function, will call printing GUI and printing board to print out the status of players the the board and subsequently call the winning conditions checking function. This is iterated until 1 of the 3 winning conditions reached and result in the endgame function. The players are then free to decide whether they wanted to start another session of the game, by pressing 1, or to close the game, by pressing any other keys.

1 Memory:

There are 104 bytes total participating in managing board and status of the game, with the first half actively involved and second half served as backup storage in case the player wishes to undo their move, because each time the player moved, I will print the board and GUI out for them to consider the prospects of undoing them, provided that the still have any undo left.

There are also 2 bytes governing turn and X-O status and a further 20 bytes determining 2 players' names and the final 20 bytes used for miscellaneous purposes, such as determine the players' inputs.

Moreover, I still have more text stored in memory for announcement purposes but it is not important and can be looked into more closely in the .asm file.

```
1 .data
2 #DATA SECTION
3 midgame_rows:     .space 42
4 midgame_abilities:   .space 10
5 backup_data:      .space 52
6 midgame_turn:     .space 1
7 midgame_xo:  .space 1
8 midgame_name1:    .space 10
9 midgame_name2:    .space 10
```

```
misc_mem:    .space 20
```

2 Initialization:

The overarching goal of the first section of the code is to initialize data of the game, hence I used this section to empty the 42 bytes of data of the board and 10 status slots using a for loop of 52 iterations.

```
    add $t0,$zero,$zero
    add $t1,$zero,52
RESET_LOOP:
    beq $t0,$t1,RESET_END
    sb $zero,midgame_rows($t0)
    addi $t0,$t0,1
    j RESET_LOOP
RESET_END:
    jal GUI_INIT    #draw the blue background in the board
    la $s0,midgame_rows
    la $s1,midgame_abilities    #0, 4: undo    1, 5: remove    2, 6: block    3,
        7: violation    8, 9: blocked status
    la $s2,midgame_turn #turn 0 is player 1, turn 1 is player 2
    la $s3,midgame_xo
    ori $t0,$zero,3 #3 undos to use
    sb $t0,($s1)
    sb $t0,4($s1)
    ori $t0,$zero,1 #1 remove and 1 block
    sb $t0,1($s1)
    sb $t0,2($s1)
    sb $t0,5($s1)
    sb $t0,6($s1)
```

Next, I let player 1 and player 2 enter their corresponding name with a maximum length of 9 characters, with the following demonstrates the procedure of entering player 1's name.

```
    addi $v0,$zero,4
    la $a0,start_name   #let the player know this is for entering name
    syscall
    la $a0,space
    syscall
    la $a0,start_player1    #let the player know it is their turn
    syscall
    la $a0,eol
    syscall
    la $a0,midgame_name1
    addi $a1,$zero,10
    addi $v0,$zero,8
    syscall
```

After that, the final section is to generate a random number in order to randomly assign X and O to player 1 and 2. Then printing the announcement to screen to let each player know their pieces' shape.

```
    ori $a1,$zero,2 #the range of random number generated is 0-1
    ori $v0,$zero,42     #system call signifying generate a random number with range
         in $a1
    syscall          #with 0 meaning player 1 is X
    sb $a0,($s3)         #1 meaning player 1 is O
    addi $v0,$zero,4     #get ready to print
    beq $a0,$zero,P1X    #proceeding to the corresponding situation, with function
        P1X just the reverse of the following code lines
    la $a0,start_player1     #Proceed as player 1 is O
    syscall
    la $a0,space
    syscall
    la $a0,start_player_o
    syscall
    la $a0,eol
    syscall
    la $a0,start_player2
    syscall
    la $a0,space
    syscall
    la $a0,start_player_x
    syscall
    la $a0,eol
    syscall
    j START_ENT #waiting for the player to press enter for confirmation
```

```
Welcome to Four in a row.
This version is built by Nguyen Huu Hao, a Computer Engineering student at the HCMUT.
Please have a great time enjoying the game.
Note: Please input everything according to instructions, don't use the enter button.
Please enter name (9 characters) for Player 1:
111111111
Please enter name (9 characters) for Player 2:
222222222
Player 1: You are O.
Player 2: You are X.
Press anykey to continue.
```

Figure 2.1: I/O after initialization step.

3 Printing GUI:

This function involves printing out players names, their pieces, undo left and violations. I divided the output screen in to 2 parts with the left part showing player 1's status and the right part showing player 2's, each separated by indentation. The following code will demonstrate one case of the function.

The first line is the 2 players' names.

```
PRINT_GUI:
    addi $v0,$zero,4
    la $a0,tab  #indentation and print the first line as players names.
    syscall
```

```
    la $a0,start_player1
    syscall
    la $a0,space
    syscall
    la $a0,midgame_name1     #load their name address stored in memory
    syscall
    la $a0,tab
    syscall
    la $a0,start_player2
    syscall
    la $a0,space
    syscall
    la $a0,midgame_name2
    syscall
    la $a0,eol
    syscall
    la $a0,tab
    syscall
```

The second line is their pieces based on the value stored at address of register s3.

```
    lb $t0,($s3)     #load value to determine which player is X, which is O
    bne $t0,$zero,PG_1O #branch accordingly
PG_1X:
    la $a0,start_player_x    #print their pieces right below their names.
    syscall
    la $a0,tab
    syscall
    syscall
    la $a0,start_player_o
    syscall
    j PG_CONT    #skip the case of Player 1 is O and print undo and violation
```

The third is undo left and finally, the fourth is violation numbers.

```
PG_CONT:
    addi $v0,$zero,4
    la $a0,eol
    syscall
    la $a0,undo #print the announcement of undo left
    syscall
    la $a0,tab
    syscall
    addi $v0,$zero,1     #print the number of undo right below players' pieces
    lb $a0,($s1)     #the number of undo of player 1
    syscall
    addi $v0,$zero,4
    la $a0,tab
    syscall
    syscall
    addi $v0,$zero,1
    lb $a0,4($s1)    #the number of undo of player 2
    syscall
    addi $v0,$zero,4
    la $a0,eol
    syscall
    la $a0,violation     #the announcement for violation numbers
```

```
    syscall
    la $a0,tab
    syscall
    addi $v0,$zero,1     #again, print out the number of violation of each player
        right below their number of undo left
    lb $a0,3($s1)    #the number of violation of player 1
    syscall
    addi $v0,$zero,4
    la $a0,tab
    syscall
    syscall
    addi $v0,$zero,1
    lb $a0,7($s1)    #the number of violations of player 2
    syscall
    jr $ra
```

```
        Player 1: 111111111     Player 2: 222222222
        You are O.              You are X.
Undo left:      3               3
Violations:     0               0
```

Figure 2.2: I/O of the printing GUI step.

4 Printing board:

This function enlists the help of the Bitmap display tool in MARS. Firstly, to play the game, players must set up the bitmap display tool as the following.

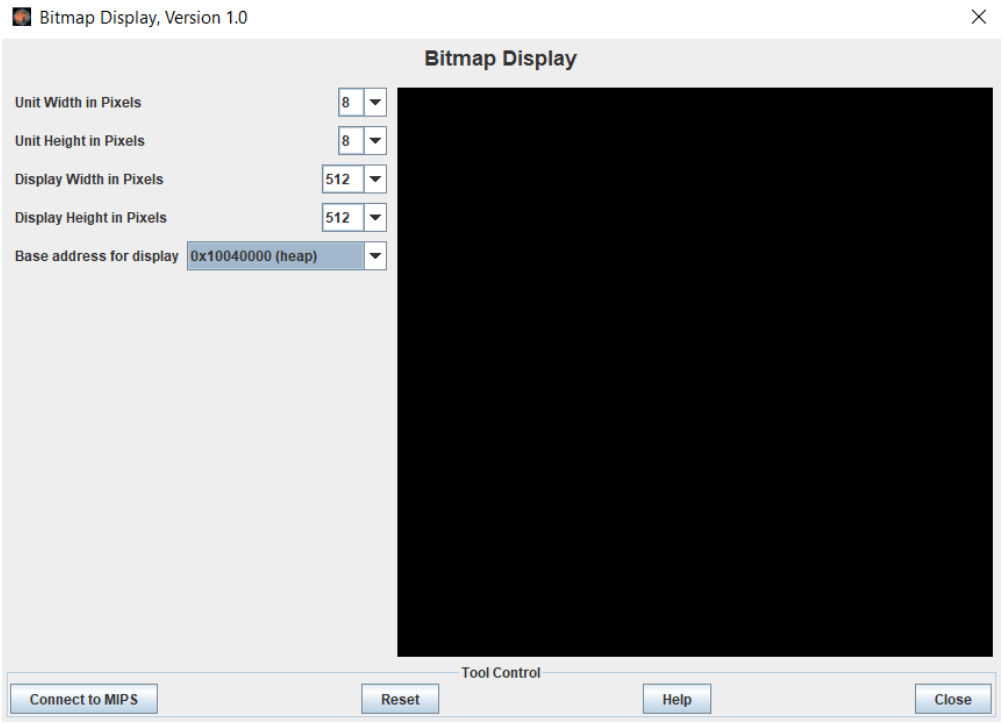

Figure 2.3: Bitmap display setup

Based on the concept discussed in chapter 1, I demonstrate the board in an area of 512x512 pixels, with each unit height and width equally 8 pixels, we then have a board of 64x64 units and each unit represents a word. These memory are stored within the heap, accounting for a total of 64x64x8=16,384 bytes=16 KB.

With the basic concept built, I then picked 4 colors to participating in drawing the board. Blue is used to fill in the background of the board, white is used to fill the each empty box, red represent player 1's pieces, gold represent player 2's pieces, be it X or O. The hex code for each color is stored in the data array ColorTable.

```
ColorTable:
.word 0x0000FF,0xFF0000,0xE5C420,0xFFFFFF    #0: Blue     1: Red   2: Gold 3: White
```

The following is the function used to build the entire background of the board and it will only be called once in the initialization part of the code. I can build it up as just 1 for loop but instead opt for 4 loops because I subdivide the board into many 8x8 boxes and this will help in the next function of printing board.

```
GUI_INIT:
    add $t0,$zero,$zero #let $t0 and $t1 be the iterators
    add $t1,$zero,$zero #$t0: row iterator $t1: column iterator
GUI_OP:
    add $t2,$zero,$zero #$t2: inner row iterator
    add $t3,$zero,$zero #$t3: inner column iterator
GUI_OP_OP:
    lui $t5,0x1004  #load heap address
    add $t4,$zero,$t2   #add up all the iterators to have the address
    add $t4,$t4,$t3
    add $t4,$t4,$t0
    add $t4,$t4,$t1
    add $t5,$t5,$t4
    lw $t4,ColorTable   #load and store blue into each pixel
    sw $t4,($t5)
GUI_OP_COND:
    addi $t2,$t2,4  #jump to another pixel
    bne $t2,32,GUI_OP_OP    #end of 8 row pixels box
    add $t2,$zero,$zero
    addi $t3,$t3,256    #jump to another row
    bne $t3,2048,GUI_OP_OP  #end of 8 column pixels box
GUI_COND:
    addi $t0,$t0,32 #jump to another box
    bne $t0,256,GUI_OP  #end of row
    add $t0,$zero,$zero
    addi $t1,$t1,2048   #jump to box by 8 rows
    bne $t1,16384,GUI_OP    #the limit of the 64x64 board
    jr $ra
```

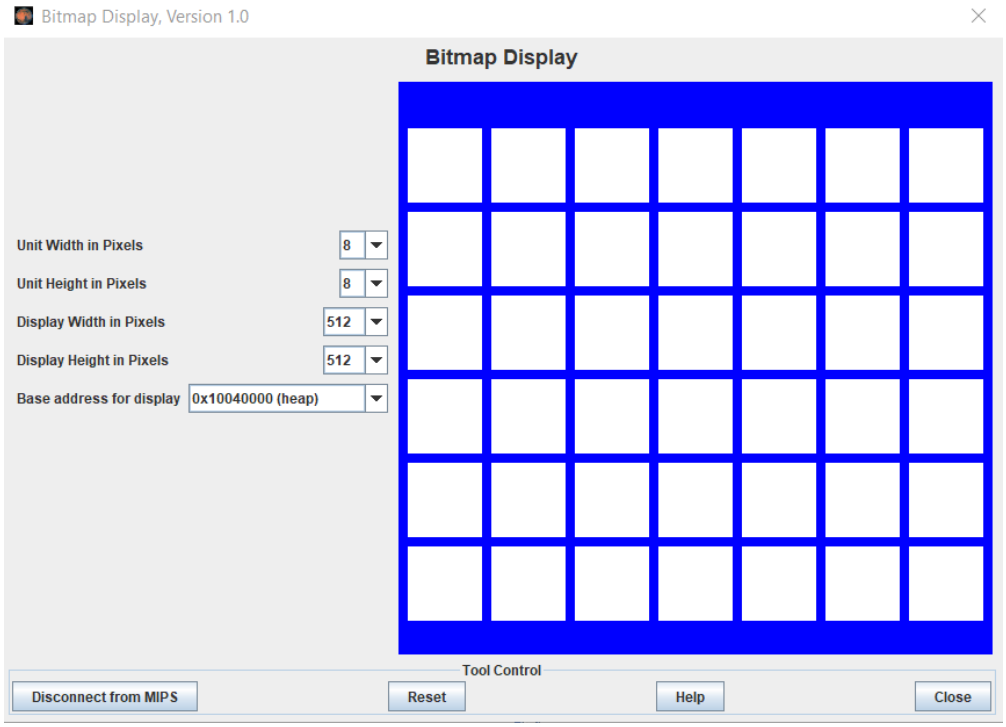

Figure 2.4: Bitmap displaying background and empty boxes.

After drawing the background, it is time to draw the boxes. Each box is 8x8, then leave 8 pixels in the column next to it and 8 pixels in the row next to it, we have a 9x9 box. With a board of 64x64, we starts from each row from pixels 4 to the end with pixel 63 to draw 7 columns, and from each column from pixels 6 to 59 to draw 6 rows.

```
PRINT_BOARD:
    addi $t0,$zero,4     #row iterator from pixel 1 horizontally
    addi $t1,$zero,1280 #column iterator from pixel 6 vertically
    add $t3,$zero,$zero
PB_OP:
    lb $t9,midgame_rows($t3)     #loading data from array to determine which type to
        draw
    add $t4,$zero,$zero #$t4: inner column iterator
    add $t5,$zero,$zero #$t5: inner row iterator
PB_CONT:
    lui $t6,0x1004  #load heap address
    add $t7,$t4,$t5
    add $t7,$t7,$t0
    add $t7,$t7,$t1
    add $t6,$t6,$t7 #add all the iterators to get correct address
    beq $t9,3,PB_O  #if the data value loaded is 3 will draw O
    beq $t9,1,PB_X  #1 will draw X
    #BOX ONLY
    j PB_WHITE  #else just fill with white for empty box
```

We now then proceed to discuss the drawing mechanics of O shape. To have the O shape, we need to have pixels filled in the following manner:
- In columns 1, 2, 7 and 8, we filled them from rows 3 to 6.
- In the remaining columns, we filled them in rows 1, 2, 7 and 8.
The color depends on the player owning the shape, with red representing player 1 and

gold representing player 2.

```
PB_O:
    addi $t8,$zero,512  #checking if in rows 1, 2 (0 and 256)
    slt $s7,$t5,$t8
    beq $s7,1,PB_O_UP   #if yes, check further
    addi $t8,$zero,1280 #checking if in rows 7, 8 (1536 and 1792)
    slt $s7,$t8,$t5
    beq $s7,1,PB_O_UP   #if yes, check further
    addi $t8,$zero,8    #in remaining rows, check for columns 1, 2 (0 and 4)
    slt $s7,$t4,$t8
    beq $s7,1,PB_O_COND #if yes, fill color
    addi $t8,$zero,20   #checking for columns 7, 8 (24 and 28)
    slt $s7,$t8,$t4
    beq $s7,1,PB_O_COND #if yes, fill color
    j PB_WHITE
PB_O_UP:
    addi $t8,$zero,8    #checking for columns 1, 2 (0 and 4)
    slt $s7,$t4,$t8
    beq $s7,1,PB_WHITE  #if yes, filled with white
    addi $t8,$zero,20   #checking for columns 7, 8 (24 and 28)
    slt $s7,$t8,$t4
    beq $s7,1,PB_WHITE  #if yes, filled with white, the remaining is with color
PB_O_COND:
    lb $t8,($s3)    #load to determine which player shape
    beq $t8,1,PB_RED    #player 1 is O
    j PB_GOLD   #player 2 is O
```

We now then proceed to discuss the drawing mechanics of X shape. To have the O shape, we need to have pixels filled in the following manner:
- In not in column 4, 5, we filled them in all rows except 4, 5.
- In the remaining columns, we filled them in rows 4, 5.
Again, the color depends on the player owning the shape, with red representing player 1 and gold representing player 2.

```
PB_X:
    addi $t8,$zero,768  #checking if in rows 1-3 (0-512)
    slt $s7,$t5,$t8
    beq $s7,1,PB_X_UP   #if yes, check further
    addi $t8,$zero,1024 #checking if in rows 6-8 (1280-1792)
    slt $s7,$t8,$t5
    beq $s7,1,PB_X_UP   #if yes, check further
    addi $t8,$zero,12   #remaining rows, check for columns 1-3 (0-8)
    slt $s7,$t4,$t8
    beq $s7,1,PB_WHITE  #if yes, fill white
    addi $t8,$zero,16   #check for columns 6-8 (20-28)
    slt $s7,$t8,$t4
    beq $s7,1,PB_WHITE  #if yes, fill white
    j PB_X_COND #fill color with the remaining
PB_X_UP:
    addi $t8,$zero,12   #check for columns 1-3 (0-8)
    slt $s7,$t4,$t8
    beq $s7,1,PB_X_COND #if yes, fill color
    addi $t8,$zero,16   #check for columns 6-8 (20-28)
    slt $s7,$t8,$t4
    beq $s7,1,PB_X_COND #if yes, fill color
    j PB_WHITE
PB_X_COND:
```

```
    lb $t8,($s3)     #load to determine which player shape
    beq $t8,1,PB_GOLD    #if yes, X belongs to player 2
```

The remaining code lines demonstrate the process of taking the color and assigning it into the pixel and also the stopping conditions, which resemble the printing background code above.

```
PB_RED:
    addi $t8,$zero,4     #get the second word for red
    lw $t7,ColorTable($t8)
    sw $t7,($t6)
    j PB_BOX_COND
PB_GOLD:
    addi $t8,$zero,8     #the third word for gold
    lw $t7,ColorTable($t8)
    sw $t7,($t6)
    j PB_BOX_COND
PB_WHITE:
    addi $t8,$zero,12    #the fourth word for white
    lw $t7,ColorTable($t8)
    sw $t7,($t6)
PB_BOX_COND:
    addi $t4,$t4,4  #the same condition as the background
    bne $t4,32,PB_CONT
    add $t4,$zero,$zero
    addi $t5,$t5,256
    bne $t5,2048,PB_CONT
PB_COND:
    addi $t3,$t3,1  #get another byte of data
    addi $t0,$t0,36
    bne $t0,256,PB_OP
    addi $t0,$zero,4     #the outer column increase by 1 more pixel
    addi $t1,$t1,2304    #the outer row increase by 1 more pixel
    bne $t1,15104,PB_OP #6 row boxes
    addi $v0,$zero,4
    la $a0,eol
    syscall
    la $a0,tab
    syscall
    jr $ra
```

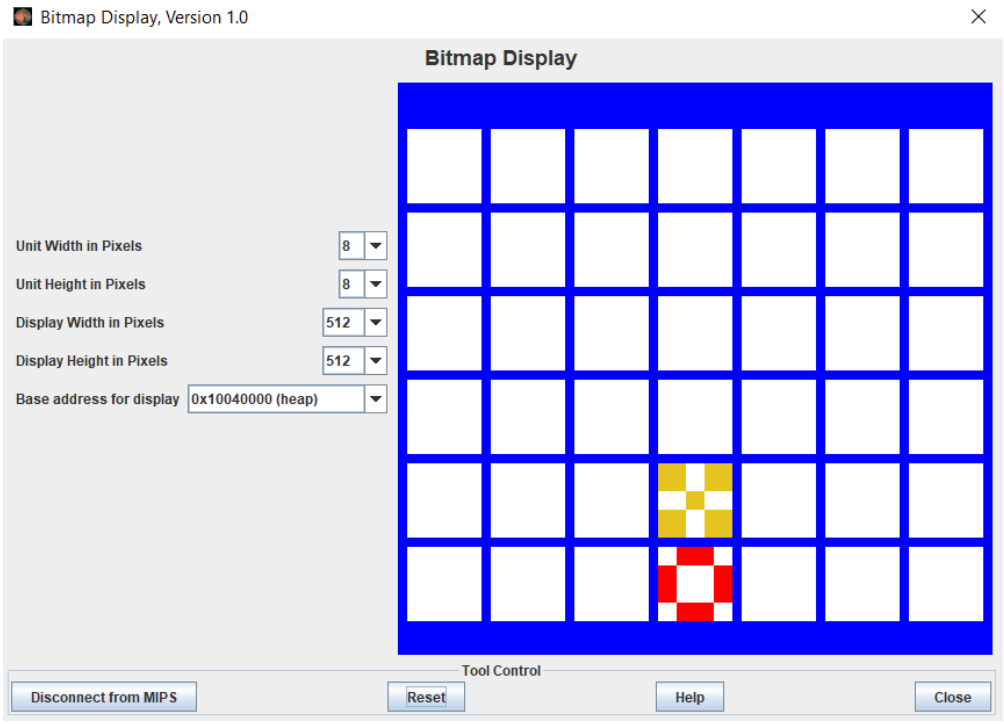

Figure 2.5: Example of Player 1 O and Player 2 X

5 Winning conditions checking:

As discussed, there are 4 types of pieces placement that may result in victory, therefore, we have to divide into 4 sub-functions. They all will belong to 1 outer loop and each sub-function have their inner loops and these loops are the "Repeat ... Until ..." loop. Like all other functions, the data will have to reinitialize and to help with the blocking ability, registers a2 and a3 will represent X and O high chance of winning and will also be initialized here.

At first, they will load the byte at the location, checking if it is empty or not, if it is empty, move to another type of winning condition. If it is not, store the status of the byte as X or O and check if it is continuous. If it is not, skip to another type of winning condition. If there are 3 continuous pieces of any winning condition type, increase a2 or a3 to 1 (corresponding to X or O chance of winning) and if there are 4, the corresponding player will win.

The winning conditions will be checked as discussed in the idea section. The implementation involves diagonally checking because horizontally and vertically require more iterations, so the conditions of diagonally checking is a subset of the other 2.

```
1 WIN_CHECK:
2     addi $t0,$zero,0     #t0: row iterator    t1: column iterator t2: row condition
          t3: column condition & winning condition
3     addi $t1,$zero,0     #t4: position variable  t5: continuous check     t6: X or O
      (1 X; 2 O)    t7: current value
4     addi $t2,$zero,4     #t8: row pos         t9: column pos       a2: X high chance
          a3: O high chance
5     addi $t3,$zero,5
6     addi $a0,$zero,7
```

```
    addi $a2,$zero,0     #a2: X high chance
    addi $a3,$zero,0     #a3: O high chance
WIN_CHECK_OP:
    #check diagonally left and right conditions first
    slt $a1,$t0,$t2 #if exceed these conditions, skip diagonally
    beq $a1,$zero,WIN_CHECK_HOR_REP
    slt $a1,$t1,$t3 #if exceed, skip also horizontally
    beq $a1,$zero,WIN_CHECK_VERT_REP
    #check diagonally left
    addi $t6,$zero,0     #preparing conditional variables
    addi $t5,$zero,0
    add $t8,$zero,$t0
    add $t9,$zero,$t1
WIN_CHECK_DIAG_L:
    mul $t4,$t8,$a0 #get the address of the byte
    add $t4,$t4,$t9
    lb $t7,midgame_rows($t4)
    beq $t7,0,WIN_CHECK_DIAG_R_REP  #if empty, go to another function
    beq $t7,1,WIN_CHECK_DIAG_L_X_CONT
    #O continuous
    beq $t6,1,WIN_CHECK_DIAG_R_REP  #is X, skip this iteration
    beq $t6,2,WIN_CHECK_DIAG_L_O_CONT_NEXT
    addi $t6,$zero,2
WIN_CHECK_DIAG_L_O_CONT_NEXT:
    addi $t5,$t5,1   #increase continuous
    beq $t5,4,O_WIN
    beq $t5,3,WIN_CHECK_DIAG_L_O_HIGH_CHANCE
    j WIN_CHECK_DIAG_L_COND
WIN_CHECK_DIAG_L_O_HIGH_CHANCE:
    addi $a3,$zero,1
    j WIN_CHECK_DIAG_L_COND
WIN_CHECK_DIAG_L_X_CONT:
    beq $t6,2,WIN_CHECK_DIAG_R_REP  #is O, skip this iteration
    beq $t6,1,WIN_CHECK_DIAG_L_X_CONT_NEXT
    addi $t6,$zero,1
WIN_CHECK_DIAG_L_X_CONT_NEXT:
    addi $t5,$t5,1
    beq $t5,4,X_WIN
    beq $t5,3,WIN_CHECK_DIAG_L_X_HIGH_CHANCE
    j WIN_CHECK_DIAG_L_COND
WIN_CHECK_DIAG_L_X_HIGH_CHANCE:
    addi $a2,$zero,1
WIN_CHECK_DIAG_L_COND:
    addi $t8,$t8,1   #another round
    addi $t9,$t9,1   #until empty
    beq $t8,6,WIN_CHECK_DIAG_R_REP  #or exceed 6 rows
    beq $t9,$a0,WIN_CHECK_DIAG_R_REP     #or exceed 7 columns
    j WIN_CHECK_DIAG_L
    #check diagonally from right
WIN_CHECK_DIAG_R_REP:
    addi $t6,$zero,0     #again, preparing conditional variables
    addi $t5,$zero,0
    add $t8,$zero,$t0
    ori $t9,$zero,6 #start from right to left
    sub $t9,$t9,$t1 #subtract to have corresponding position
WIN_CHECK_DIAG_R:
    mul $t4,$t8,$a0
    add $t4,$t4,$t9
    lb $t7,midgame_rows($t4)     #get adress of the current byte
```

```
66      beq $t7,0,WIN_CHECK_HOR_REP #is empty, skip
67      beq $t7,1,WIN_CHECK_DIAG_R_X_CONT
68      #O continuous
69      beq $t6,1,WIN_CHECK_HOR_REP #is X, skip
70      beq $t6,2,WIN_CHECK_DIAG_R_O_CONT_NEXT
71      addi $t6,$zero,2
72  WIN_CHECK_DIAG_R_O_CONT_NEXT:
73      addi $t5,$t5,1
74      beq $t5,4,O_WIN
75      beq $t5,3,WIN_CHECK_DIAG_R_O_HIGH_CHANCE
76      j WIN_CHECK_DIAG_R_COND
77  WIN_CHECK_DIAG_R_O_HIGH_CHANCE:
78      addi $a3,$zero,1
79      j WIN_CHECK_DIAG_R_COND
80  WIN_CHECK_DIAG_R_X_CONT:
81      beq $t6,2,WIN_CHECK_HOR_REP #is O, skip
82      beq $t6,1,WIN_CHECK_DIAG_R_X_CONT_NEXT
83      addi $t6,$zero,1
84  WIN_CHECK_DIAG_R_X_CONT_NEXT:
85      addi $t5,$t5,1
86      beq $t5,4,X_WIN
87      beq $t5,3,WIN_CHECK_DIAG_R_X_HIGH_CHANCE
88      j WIN_CHECK_DIAG_R_COND
89  WIN_CHECK_DIAG_R_X_HIGH_CHANCE:
90      addi $a2,$zero,1
91  WIN_CHECK_DIAG_R_COND:
92      addi $t8,$t8,1  #increase row but decrease column
93      addi $t9,$t9,-1
94      beq $t8,6,WIN_CHECK_HOR_REP #exceed row, skip
95      beq $t9,-1,WIN_CHECK_HOR_REP    #exceed column, skip
96      j WIN_CHECK_DIAG_R
97  WIN_CHECK_HOR_REP:
98      slt $a1,$t1,$t3
99      beq $a1,$zero,WIN_CHECK_VERT_REP    #exceed conditions, skip to vertical
100     addi $t6,$zero,0    #preparing conditional variables
101     addi $t5,$zero,0
102     add $t8,$zero,$t0
103     add $t9,$zero,$t1
104 WIN_CHECK_HOR:
105     mul $t4,$t8,$a0
106     add $t4,$t4,$t9
107     lb $t7,midgame_rows($t4)
108     beq $t7,0,WIN_CHECK_VERT_REP    #empty, skip
109     beq $t7,1,WIN_CHECK_HOR_X_CONT
110     #O continuous
111     beq $t6,1,WIN_CHECK_VERT_REP    #is X, skip
112     beq $t6,2,WIN_CHECK_HOR_O_CONT_NEXT
113     addi $t6,$zero,2
114 WIN_CHECK_HOR_O_CONT_NEXT:
115     addi $t5,$t5,1
116     beq $t5,4,O_WIN
117     beq $t5,3,WIN_CHECK_HOR_O_HIGH_CHANCE
118     j WIN_CHECK_HOR_COND
119 WIN_CHECK_HOR_O_HIGH_CHANCE:
120     addi $a3,$zero,1
121     j WIN_CHECK_HOR_COND
122 WIN_CHECK_HOR_X_CONT:
123     beq $t6,2,WIN_CHECK_VERT_REP    #is O, skip
124     beq $t6,1,WIN_CHECK_HOR_X_CONT_NEXT
```

```
125     addi $t6,$zero,1
126 WIN_CHECK_HOR_X_CONT_NEXT:
127     addi $t5,$t5,1
128     beq $t5,4,X_WIN
129     beq $t5,3,WIN_CHECK_HOR_X_HIGH_CHANCE
130     j WIN_CHECK_HOR_COND
131 WIN_CHECK_HOR_X_HIGH_CHANCE:
132     addi $a2,$zero,1
133 WIN_CHECK_HOR_COND:
134     addi $t9,$t9,1  #increase column
135     beq $t9,$a0,WIN_CHECK_VERT_REP
136     j WIN_CHECK_HOR
137 WIN_CHECK_VERT_REP:
138     slt $a1,$t0,$t2
139     beq $a1,$zero,WIN_CHECK_COND    #exceed condition, skip
140     addi $t6,$zero,0    #preparing conditional variables
141     addi $t5,$zero,0
142     add $t8,$zero,$t0
143     add $t9,$zero,$t1
144 WIN_CHECK_VERT:
145     mul $t4,$t8,$a0 #get the correct address
146     add $t4,$t4,$t9
147     lb $t7,midgame_rows($t4)    #get byte
148     beq $t7,0,WIN_CHECK_COND    #empty, skip
149     beq $t7,1,WIN_CHECK_VERT_X_CONT
150     #O continuous
151     beq $t6,1,WIN_CHECK_COND    #is X, skip
152     beq $t6,2,WIN_CHECK_VERT_O_CONT_NEXT
153     addi $t6,$zero,2
154 WIN_CHECK_VERT_O_CONT_NEXT:
155     addi $t5,$t5,1
156     beq $t5,4,O_WIN
157     beq $t5,3,WIN_CHECK_VERT_O_HIGH_CHANCE
158     j WIN_CHECK_VERT_COND
159 WIN_CHECK_VERT_O_HIGH_CHANCE:
160     addi $a3,$zero,1
161     j WIN_CHECK_VERT_COND
162 WIN_CHECK_VERT_X_CONT:
163     beq $t6,2,WIN_CHECK_COND    #is O, skip
164     beq $t6,1,WIN_CHECK_VERT_X_CONT_NEXT
165     addi $t6,$zero,1
166 WIN_CHECK_VERT_X_CONT_NEXT:
167     addi $t5,$t5,1
168     beq $t5,4,X_WIN
169     beq $t5,3,WIN_CHECK_VERT_X_HIGH_CHANCE
170     j WIN_CHECK_VERT_COND
171 WIN_CHECK_VERT_X_HIGH_CHANCE:
172     addi $a2,$zero,1
173 WIN_CHECK_VERT_COND:
174     addi $t8,$t8,1  #increase the row
175     bne $t8,6,WIN_CHECK_VERT
176 WIN_CHECK_COND:
177     addi $t1,$t1,1
178     bne $t1,7,WIN_CHECK_OP  #condition checking for iteration
179     addi $t1,$zero,0
180     addi $t0,$t0,1
181     bne $t0,6,WIN_CHECK_OP
```

Right after that, we have a section for check if the board is tied or not. Tie condition is simple, if all of the bytes are occupied and after checking winning conditions, there are still no players who win, the game is draw.

```
    #begin checking for the tie condition
    add $t0,$zero,$zero
    add $t1,$zero,$zero
TIE_CHECK_OP:
    lb $t2,midgame_rows($t1)     #load  byte
    beq $t2,0,TIE_CHECK_EXIT    #if empty, return to game
    addi $t0,$t0,1
    addi $t1,$t1,1  #increase the iterator by 1
    bne $t1,42,TIE_CHECK_OP #condition checking
    beq $t0,$t1,TIE
TIE_CHECK_EXIT:
    jr $ra
```

Another type of winning condition is when a player have violated rules more than 3 times. Types of rule violation includes: removing your piece, removing an empty location, adding to an already full column, blocking an opponent who has high chances of winning, trying to use an already used up ability and lastly, adding to an out of range column. The violation number will be checked and if equal to 3, result in the winning of the other player. However, these functions are only invoked depending on players' inputs, not sequentially.

```
PROC_ERR_YOURS:
    #the piece is yours, can't remove
    addi $v0,$zero,4
    la $a0,error_yours
    syscall
    j RULES_VIOLATED

PROC_ERR_EMPTY:
    #location is empty, can't remove
    addi $v0,$zero,4
    la $a0,error_empty
    syscall
    j RULES_VIOLATED

PROC_ERR_FULL_ROW:
    #column is full
    addi $v0,$zero,4
    la $a0,error_full_row
    syscall
    j RULES_VIOLATED

PROC_ERR_BLOCK:
    #high chance of winning, cant block
    addi $v0,$zero,4
    la $a0,error_block
    syscall
    j RULES_VIOLATED

PROC_ERR_USED:
    #already used up the ability
    addi $v0,$zero,4
    la $a0,error_used
    syscall
```

```
34      j RULES_VIOLATED
35
36  PROC_ERR:
37      # wrong format error handling
38      addi $v0,$zero,4
39      la $a0,error_wrong_string
40      syscall
41      j RULES_VIOLATED
42
43  RULES_VIOLATED:
44      la $a0,start_game    #press anykey to continue
45      syscall
46      addi $v0,$zero,8
47      la $a0,misc_mem
48      addi $a1,$zero,1
49      syscall
50      lb $t0,($s2)     #checking for whose turn to know who violates
51      beq $t0,$zero,PROC_ERR_P1    #the code for p1 is similar to p2
52      #player 2 is violating rules
53      lb $t0,7($s1)    #get and increase the number of violation
54      addi $t0,$t0,1
55      addi $t1,$zero,3
56      beq $t0,$t1,P2_VIO_LOST #if equal to 3, p2 lost
57      sb $t0,7($s1)
58      j MID_GAME
```

6 First 2 moves:

The first 2 moves are rather simple, just let player input the column and check if it is 4 or not. If it is 4 then set and if it is not then link it to the rule violation functions. Here, the 2 players move sequentially, player 1 and then player 2, so we just let 2 functions of first moves and if any errors occurred, deal with it according to the player. The code listed below are just for player 1 because for player 2, it is pretty much the same.

```
1   PROC_MOVE1:
2       addi $v0,$zero,4
3       lb $t0,($s2)
4       la $a0,start_player1
5       syscall #signifying player 1's turn
6       la $a0,space
7       syscall
8       la $a0,start_your_turn
9       syscall
10      la $a0,start_input_first_move
11      syscall #this is your first move
12      la $a0,start_input
13      syscall
14      addi $v0,$zero,8
15      la $a0,misc_mem
16      addi $a1,$zero,2
17      syscall
18      addi $v0,$zero,4
19      lb $t3,($a0)
20      la $a0,eol
21      syscall
```

```
22      addi $t0,$zero,82    #trying to use ability is violation
23      beq $t3,$t0,PROC_ERR_ABILITY_M1
24      addi $t0,$zero,66
25      beq $t3,$t0,PROC_ERR_ABILITY_M1
26      addi $t0,$zero,52    #not column 4 is violation
27      bne $t3,$t0,PROC_ERR_M1
28      lb $t1,($s3)
29      addi $t2,$zero,1
30      addi $t3,$zero,3
31      beq $t1,0,M1_X_SET  #set according to their pieces
32      sb $t3,38($s0)  #set to column 4
33      jr $ra
34  M1_X_SET:
35      sb $t2,38($s0)  #set to column 4
36      jr $ra
```

7 Full game:

Each full game iteration involves calling printing GUI function, printing board, processing, winning condition checking and changing turns. Having looked at 3 of the 5 listed, we now looked into the process function.

At the beginning of each move, the data is always backed up at the adjacent data array. If this is the turn of a blocked player, the turn immediately changed to another player's. If a player already used block or remove, the system will not print the ability anymore and any attempts at using it will result in rule violation. If a player drop a piece outside the range of [1;7] will also result in rule violation. Opting remove a piece but wrong location is also violating rule. Any rule violation will just increase the number of violation but the turn is still belong to the current player. If a player input a correct column, the system will look for an empty row from the end of the array up.

Finally, if a player has made his move, if there are any undo left, the system will ask if he wanted to undo the move or not. If yes, then copy the backed up data into the main data, decrease the number of undo and restart the move. The undo procedure is just load data from the backed up data and restore into the main data.

```
1   PROCESS:
2       #backup the entire data set
3       add $t0,$zero,$zero
4       addi $t1,$zero,52
5       beq $s6,$zero,BACKUP_OP #s6 means someone have recently used undo
6       addi $v0,$zero,4
7       la $a0,start_input_undo_success
8       syscall
9       la $a0,tab
10      syscall
11      add $s6,$zero,$zero
12  BACKUP_OP:
13      lb $t2,midgame_rows($t0)
14      sb $t2,backup_data($t0)
15      addi $t0,$t0,1
16      bne $t0,$t1,BACKUP_OP
```

```
17      #checking for whose turn, is blocked or not
18      addi $v0,$zero,4
19      lb $t0,($s2)
20      beq $t0,1,P2_TURN
21      lb $t0,8($s1)
22      beq $t0,$zero,P1_TURN
23      la $a0,start_player1
24      syscall
25      la $a0,space
26      syscall
27      la $a0,start_is_blocked
28      syscall
29      addi $t0,$zero,1     #is blocked, so change turn to another player 2's
30      sb $t0,($s2)
31      sb $zero,8($s1) #already lost a move, return the normal status
32      j P2_TURN_START
33  P1_TURN:
34      addi $v0,$zero,4
35      la $a0,start_player1
36      syscall
37      la $a0,space
38      syscall
39      la $a0,start_your_turn
40      syscall
41      lb $t0,1($s1)
42      beq $t0,$zero,P1_SKIP_REM    #if used remove, do not print use remove
43      la $a0,start_input_remove
44      syscall
45  P1_SKIP_REM:
46      lb $t0,2($s1)
47      beq $t0,$zero,PROC_CONT
48      la $a0,start_input_block     #if used block, do not print use block
49      syscall
50      j PROC_CONT
51  P2_TURN:
52      addi $v0,$zero,4     #checking if p2 is blocked or not
53      lb $t0,9($s1)
54      beq $t0,$zero,P2_TURN_START
55      la $a0,start_player2
56      syscall
57      la $a0,space
58      syscall
59      la $a0,start_is_blocked
60      syscall
61      sb $zero,($s2)  #change turn to p1's
62      sb $zero,9($s1) #blocked, changed turn so return p2 the normal status
63      j P1_TURN
64  P2_TURN_START:
65      addi $v0,$zero,4
66      la $a0,start_player2
67      syscall
68      la $a0,space
69      syscall
70      la $a0,start_your_turn
71      syscall
72      lb $t0,5($s1)
73      beq $t0,$zero,P2_SKIP_REM    #if used remove, dont print use remove
74      la $a0,start_input_remove
75      syscall
```

```
P2_SKIP_REM:
    lb $t0,6($s1)
    beq $t0,$zero,PROC_CONT #if used block, dont print use block
    la $a0,start_input_block
    syscall
PROC_CONT:
    la $a0,start_input
    syscall
    addi $v0,$zero,8
    la $a0,misc_mem
    addi $a1,$zero,2    #input column or ability to use
    syscall
    addi $v0,$zero,4
    lb $t3,($a0)
    la $a0,eol
    syscall
    addi $t0,$zero,82   #82 is R
    beq $t3,$t0,PROC_REM
    addi $t0,$zero,66   #66 is B
    bne $t3,$t0,PROC_SKIP_BLOCK
    #block opponent
    addi $v0,$zero,4
    lb $t0,($s2)
    beq $t0,$zero,PROC_P1_BLOCK_P2
    #P2 block P1
    lb $t0,6($s1)
    beq $t0,$zero,PROC_ERR_USED #check if it is used or not
    lb $t9,($s3)
    beq $t9,$zero,PROC_P2_BLOCK_P1_X    #P1 is X
    bne $a3,$zero,PROC_ERR_BLOCK    #P1 is O so if P1 have high chance, cant block
    j PROC_P2_BLOCK_P1_SKIP
PROC_P2_BLOCK_P1_X:
    bne $a2,$zero,PROC_ERR_BLOCK    #P1 is X so if P1 have high chance, cant block
PROC_P2_BLOCK_P1_SKIP:
    sb $zero,6($s1)
    addi $t0,$zero,1
    sb $t0,8($s1)
    addi $v0,$zero,4
    la $a0,start_block_success
    syscall #block success, get to drop a piece before end turn
    lb $t0,4($s1)
    beq $t0,$zero,P2_TURN_START
    la $a0,start_confirm_setting    #asked if the player wanted to undo or not
    syscall
    addi $v0,$zero,8
    addi $a1,$zero,2
    la $a0,misc_mem
    syscall
    lb $t0,($a0)
    addi $t1,$zero,49
    bne $t0,$t1,PROCESS
    lb $t0,backup_data+46   #wanted undo, so decrease number of undo before
        reloading data
    addi $t0,$t0,-1
    sb $t0,backup_data+46
    j UNDO
PROC_P1_BLOCK_P2:
    lb $t0,2($s1)
    beq $t0,$zero,PROC_ERR_USED
```

```
134     lb $t9,($s3)
135     bne $t9,$zero,PROC_P1_BLOCK_P2_X     #P2 is X
136     bne $a3,$zero,PROC_ERR_BLOCK     #P2 is O so if P2 have high chance, cant block
137     j PROC_P1_BLOCK_P2_SKIP
138 PROC_P1_BLOCK_P2_X:
139     bne $a2,$zero,PROC_ERR_BLOCK     #P2 is X so if P2 have high chance, cant block
140 PROC_P1_BLOCK_P2_SKIP:
141     sb $zero,2($s1)
142     addi $t0,$zero,1
143     sb $t0,9($s1)
144     addi $v0,$zero,4
145     la $a0,start_block_success
146     syscall #block success, get to drop a piece before end turn
147     lb $t0,($s1)
148     beq $t0,$zero,P1_TURN
149     la $a0,start_confirm_setting     #asked if the player wanted to undo or not
150     syscall
151     addi $v0,$zero,8
152     addi $a1,$zero,2
153     la $a0,misc_mem
154     syscall
155     lb $t0,($a0)
156     addi $t1,$zero,49
157     bne $t0,$t1,PROCESS
158     lb $t0,backup_data+42    #wanted undo, so decrease number of undo before
            reloading data
159     addi $t0,$t0,-1
160     sb $t0,backup_data+42
161     j UNDO
162 PROC_SKIP_BLOCK:
163     addi $t0,$zero,49    #drop pieces normally
164     addi $t1,$zero,55
165     slt $t2,$t3,$t0 #row format checking
166     bne $t2,$zero,PROC_ERR
167     slt $t2,$t1,$t3
168     bne $t2,$zero,PROC_ERR
169     #check whether the column is full or not
170     addi $t5,$t3,-7
171 PROC_LOOP2: #Loop2: find empty spot
172     addi $t5,$t5,-7
173     slt $t1,$t5,$zero
174     bne $t1,$zero,PROC_ERR_FULL_ROW
175     lb $t7,midgame_rows($t5)
176     beq $t7,0,PROC_LOOP2_EXIT
177     j PROC_LOOP2
178 PROC_LOOP2_EXIT:
179     lb $t0,($s2)
180     lb $t1,($s3)
181     addi $t2,$zero,1
182     addi $t3,$zero,3
183     beq $t0,1,P2_SET     #set location according to player
184     beq $t1,0,X_SET #set location according to piece
185     j O_SET
186 P2_SET:
187     beq $t1,1,X_SET
188 O_SET:
189     sb $t3,midgame_rows($t5)     #set location
190     j PROC_CONT_CHECK
191 X_SET:
```

```
192     sb $t2,midgame_rows($t5)
193 PROC_CONT_CHECK:
194     lb $t0,($s2)     #check whose turn is it
195     beq $t0,$zero,P1_TURN_UNDO
196     lb $t0,4($s1)    #check if p2 have any undo left
197     beq $t0,$zero,PROC_EXIT
198     jal PRINT_GUI    #if yes, print GUI and board again
199     jal PRINT_BOARD #then ask if he wanted to undo or not
200     addi $v0,$zero,4
201     la $a0,start_input_undo_ask
202     syscall
203     la $a0,start_confirm_setting
204     syscall
205     addi $v0,$zero,8
206     la $a0,misc_mem
207     addi $a1,$zero,2
208     syscall
209     lb $t0,($a0)
210     addi $t1,$zero,49
211     bne $t0,$t1,PROC_EXIT
212     lb $t2,backup_data+46    #if yes, reduce number of undo and reload data
213     addi $t2,$t2,-1
214     sb $t2,backup_data+46
215     j UNDO
216 P1_TURN_UNDO:
217     lb $t0,($s1)     #check if p1 have any undo left
218     beq $t0,$zero,PROC_EXIT
219     jal PRINT_GUI    #if yes, print GUI and board again
220     jal PRINT_BOARD #then ask if he wanted to undo or not
221     addi $v0,$zero,4
222     la $a0,start_input_undo_ask
223     syscall
224     la $a0,start_confirm_setting
225     syscall
226     addi $v0,$zero,8
227     la $a0,misc_mem
228     addi $a1,$zero,2
229     syscall
230     lb $t0,($a0)
231     addi $t1,$zero,49
232     bne $t0,$t1,PROC_EXIT
233     lb $t2,backup_data+42    #if yes, reduce number of undo and reload data
234     addi $t2,$t2,-1
235     sb $t2,backup_data+42
236 UNDO:
237     add $t0,$zero,$zero #undo iterator
238     addi $t1,$zero,52    #undo loop condition
239 UNDO_OP:
240     lb $t2,backup_data($t0) #load in backup and restore
241     sb $t2,midgame_rows($t0)
242     addi $t0,$t0,1
243     bne $t0,$t1,UNDO_OP
244     addi $s6,$zero,1
245     jal PRINT_GUI
246     jal PRINT_BOARD
247     j PROCESS
248 PROC_EXIT:
249     j CHECK_CHECK    #check winning condition and change turns
```

The last ability of the players to be implemented is the removal of 1 opponent's piece. This ability is invoked by input R into the I/O and the player is then asked to input a row and column coordinate, provided that they have not used it up. The system will check according to the idea and the system is then process the process of blocks falling down or rule violation depending on the player's input.

```
PROC_REM:
    lb $t0,($s2)     #check for whose turn
    beq $t0,$zero,P1_REMOVE #if it is P1's, its their removal
    #P2 remove
    lb $t0,5($s1)    #load the byte to check if it is used up
    beq $t0,$zero,PROC_ERR_USED
    addi $v0,$zero,4
    la $a0,start_input_remove_coordinate
    syscall
    la $a0,remove_coordinate_x
    syscall
    addi $v0,$zero,8     #enter the X coordinate
    la $a0,misc_mem
    addi $a1,$zero,2
    syscall
    lb $t1,($a0)
    addi $v0,$zero,4
    la $a0,eol
    syscall
    la $a0,remove_coordinate_y
    syscall
    addi $v0,$zero,8     #enter the Y coordinate
    la $a0,misc_mem
    syscall
    lb $t2,($a0)
    addi $v0,$zero,4
    la $a0,eol
    syscall
    slt $t5,$t1,$t4
    bne $t5,$zero,PROC_ERR
    addi $t4,$zero,54    #if row is not in range of 1-6, error
    slt $t5,$t4,$t1
    bne $t5,$zero,PROC_ERR
    addi $t4,$zero,49
    slt $t5,$t2,$t4
    bne $t5,$zero,PROC_ERR
    addi $t4,$zero,55    #if column is not in range of 1-7, error
    slt $t5,$t4,$t2
    bne $t5,$zero,PROC_ERR
    #load to check if it belongs to the other player or current player
    addi $t1,$t1,-49
    addi $t2,$t2,-49
    addi $t3,$zero,7
    mul $t5,$t1,$t3
    add $t5,$t5,$t2
    lb $t4,midgame_rows($t5)
    lb $t7,($s3)     #load X-O
    beq $t7,$zero,P2O_REM
    addi $t6,$zero,1     #P2 is X
    j P2REM_CONT
```

```
P2O_REM:
    addi $t6,$zero,3
P2REM_CONT:
    beq $t6,$t4,PROC_ERR_YOURS  #if it is current player's piece, error
    beq $zero,$t4,PROC_ERR_EMPTY    #if it is empty, error
    sb $zero,5($s1) #else, success and deplete the use
    sb $zero,midgame_rows($t5)
    j REMOVED_SUCCESSFULLY
P1_REMOVE:
    lb $t0,1($s1)   #load the byte to check if it is used up
    beq $t0,$zero,PROC_ERR_USED
    addi $v0,$zero,4
    la $a0,start_input_remove_coordinate
    syscall
    la $a0,remove_coordinate_x
    syscall
    addi $v0,$zero,8    #input the X coordinate
    la $a0,misc_mem
    addi $a1,$zero,2
    syscall
    lb $t1,($a0)
    addi $v0,$zero,4
    la $a0,eol
    syscall
    la $a0,remove_coordinate_y
    syscall
    addi $v0,$zero,8    #input the Y coordinate
    la $a0,misc_mem
    syscall
    lb $t2,($a0)
    addi $v0,$zero,4
    la $a0,eol
    syscall
    slt $t5,$t1,$t4
    bne $t5,$zero,PROC_ERR
    addi $t4,$zero,54   #if row is not in range of 1-6, error
    slt $t5,$t4,$t1
    bne $t5,$zero,PROC_ERR
    addi $t4,$zero,49
    slt $t5,$t2,$t4
    bne $t5,$zero,PROC_ERR
    addi $t4,$zero,55   #if column is not in range of 1-7, error
    slt $t5,$t4,$t2
    bne $t5,$zero,PROC_ERR
    #load to check if it belongs to the other player or current player
    addi $t1,$t1,-49
    addi $t2,$t2,-49
    addi $t3,$zero,7
    mul $t5,$t1,$t3
    add $t5,$t5,$t2
    lb $t4,midgame_rows($t5)
    lb $t7,($s3)    #load X-O
    beq $t7,$zero,P1X_REM
    addi $t6,$zero,3    #P1 is O
    j P1REM_CONT
P1X_REM:
    addi $t6,$zero,1
P1REM_CONT:
    beq $t6,$t4,PROC_ERR_YOURS  #if it is current player's piece, error
```

```
110     beq $zero,$t4,PROC_ERR_EMPTY     #if it is empty, error
111     sb $zero,1($s1)  #else, success and deplete the use
112     sb $zero,midgame_rows($t5)
113 REMOVED_SUCCESSFULLY:
114     addi $v0,$zero,4
115     la $a0,remove_success
116     syscall
```

After that, the game move to the function in charge of taking care if block falling down involve first checking the last line (7 iterations for 7 columns, checking from right to left), if there are any empty place, check the row immediately above it in the same column, and if it is also empty, the process move up the column (reduce row) and until out of top, can move on to another column, if not then store the byte in the empty place and empty the place and continue checking in the same manner until there are 2 consecutively empty place or out of the roof (exceed the condition).

```
1     addi $t0,$zero,6     #initialize condition to check for block falling, t0 is the
          last column
2     addi $t1,$zero,5     #check last row first
3     addi $t2,$zero,7
4 REM_SUC_OP:
5     add $t5,$zero,$t1
6 REM_SUC_FALLS:
7     mul $t3,$t2,$t5
8     add $t3,$t3,$t0
9     lb $t4,midgame_rows($t3)
10    bne $t4,$zero,REM_SUC_COND  #if the place is occupied, skip
11    addi $t6,$t3,-7 #the place is empty, check for any row above it is empty
12    slt $t7,$t6,$zero    #stopping condition is when check full of row
13    addi $t8,$zero,1
14    beq $t7,$t8,REM_SUC_COND
15    lb $t9,midgame_rows($t6)     #load the byte on top of the previously checked
          byte
16    beq $t9,$zero,REM_SUC_COND  #if empty, skip
17    sb $t9,midgame_rows($t3)     #not empty, falls down
18    sb $zero,midgame_rows($t6)
19    addi $t5,$t5,-1
20    j REM_SUC_FALLS #also, check again to be sure
21 REM_SUC_COND:
22    beq $t0,$zero,REM_SUC_OUT_COND
23    addi $t0,$t0,-1 #gradually move to top
24    j REM_SUC_OP
25 REM_SUC_OUT_COND:
26    addi $t1,$t1,-1 #jump to adjacent left column
27    beq $t1,$zero,REM_SUC_EXIT
28    addi $t0,$zero,6     #recheck from bottom
29    j REM_SUC_OP
30 REM_SUC_EXIT:
31    j PROC_CONT_CHECK    #return to check the undo
```

Change turn procedure is simple, just load value from the address at register s2 (which holds the array that indicate which player's turn), check this turn belongs to which player before changing it to 0 (player 1's turn) or 1 (player 2's turn).

```
1     lb $t0,($s2)     #0 is P1's turn, 1 is P2's turn
```

```
    beq $t0,$zero,CHANGE_TURN_0
    sb $zero,($s2)
    j MID_GAME
CHANGE_TURN_0:
    addi $t0,$t0,1
    sb $t0,($s2)
    j MID_GAME
```

8 Endgame:

Finally, the endgame involves listing out the 2 main outcomes: 1 player win, the game is tied. These functions pretty much involve only output.

```
TIE:
    addi $v0,$zero,4
    la $a0,game_tie #the game is tied
    syscall
    j END_GAME
```

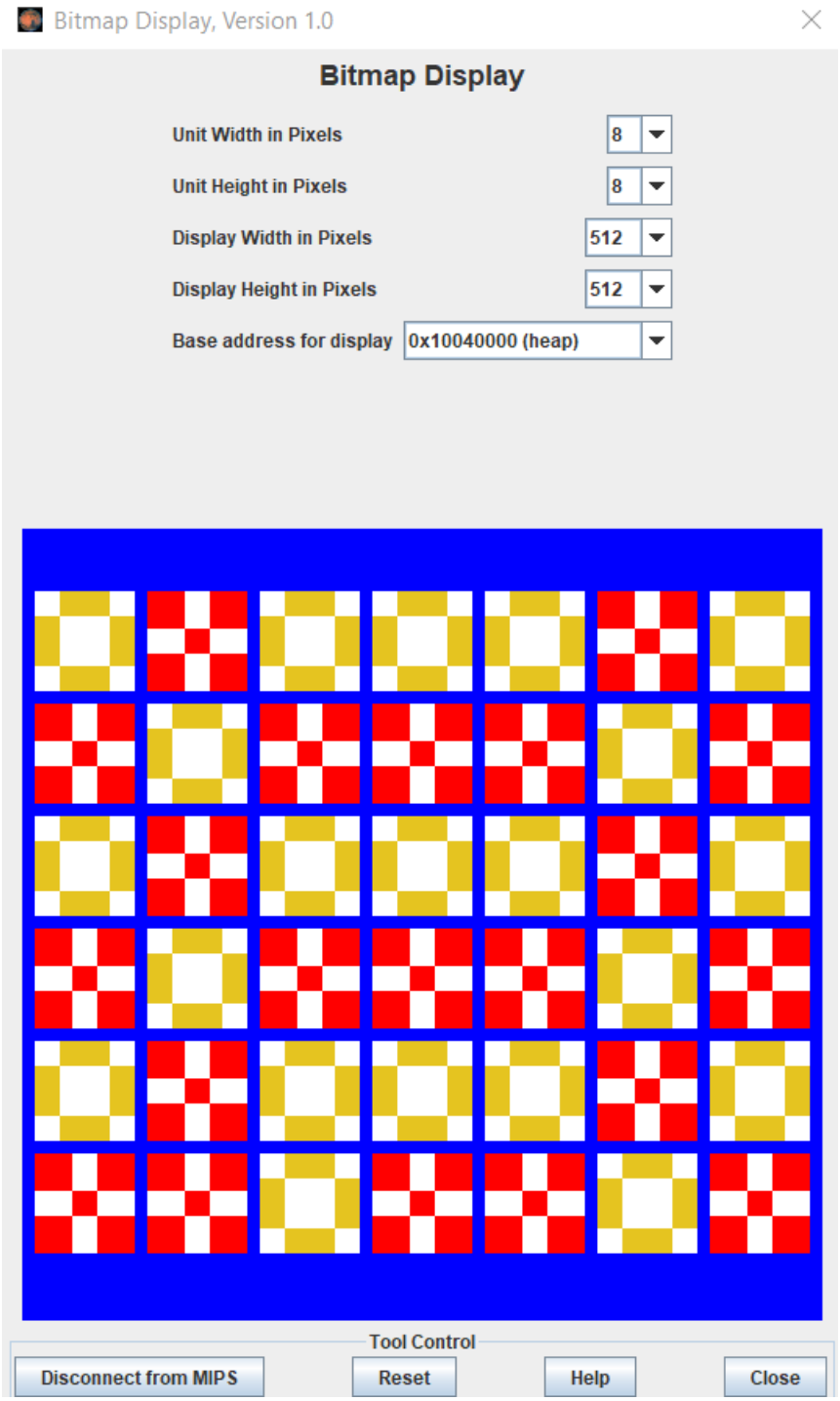

Figure 2.6: A tied game

```
P1_VIO_LOST:
    addi $v0,$zero,4     #p1 is lost due to 3 violations
    la $a0,start_player1     #this function is invoked by rule violation function
    syscall
    la $a0,space
    syscall
    la $a0,midgame_name1
    syscall
    la $a0,eol
    syscall
    la $a0,lose_violation
    syscall
    j P2_WIN

P2_VIO_LOST:
    addi $v0,$zero,4     #p2 is lost due to 3 violations
    la $a0,start_player2     #this function is invoked by rule violation function
    syscall
    la $a0,space
    syscall
    la $a0,midgame_name2
    syscall
    la $a0,eol
    syscall
    la $a0,lose_violation
    syscall
    j P1_WIN

X_WIN:
    lb $t0,($s3)     #this function is invoked by the winning condition checking
        function
    beq $t0,0,P1_WIN
    j P2_WIN

O_WIN:
    lb $t0,($s3)     #this function is invoked by the winning condition checking
        function
    beq $t0,1,P1_WIN
    j P2_WIN

P1_WIN:
    addi $v0,$zero,4     #announce that p1 wins
    la $a0,start_player1
    syscall
    la $a0,space
    syscall
    la $a0,midgame_name1
    syscall
    la $a0,space
    syscall
    la $a0,wins
    syscall
    j END_GAME

P2_WIN:
    addi $v0,$zero,4     #announce that p2 wins
    la $a0,start_player2
    syscall
    la $a0,space
```

```
58      syscall
59      la $a0,midgame_name2
60      syscall
61      la $a0,space
62      syscall
63      la $a0,wins
64      syscall
65      j END_GAME
```

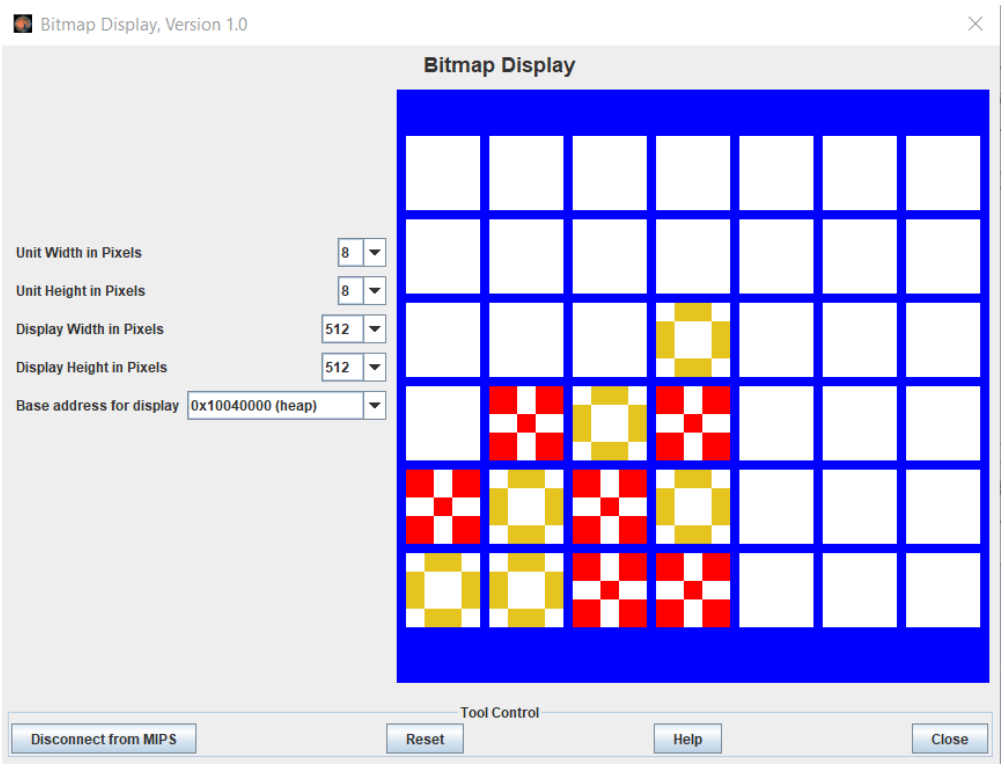

Figure 2.7: A normal game

The system then ask if the players wanted to restart the game by input 1, which will take the system return to the initialization function and keep doing the same manner as discussed.

```
1  END_GAME:
2       addi $v0,$zero,4      #ask if the players wanted to restart
3       addi $t1,$zero,49
4       la $a0,start_exit
5       syscall
6       ori $v0,$zero,8
7       la $a0,misc_mem
8       addi $a1,$zero,2      #input
9       syscall
10      lb $t0,($a0)      #check for input
11      beq $t0,$t1,START     #if 1 then restart
12 EXIT:
13      ori $v0,$zero,10      #if not, exit
14      syscall
```