



UNIVERSIDADE FEDERAL DO CARIRI

**UNIVERSIDADE FEDERAL DO CARIRI
CENTRO DE CIÊNCIAS E TECNOLOGIA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO**

**Projeto 01 - Minesweeper em Assembly
Professor: Ramon Santos Nepomuceno**

Wanderson Faustino Patricio

Juazeiro do Norte, 19 de fevereiro de 2024

1 Introdução

Campo minado é um popular jogo de computador para um jogador. Foi inventado por Robert Donner em 1989 e tem como objectivo revelar um campo de minas sem que alguma seja detonada. Este jogo tem sido reescrito para as mais diversas plataformas, sendo a sua versão mais popular a que vinha nativamente nas edições anteriores ao Windows 10.

1.1 Regras

A área de jogo consiste num campo de quadrados retangular. Cada quadrado pode ser revelado clicando sobre ele, e se o quadrado clicado contiver uma mina, então o jogo acaba. Se, por outro lado, o quadrado não contiver uma mina, uma de duas coisas poderá acontecer:

1. Um número aparece, indicando a quantidade de quadrados adjacentes que contêm minas;
2. Nenhum número aparece. Neste caso, o jogo revela automaticamente os quadrados que se encontram adjacentes ao quadrado vazio, já que não podem conter minas;

O jogo é ganho quando todos os quadrados que não têm minas são revelados.

1.2 Implementação

O jogo foi implementado em linguagem assembly através do simulador MARS (MIPS Assembler and Runtime Simulator), desenvolvido pelo Missouri State University. A parte gráfica do jogo é apresentada através do console.

Quando uma casa está marcada como não revelada é mostrado um sinal de '#' na célula correspondente, caso seja revelada e seja uma bomba é mostrado um símbolo de '*'.

Ao ser revelado uma célula que não é bomba é contabilizada a quantidade de bombas nas células vizinhas e a célula mostra o valor calculado.

	0	1	2	3	4	5	6	7	8	9
0		#	#	#	#	#	#	#	#	#
1		#	#	#	#	#	#	#	#	#
2		#	#	#	#	#	#	#	#	#
3		#	#	#	#	#	#	#	#	#
4		#	#	#	#	#	#	#	#	#
5		#	#	#	#	#	#	#	#	#
6		#	#	#	#	#	#	#	#	#
7		#	#	#	#	#	#	#	#	#
8		#	#	#	#	#	#	#	#	#
9		#	#	#	#	#	#	#	#	#
Enter the row for the move: 5										
Enter the column for the move: 2										

	0	1	2	3	4	5	6	7	8	9
0		0	0	0	0	0	0	0	0	0
1		1	1	1	0	0	0	0	0	0
2		1	#	2	1	0	0	0	0	0
3		1	2	#	1	0	0	0	0	0
4		0	1	1	2	1	1	0	0	0
5		0	0	0	1	#	1	0	0	0
6		0	0	0	2	2	3	1	1	1
7		0	0	0	1	#	2	#	2	#
8		1	1	1	2	3	4	2	2	#
9		#	#	#	#	#	1	1	1	1
Enter the row for the move:										

Figura 1: Visão do console no simulador

2 Revisão de códigos

O repositório do game foi dividido em nove arquivos: macros.asm, initializeBoard.asm, plantBombs.asm, printBoard.asm, play.asm, countAdjacentBombs.asm, revealAdjacentCells.asm, checkVictory.asm e main.asm

2.1 macros.asm

```
1 .eqv SIZE 8
2 .eqv BOMB_COUNT 10
3
4 .macro save_context
5     addi $sp, $sp, -36
6     sw $s0, 0 ($sp)
7     sw $s1, 4 ($sp)
8     sw $s2, 8 ($sp)
9     sw $s3, 12 ($sp)
10    sw $s4, 16 ($sp)
11    sw $s5, 20 ($sp)
12    sw $s6, 24 ($sp)
13    sw $s7, 28 ($sp)
14    sw $ra, 32 ($sp)
15 .end_macro
16
17 .macro restore_context
18     lw $s0, 0 ($sp)
19     lw $s1, 4 ($sp)
20     lw $s2, 8 ($sp)
21     lw $s3, 12 ($sp)
22     lw $s4, 16 ($sp)
23     lw $s5, 20 ($sp)
24     lw $s6, 24 ($sp)
25     lw $s7, 28 ($sp)
26     lw $ra, 32 ($sp)
27     addi $sp, $sp, 36
28 .end_macro
29
30 .macro get_ij_address
31     //esse macro retorna a posição [i][j] de um array bidimensional de
32     //inteiros com tamanho SIZE de linha
33
34     move $t1, $a1           // a1 := i
35     move $t2, $a2           // a2 := j
36     li $t3, SIZE
37
38     mul $t0, $t1, $t3
39     add $t0, $t0, $t2
40
41     // t0 := i*SIZE + j
42     //4 == sizeof(int)
43     mul $t0, $t0, 4
44 .end_macro
```

Esse arquivo é responsável por alguns macros ao decorrer do programa.

Foram definidas duas constantes para serem utilizadas: SIZE, que define o tamanho do tabuleiro, e BOMB_COUNT, que define a quantidade de bombas que serão escondidas no tabuleiro.

As macros save_context e restore_context respectivamente armazenam e restauram os valores dos registradores *s0* a *s7*, 4 bytes para cada número. Essas macros são utilizadas ao início e ao final de cada função para impedir que valores que serão utilizados em outros pontos do código sejam perdidos.

A macro `get_ij_address` retorna a quantidade de memória a ser avançada a partir da posição inicial do tabuleiro. Se considerarmos que no registrador *a1* está armazenado qual a linha (row), em *a2* a coluna (col) e que o tabuleiro tem tamanho *SIZE* devemos retornar o valor $row * SIZE + col$, como cada palavra é armazenada em 4 bytes precisamos multiplicar o resultado por 4. O retorno é dado no registrador temporário *t0*.

2.2 initializeBoard.asm

```

1  void initializeBoard(int board[][SIZE]) {
2      // Initializes the board with zeros
3      for (int i = 0; i < SIZE; ++i) {
4          for (int j = 0; j < SIZE; ++j) {
5              board[i][j] = -2; // -2 means no bomb
6          }
7      }
8  }
```

A função **initializeBoard** é responsável por iniciar todo o tabuleiro sem bombas. Como estamos trabalhando com um array de inteiros, o código para uma célula sem bomba é -2 . É necessário fazer dois loops aninhados para percorrer as linhas e as colunas.

```

1  .include "macros.asm"
2  .globl initializeBoard
3
4  initializeBoard:
5      save_context
6
7      move $s0, $a0          //s0 := &board
8
9      li $s1, 0              // i = 0
10     begin_ib_i_loop:
11         li $t0, SIZE
12         bge $s1, $t0, end_ib_i_loop
13
14         li $s2, 0           //j = 0
15         begin_ib_j_loop:
16             li $t0, SIZE
17             bge $s2, $t0, end_ib_j_loop
18
19             move $a1, $s1
20             move $a2, $s2
21             get_ij_address
22             add $t0, $t0, $s0
23             li $t1, -2       //-2 means no bomb
24             sw $t1, 0($t0)    //board[i][j] = -2
25
26             addi $s2, $s2, 1
27             j begin_ib_j_loop
28         end_ib_j_loop:
29
30         addi $s1, $s1, 1
31         j begin_ib_i_loop
32     end_ib_i_loop:
33
34     restore_context
35     jr $ra
```

Inicialmente é chamada a macro `save_context` e armazenado em *s0* o valor do endereço de memória do tabuleiro (*&board*). Para a iteração em linhas (*i*) e colunas (*j*) utilizaremos os regis-

tradadores $s1 \equiv i$ e $s2 \equiv j$, iniciamos o valor em 0 (li \$s1, 0). Ao entrar no loop carregamos o valor de SIZE no registrador $t0$ e comparamos se o valor de i é maior ou igual a SIZE, caso seja pulamos para fora do loop em i . Ao final do loop em i somamos uma unidade ao registrador \$s1 (addi \$s1, \$s1, 1) e retornamos ao início do loop. O mesmo procedimento é repetido para o loop em j com o registrador \$s2. Como esse procedimento é repetido várias vezes no programa utilizaremos a mesma rotina.

Dentro do loop mais interno carregamos em $a1$ o valor de i e em $a2$ o valor de j e calculamos a posição da célula $[i][j]$ pela macro get_ij_address e somamos ao endereço do início do tabuleiro. Após calcular a posição final da célula armazenamos o valor de -2 na referida posição de memória.

Ao final do código restauramos os valores nos registradores $s0$ a $s7$ e retornamos para a parte do programa que chamou a função, que está armazenada no registrador \$ra.

2.3 plantBombs.asm

```

1  void placeBombs(int board[][SIZE]) {
2      srand(time(NULL));
3      // Places bombs randomly on the board
4      for (int i = 0; i < BOMB_COUNT; ++i) {
5          int row, column;
6          do {
7              row = rand() % SIZE;
8              column = rand() % SIZE;
9          } while (board[row][column] == -1);
10         board[row][column] = -1; // -1 means bomb present
11     }
12 }

```

Devemos repetir um loop BOMB_COUNT vezes, escolher uma posição de memória aleatória dentro do tabuleiro. Se aquela posição já tiver uma bomba nós repetimos esse processo. Ao encontrarmos uma célula que não contém bomba carregamos uma bomba lá, armazenando o valor -1

```

1  .include "macros.asm"
2
3  .globl plantBombs
4
5  plantBombs:
6      save_context
7      move $s0, $a0
8
9      li $s1, 0    # i = 0
10     begin_for_i_pb:          // for (int i = 0; i < BOMB_COUNT; ++i) {
11         li $t0, BOMB_COUNT
12         bge $s1, $t0, end_for_i_pb
13
14         do_cb:                // do {
15             li $v0, 42
16
17             li $a0, 0          # srand(time(NULL));
18             li $a1, SIZE
19             syscall
20             move $s2, $a0      // row = rand() % SIZE;
21
22             li $a0, 0          # srand(time(NULL));
23             li $a1, SIZE
24             syscall
25             move $s3, $a0      // column = rand() % SIZE;

```

```

26
27         move $a1, $s2
28         move $a2, $s3
29         get_ij_address
30
31         add $t0, $t0, $s0
32         lw $t1, 0($t0)
33         li $t2, -1
34         beq $t2, $t1, do_cb // while (board[row][column] == -1);
35
36         sw $t2, 0($t0)      // board[row][column] = -1; // -1 means bomb
present
37         addi $s1, $s1, 1
38         j begin_for_i_pb
39     end_for_i_pb:
40     restore_context
41     jr $ra

```

2.4 printBoard.asm

```

1  .include "macros.asm"
2  .globl printBoard
3
4  .data
5      new_line: .asciiz "\n"
6      single_space: .asciiz " "
7      four_spaces: .asciiz "    "
8      underline: .asciiz "___"
9      bar: .asciiz "| "
10     star: .asciiz " * "
11     hashtag: .asciiz " # "
12
13  .text
14  printBoard:
15      save_context
16      move $s0, $a0          #&board
17      move $s1, $a1          #showBombs
18
19      #printf("    ")
20      la $a0, four_spaces
21      li $v0, 4
22      syscall
23
24      li $s2, 0              #j = 0
25      first_loop:
26          li $t0, SIZE
27          bge $s2, $t0, end_first_loop
28
29          # printf(" %d ", j)
30          la $a0, single_space
31          li $v0, 4
32          syscall
33          move $a0, $s2
34          li $v0, 1
35          syscall
36          la $a0, single_space
37          li $v0, 4
38          syscall
39

```

```

40         addi $s2, $s2, 1
41         j first_loop
42     end_first_loop:
43
44     # printf("\n")
45     la $a0, new_line
46     li $v0, 4
47     syscall
48
49     #printf(" ")
50     la $a0, four_spaces
51     li $v0, 4
52     syscall
53
54     li $s2, 0                #j = 0
55     second_loop:
56         li $t0, SIZE
57         bge $s2, $t0, end_second_loop
58
59         # printf("___")
60         la $a0, underline
61         li $v0, 4
62         syscall
63
64         addi $s2, $s2, 1
65         j second_loop
66     end_second_loop:
67
68     # printf("\n")
69     la $a0, new_line
70     li $v0, 4
71     syscall
72
73     li $s2, 0                #i = 0
74     pb_i_loop:
75         li $t0, SIZE
76         bge $s2, $t0, end_pb_i_loop
77
78         #printf("%d | ", i)
79         move $a0, $s2
80         li $v0, 1
81         syscall
82         la $a0, bar
83         li $v0, 4
84         syscall
85
86         li $s3, 0            #j = 0
87         pb_j_loop:
88             li $t0, SIZE
89             bge $s3, $t0, end_pb_j_loop
90
91             move $a1, $s2
92             move $a2, $s3
93             get_ij_address
94             add $t0, $t0, $s0
95             lw $s4, 0($t0)    # s4 := board[i][j]
96
97             li $t1, -1
98             seq $t0, $t1, $s4
99             seq $t1, $s1, 1

```

```

100         and $t0, $t0, $t1          #board[i][j] == -1 && showBombs == 1
101         li $t1, 1
102         beq $t0, $t1, if
103
104         bge $s4, $0, else_if        #board[i][j] >= 0
105
106         j else
107
108         if:
109             la $a0, star
110             li $v0, 4
111             syscall
112             j end_if
113         else_if:
114             la $a0, single_space
115             li $v0, 4
116             syscall
117
118             move $a0, $s4
119             li $v0, 1
120             syscall
121
122             la $a0, single_space
123             li $v0, 4
124             syscall
125             j end_if
126         else:
127             la $a0, hashtag
128             li $v0, 4
129             syscall
130         end_if:
131
132         addi $s3, $s3, 1
133         j pb_j_loop
134     end_pb_j_loop:
135
136     addi $s2, $s2, 1
137     # printf("\n")
138     la $a0, new_line
139     li $v0, 4
140     syscall
141     j pb_i_loop
142 end_pb_i_loop:
143
144 restore_context
145 jr $ra

```

O princípio dessa função é imprimir no console o tabuleiro dependendo da situação do jogo.

Na seção `.data` iniciamos as mensagens que precisaremos utilizar durante a função, como `new_line` e `space`.

O primeiro parâmetro (`a0`) passado para a função é o endereço de memória do tabuleiro e o segundo (`a1`) é uma variável 'booleana' que diz se é preciso mostrar as bombas ou deixá-las escondidas.

É verificado se a célula é uma bomba (-1), se ainda não foi revelado (-2) ou se já foi revelado (≥ 0). É então mostrado no console o símbolo correto.

2.5 play.asm

```
1 .include "macros.asm"
2
3 .globl play
4
5 play:
6     save_context
7
8     move $s0, $a0
9     move $s1, $a1
10    move $s2, $a2
11
12    get_ij_address
13    add $t0, $t0, $s0
14    lw $t0, 0($t0)           # t0 := board[i][j]
15
16    li $v0, 0
17    beq $t0, -1, end_zero
18
19    bne $t0, -2, end_one
20
21    move $s3, $ra
22    jal countAdjacentBombs
23    move $ra, $s3
24
25    move $a1, $s1
26    move $a2, $s2
27    get_ij_address
28    add $t0, $t0, $s0
29    sw $v0, 0($t0)
30
31    bne $v0, $0, end_if_play
32        move $a0, $s0
33        move $a1, $s1
34        move $a2, $s2
35        jal revealAdjacentCells
36    end_if_play:
37
38    end_one:
39    li $v0, 1
40
41    end_zero:
42    restore_context
43    jr $ra
```

Primeiramente pegamos o valor armazenado em *board*[*i*][*j*], se a célula for uma bomba carregamos o valor -1 em *v0* e pulamos para o final do código. Caso o valor não seja -2 (célula já revelada) pulamos para o final e retornamos 1, caso seja -2 , calculamos a quantidade de bombas adjacente a célula, esse valor é armazenado na célula e então chamamos a função para revelar as células adjacentes a célula atual.

2.6 countAdjacent.asm

```
1 .include "macros.asm"
2 .globl countAdjacentBombs
3
4 countAdjacentBombs:
5     save_context
6
7     move $s0, $a0          # &board
8     move $s1, $a1          # row
9     move $s2, $a2          # col
10
11     li $s3, 0              # count = 0
12
13     li $s4, -1
14     add $s4, $s4, $s1
15     cb_i_loop:
16         move $t0, $s1
17         addi $t0, $t0, 1
18         bgt $s4, $t0, end_cb_i_loop
19
20         li $s5, -1
21         add $s5, $s5, $s2
22         cb_j_loop:
23             move $t0, $s2
24             addi $t0, $t0, 1
25             bgt $s5, $t0, end_cb_j_loop
26
27             sge $t0, $s4, $0          # i >= 0
28             li $t1, SIZE
29             slt $t1, $s4, $t1        # i < SIZE
30             sge $t2, $s5, $0          # j >= 0
31             li $t3, SIZE
32             slt $t3, $s5, $t3        # j < SIZE
33
34             and $t0, $t0, $t1
35             and $t2, $t2, $t3
36             and $s6, $t0, $t2
37
38             move $a1, $s4
39             move $a2, $s5
40             get_ij_address
41             add $t0, $t0, $s0
42             lw $t0, 0($t0)           # t0 := board[i][j]
43
44             seq $t0, $t0, -1
45             and $t0, $t0, $s6
46
47             bne $t0, 1, end_if_cb     # i>=0 && i<SIZE && j>=0 && j<SIZE &&
board[i][j] == -1
48             addi $s3, $s3, 1         #count ++
49             end_if_cb:
50             addi $s5, $s5, 1
51             j cb_j_loop
52         end_cb_j_loop:
53
54         addi $s4, $s4, 1
55         j cb_i_loop
56     end_cb_i_loop:
57
```

```

58     move $v0, $s3
59
60     restore_context
61     jr $ra

```

Iniciamos o registrador *s3* como 0 para ser o nosso contador de bombas e começamos um loop aninhado para percorrer as linhas e colunas vizinhas à célula atual ($[row - 1, row + 1] \times [col - 1, col + 1]$).

Verificamos se a vizinhança está dentro dos limites do tabuleiro. Caso esteja, verificamos se é uma bomba. Se for aumentamos o contador em 1.

Ao final retornamos em *v0* o valor do contador.

2.7 revealAdjacentCells.asm

```

1  .include "macros.asm"
2  .globl revealAdjacentCells
3
4  revealAdjacentCells:
5
6      addi $sp, $sp, -48
7      sw $s0, 0($sp)
8      sw $s1, 4($sp)
9      sw $s2, 8($sp)
10     sw $s3, 12($sp)
11     sw $s4, 16($sp)
12     sw $s5, 20($sp)
13     sw $s6, 24($sp)
14     sw $s7, 28($sp)
15     sw $ra, 32($sp)
16     sw $a0, 36($sp)
17     sw $a1, 40($sp)
18     sw $a2, 44($sp)
19
20     move $s0, $a0
21     move $s1, $a1
22     move $s2, $a2
23
24     jal countAdjacentBombs
25
26     move $a0, $s0
27     move $a1, $s1
28     move $a2, $s2
29     get_ij_address
30     add $t0, $t0, $s0
31     sw $v0, 0($t0)
32
33     bnez $v0, end_rv_i_loop
34
35     addi $s3, $s1, -1                # s3 := row - 1
36     rv_i_loop:
37         addi $t0, $s1, 1              # t0 := row + 1
38         bgt $s3, $t0, end_rv_i_loop
39
40         addi $s4, $s2, -1            # s4 := col - 1
41         rv_j_loop:
42             addi $t0, $s2, 1          # t0 := col + 1
43             bgt $s4, $t0, end_rv_j_loop
44

```

```

45         li $t1, SIZE
46         bltz $s3, end_if           # i < 0
47         bge $s3, $t1, end_if      # i >= SIZE
48         bltz $s4, end_if           # j < 0
49         bge $s4, $t1, end_if      # j >= SIZE
50
51         move $a1, $s3
52         move $a2, $s4
53         get_ij_address
54         add $t0, $t0, $s0
55         lw $t0, 0($t0)
56         bne $t0, -2, end_if        # board[i][j] != -2
57
58         #Entrou no caso recursivo
59         jal recursive_case
60
61         end_if:
62
63         addi $s4, $s4, 1
64         j rv_j_loop
65     end_rv_j_loop:
66
67         addi $s3, $s3, 1
68         j rv_i_loop
69     end_rv_i_loop:
70
71     #se chegou aqui n o entrou na recurs o , ou saiu do loop
72     end_reveal:
73     lw $s0, 0 ($sp)
74     lw $s1, 4 ($sp)
75     lw $s2, 8 ($sp)
76     lw $s3, 12 ($sp)
77     lw $s4, 16 ($sp)
78     lw $s5, 20 ($sp)
79     lw $s6, 24 ($sp)
80     lw $s7, 28 ($sp)
81     lw $ra, 32($sp)
82     lw $a0, 36($sp)
83     lw $a1, 40($sp)
84     lw $a2, 44($sp)
85     addi $sp, $sp, 48
86
87     jr $ra
88
89
90 recursive_case:
91     move $a1, $s3
92     move $a2, $s4
93     jal revealAdjacentCells
94
95     lw $s0, 0 ($sp)
96     lw $s1, 4 ($sp)
97     lw $s2, 8 ($sp)
98     lw $s3, 12 ($sp)
99     lw $s4, 16 ($sp)
100    lw $s5, 20 ($sp)
101    lw $s6, 24 ($sp)
102    lw $s7, 28 ($sp)
103    lw $ra, 32($sp)
104    lw $a0, 36($sp)

```

```

105     lw $a1, 40($sp)
106     lw $a2, 44($sp)
107     addi $sp, $sp, 48
108
109     jr $ra
110
111

```

Inicialmente contamos a quantidade de bombas adjacentes a célula e revelamos no console. Caso não seja zero nós retornamos da função. Caso contrário entramos em um loop aninhado para verificar as células adjacentes.

Da mesma forma como a função anterior verificamos se as células adjacentes estão nos limites do tabuleiro. Caso esteja verificamos se a mesma já foi revelada, se não nós entramos recursivamente na função, com os parâmetros atualizados para a linha e a coluna adjacente.

2.8 checkVictory.asm

```

1  .include "macros.asm"
2  .globl checkVictory
3
4  checkVictory:
5      save_context
6
7      move $s0, $a0          #s0 := &board
8
9      li $s4, 0              # count = 0
10
11     li $s1, 0              # i = 0
12     begin_cv_i_loop:
13         li $t0, SIZE
14         bge $s1, $t0, end_cv_i_loop
15
16         li $s2, 0          #j = 0
17         begin_cv_j_loop:
18             li $t0, SIZE
19             bge $s2, $t0, end_cv_j_loop
20
21             move $a1, $s1
22             move $a2, $s2
23             get_ij_address
24             add $t0, $t0, $s0
25             lw $t0, 0($t0)
26
27             blt $t0, 0, end_if_cv
28             addi $s4, $s4, 1
29             end_if_cv:
30
31             addi $s2, $s2, 1
32             j begin_cv_j_loop
33         end_cv_j_loop:
34
35         addi $s1, $s1, 1
36         j begin_cv_i_loop
37     end_cv_i_loop:
38
39     li $t0, SIZE
40     mul $t0, $t0, $t0
41     li $t1, BOMB_COUNT

```

```

42     sub $t0, $t0, $t1
43
44     li $v0, 0
45     blt $s4, $t0, end_check
46         li $v0, 1
47     end_check:
48
49     restore_context
50     jr $ra

```

Para essa função iniciamos um contador em zero e entramos em um loop aninhado por todas as células, caso a célula já esteja revelada nós aumentamos o contador em 1.

Ao final comparamos a quantidade de casas reveladas com a quantidade de casas disponíveis para escolha ($n = \text{SIZE} * \text{SIZE} - \text{BOMB_COUNT}$), caso a quantidade de casas seja maior a n significa que o jogador venceu.

2.9 main.asm

```

1  .include "macros.asm"
2
3  .data
4      msg_row:          .asciiz "Enter the row for the move: "
5      msg_column:       .asciiz "Enter the column for the move: "
6      msg_win:          .asciiz "Congratulations! You won!\n"
7      msg_lose:         .asciiz "Oh no! You hit a bomb! Game over.\n"
8      msg_invalid:      .asciiz "Invalid move. Please try again.\n"
9
10 .globl main
11 .text
12
13 main:
14     li $t0, SIZE
15     mul $t0, $t0, $t0
16     mul $t0, $t0, -4
17     add $sp, $sp, $t0    # board;
18     li $s1, 1           # int gameActive = 1;
19     move $s0, $sp
20     move $a0, $s0
21
22     jal initializeBoard # initializeBoard(board);
23     move $a0, $s0
24     jal plantBombs      # placeBombs(board);
25
26     begin_while:        # while (gameActive) {
27     beqz $s1, end_while
28     move $a0, $s0
29     li $a1, 0
30     jal printTable      # printBoard(board,0); // Shows the board without
bombs
31
32     la $a0, msg_row
33     li $v0, 4           # printf("Enter the row for the move:
");
34     syscall
35
36     li $v0, 5           # scanf("%d", &row);
37     syscall
38     move $s2, $v0
39

```

```

40     la $a0, msg_column
41     li $v0, 4                # printf("Enter the column for the move: ")
42     ;
43     syscall
44     li $v0, 5                # scanf("%d", &column);
45     syscall
46     move $s3, $v0
47
48     li $t0, SIZE
49     blt $s2, $zero, else_invalid    #if (row < 0 || row >= SIZE || column < 0
|| column >= SIZE) {
50     bge $s2, $t0, else_invalid
51     blt $s3, $zero, else_invalid
52     bge $s3, $t0, else_invalid
53
54     move $a0, $s0
55     move $a1, $s2
56     move $a2, $s3
57     jal play
58
59     bne $v0, $zero, else_if_main    # if (!play(board, row, column)) {
60     li $s1, 0                        # gameActive = 0;
61     la $a0, msg_lose                # printf("Oh no! You hit a bomb
! Game over.\n");
62     li $v0, 4
63     syscall
64     j end_if_main
65
66     else_if_main:
67     move $a0, $s0
68     jal checkVictory                # else if (checkVictory(board))
{
69     beq $v0, $zero, end_if_main
70     la $a0, msg_win                # printf("Congratulations! You
won!\n");
71     li $v0, 4
72     syscall
73     li $s1, 0                        # gameActive = 0; //
Game ends
74     j end_if_main
75     else_invalid:
76     la $a0, msg_invalid            # printf("Invalid move. Please try
again.\n");
77     li $v0, 4
78     syscall
79     end_if_main:
80     j begin_while
81     end_while:
82     move $a0, $s0
83     li $a1, 1
84     jal printBoard                # printBoard(board,1);
85     li $v0, 10
86     syscall

```

A função main é a principal do jogo, é ela que controla todo o fluxo de funcionamento.

Inicialmente carregamos as mensagens que serão exibidas e colocamos um loop para verificar se o jogo terminou ou não. enquanto o jogo não terminar pedimos ao jogador para digitar a linha e a coluna em em deseja jogar, revelamos a casa.

Após revelar verificamos se o jogador perdeu ou se o jogo continuará.
Fazemos isso até o jogador perder ou ganhar.