Arthur Ricardo Macêdo Pereira

Everton Lohan Pereira Ferreira

Arquiterura e Organização de Computadore - Ciência da Computação (UFCA)

Relatório MineSweeper

- Primeiramente, após recebermos o projeto nós dividimos para a dupla, Arthur ficou com as funções *PLAY e COUNTADJACENTBOMBS,* e Everton com as funções *REVEALNEIGHTBORINGCELLS* e *CHECKVICTORY.* Após as funções divididas começamos a fazer o projeto, inicialmente analisando os códigos das outras funções fornecidos pelo professor.

A função PLAY:

```
save_context
move $s0, $a0 #endereço do board
move $s1, $a1 #em a1 está a linha
move $s2, $a2 #en a2 está a coluna
```

Primeiramente, usamos a função save_context para podermos usar os registradores sem ocorrer problemas. Em seguida usamos a instrução **move** para mover os valores dos registradores a0, a1 e a2 para os registradores s0, s1 e s2, foi usados os registradores A, pois são usados para passar parâmetros para as funções, e usamos os registradores S dentro da função, pois é de melhor uso com o save_context.

Após isso, temos que usar a instrução **sll** para andarmos no vetor board (tabuleiro), para isso devemos saber que ela “anda para esquerda” sempre em casas de potência de 2, outras informações que temos que saber é que cada espaço do vetor tem 4 bits e que as linhas são em 8 e para andarmos no vetor board usaremos principalmente elas, e as colunas usaremos somente a quantidade de espaço ocupado. Assim, pegamos o valor digitado na linha e multiplicamos por 2 elevado a 5 e o valor da coluna por 2 elevado a 2 e guardamos nos registradores t1 e t2 respectivamente. Depois, usamos a instrução **add** para somarmos o valor de t1 e t2 e guardamos em t0, assim temos a coordenada exata da posição do vetor, e guardamos em t4 essa coordenada, pois usaremos o t0 ainda. Todo esse código eu vou chamar de **“Esquema de andar no vetor”** nos próximos códigos.

```
sll $t1, $s1, 5 #Multiplica o que ta em s1(a linha desejada) por 2 elevado a 5, pois é 8 colunas e cada uma usa 4bit logo 32
sll $t2, $s2, 2 #Multiplica o que ta em s2(a coluna desejada) mesmo esquema

add $t0, $t1, $t2 #soma a quantidade de "casas no vetor", ou seja, soma os endereços da linha e das colunas e armazena em t0
add $t0, $t0, $s0 #soma o endereço do board com os endereços das linhas e colunas
lw $t4, 0($t0)
```

Seguindo no código, usamos as instruções **beq e bne,** a primeira para sabermos se o valor encontrado em t4 é **igual** a -1, ou seja, se for igual ele encontrou uma bomba, logo pula para a função *retorna 0* (será explicada mais à frente), a segunda para sabermos se o valor encontrado em t4 é **diferente** de -2, ou seja, se ele for diferente de -2 a célula já foi revelada, pulando assim para a função *retorna 1* (isso quer dizer que todas as células já foram reveladas e o jogador ganhou).

```
beq $t4, -1, retorne0 #se t4 for igual a -1 pula para retorne 0 (encontrou uma bomba)
bne $t4, -2, retorne1 #se t4 for diferente de -2 pula para retorne 1
```

Após isso, temos um código para chamarmos a função countadjacentbombs com a instrução **jal**

```
addi $sp, $sp, -4 #aloca espaço na pilha
sw $s0, 0($sp) #guarda em s0 o endereço do ponteiro sp
move $a3, $t0 #move o endereço do board para a3
jal countAdjacentBombs #chama a função
addi $sp, $sp, 4 #libera o espaço alocado
sw $v0, 0($a3) #guarda em a3 valor retornado da função
```

Seguindo usamos a instrução **bne** para compararmos o valor de v0, retornado da função countadjacentbombs, comparamos esse vetor com 0, assim se o valor de v0 for diferente de 0 ele pula para retorne 1, se for igual a 0 chamamos a função revelneightboringcells para revelar as células adjacentes.

```
bne $v0, $zero, retorne1 #se a celula revelada nao tiver o valor zero, pula pra retorne1
addi $sp, $sp, -4 #aloca espaço na pilha
sw $s0, 0 ($sp) #guarda em s0 o endereço do ponteiro sp
jal revealNeighboringCells #chama a função
addi $sp, $sp, 4 #libera o espaço alocado
```

Por último temos as funções *retorne 1 e retorne 0*, que basicamente o que muda é o valor que elas guardam em v0 que é o retorno da função PLAY.

```
retorne1:
li $v0, 1
restore_context
jr $ra
retorne0:
li $v0, 0
restore_context
jr $ra
```

A função CountAdjacentBombs:

Primeiramente, usamos a função save_context e passamos os valores dos registradores A para os registradores S.

```
save_context
move $s0, $a0 #endereço do board
move $s1, $a1 #em a1 está a linha
move $s2, $a2 #en a2 está a coluna
```

Depois declaramos a variável cont, com a instrução **lw** e guardando zero em v0, assim colocamos cont = 0.

```
li $v0, 0 #cont = 0
```

Depois disso vamos iniciar os laços de repetição *(for)*, para isso vamos iniciar o **i**, para isso devemos somar -1 no registrador que está guardado as linhas, assim ficando *row - 1*. E depois, declaramos outra variável para a condição do for, com isso adicionamos 1 nas linhas, ficando *row + 1,* tudo isso usando a instrução **addi**.

```
addi $s3, $s1, -1 #pegando o valor das linhas e diminuindo 1 e colocando em i, row - 1 = i
addi $s4, $s1, 1 #somando um ao valor das linhas, row + 1
```

Vamos agora criar o for mesmo, para isso vamos usar a lógica inversa, ou seja, quando o valor de i (s3) for maior que *row + 1* (s4) ele sai do for, pulando para a função *finaldofordoi* , se ele não entrar nessa condição quer dizer que o for inicia.

```
fordoi:
       bgt $s3, $s4, finaldofordoi #(pule se for maior) pula para o finaldofordoi se o valor de s3(i) for maior que o valor de s4
```

Dentro do for do i, temos o for do j, que é a mesma coisa do for do i.

```
addi $s5, $s2, -1 #pegando o valor das colunas e diminuindo 1 e colocando em j, column - 1 = j
addi $s6, $s2, 1 #somando um ao valor das linhas, column + 1
fordoj:
       bgt $s5, $s6, finaldofordoj #pula para o finaldofordoj se o valor de s5(j) for maior que o valor de s6
```

E dentro do for do j, temos um if com algumas condições:
vamos usar sempre a lógica inversa;
i >= 0: usando a instrução **blt** compara o valor de s3 com zero, se for menor não entra no if;
i < SIZE: usando a instrução **bge** compara o valor de s3 com SIZE, se for maior não entra;
j >= 0: mesma coisa do i;
j < SIZE: mesma coisa do i também;
board[i][j] == -1: temos que fazer aquele "**Esquema de andar no vetor**" para encontrar a coordenada e comparar ela com -1, se for igual entra, se não a condição não é satisfeita.

```
blt $s3, $zero, continua #se o numero de linhas for menor que zero, sai do for, no caso vai para o continua
bge $s3, SIZE, continua #se o numero de linhas for maior que o tamanha, sai do for
blt $s5, $zero, continua #se o numero de colunas for menor que zero, sai do for, no caso vai para o continua
bge $s5, SIZE, continua #mesma coisa das linhas

sll $t1, $s3, 5 #esquema para andar no vetor(board) 2^5=32, que é a mesma coisa de 8*4
sll $t2, $s5, 2 #ele "anda para esquerda" na verdade pega o valor que ta em s5 multiplica por 2^2 e guarda em t2

add $t0, $t1, $t2 #soma os valores da linhas e colunas e guarda em t0, pra andar no board
add $t0, $t0, $s0 #soma as cordenadas e localiza no board
lw $s7, 0($t0) #guarda em s7 o valor de t0

bne $s7, -1, continua #se o valor que tiver em s7 não for igual a -1(uma bomba) ele sai do for
```

Se todas essas condições não forem satisfeitas, usando a lógica inversa, o código em c é satisfeito, assim usamos a instrução **addi,** para adicionar +1 no cont.

```
addi $v0, $v0, 1 #cont++
```

No final temos as funções *continua, finaldofordoj e finaldofordoi*

```
continua:
        addi $s5, $s5, 1 #j++
        j fordoj #pula para o fordoj, ou seja, continua o for

finaldofordoj:
        addi $s3, $s3, 1 #i++
        j fordoi #mesmo esquema

finaldofordoi:
        restore_context
        jr $ra
```

Na função continua ela incrementa o j em 1 e pula para o for do j, continuando o for;
Na função finaldofordoj ela incrementa o i em 1 e pula para o for o i;
Na função finaldofordoi ela usa a função restore_context e sai da função countadjacentbombs.

Função RevealNeighboringCells:

- Função que fica responsável em revelar as próximas células:

```
void revealAdjacentCells(int board[][SIZE], int row, int column) {
    // Reveals the adjacent cells of an empty cell
    for (int i = row - 1; i <= row + 1; ++i) {
        for (int j = column - 1; j <= column + 1; ++j) {
            if (i >= 0 && i < SIZE && j >= 0 && j < SIZE && board[i][j] == -2) {
                int x = countAdjacentBombs(board, i, j); // Marks as revealed
                board[i][j] = x;
                if (!x)
                    revealAdjacentCells(board, i, j); // Continues the revelation recursively
            }
        }
    }
}
```

para traduzir essa função em C , para Assembly usamos:

```
save_context
move $s0, $a0 #endereço do board
move $s1, $a1 #em a1 está a linha
move $s2, $a2 #en a2 está a coluna
```

Para iniciar utilizamos novamente o **save_context**: para salvar o contexto atual na pilha preservando o estado dos registradores. Depois utilizamos o **move $s0, $a0, move $s1, $a1, move $s2, $a2:** Para Mover os argumentos da função (board, row e column) para registradores para os facilitar a manipulação.

```
li $v0, 0
```

Em seguida Utilizamos a instrução **LI (Load immediate)** para guardar o 0 no registrador **$v0.** Que no caso ele Inicializa o contador $v0 como zero e foi usado para marcar as células reveladas.

- `for (int i = row - 1; i <= row + 1; ++i) {` essa linha de código em C foi “traduzida” utilizando a instrução **bgt**(branch on greater than ou pule se for maior) e **addi**(add immediate).

```
addi $s3, $s1, -1
addi $s4, $s1, 1
```

Utiliza a instrução **addi (add immediate)** ele pega o valor das linhas e diminui 1 e colocando em i, **row - 1 = i** e soma um ao valor das linhas,**row + 1** respectivamente. Essas coordenadas são necessárias para garantir que todas as células adjacentes à célula atual sejam percorridas corretamente durante o processo de contagem de bombas adjacentes.

```
fordoi:
        bgt $s3, $s4,finalfordoi
```

“fordoi:”(pra iniciar o loop) Então ela pula para o **“finaldofordoi”** se o valor de $s3(i) for maior que o valor de $s4 Essa instrução verifica se já percorremos todas as linhas adjacentes. Se sim, o loop termina.

- `for (int j = column - 1; j <= column + 1; ++j) {` O for do j, que está dentro do for do i, foi traduzido seguindo os mesmos parâmetros do for do i. Portanto,

```
addi $s5,$s2,-1
addi $s6,$s2,1
```

Utiliza a mesma instrução **addi** (add immediate) ele pega o valor das colunas e diminui 1 e colocando em J, **column- 1 = j** e soma um ao valor das linhas, **column+ 1** respectivamente.

- ```
  if (i >= 0 && i < SIZE && j >= 0 && j < SIZE && board[i][j] == -2) {
      int x = countAdjacentBombs(board, i, j); // Marks as revealed
      board[i][j] = x;
      if (!x)
          revealAdjacentCells(board, i, j); // Continues the revelation recursively
  }
  ```

Para traduzir esse trecho do código onde é implementado o if:

```
if (i >= 0 && i < SIZE && j >= 0 && j < SIZE && board[i][j] == -2) {
```

- ★ **blt(branch on less than ou pule se for menor)**
- ★ **bge(branch on greater than or equal ou pule se for maior ou igual)**
- ★ **sll (shift left logical ou “ande” para esqueda)**
- ★ **add**
- ★ **lw(load word)**
- ★ **bne(branch on not equal ou pule se não for igual)**
- ★ **addi (add immediate)**

```
blt $s3,$zero,continua
bge $s3, SIZE,continua
blt $s5,$zero,continua
bge $s5,SIZE,continua
```

Estas instruções verificam se os índices de linha e coluna estão dentro dos limites válidos do board. Portanto,

1. Se o número de linhas($s3) for menor que zero, sai do for, no caso vai para o “continua”.
2. Se o número de linhas($s3) for maior que o tamanho, sai do for.
3. Se o número de colunas($s5) for menor que zero, sai do for, no caso vai para o continua.
```

4. Se o número de colunas ($s5) for maior que o tamanho, sai do for.

```
sll $t1,$s3,5
sll $t2,$s5,2
```

Estas instruções calculam os deslocamentos necessários para acessar a célula adjacente no board.
Então, 1° esquema para andar no vetor(board) 2^5=32, que é a mesma coisa de 8*4. 2° ele "anda para esquerda" que na verdade pega o valor que ta em $s5 multiplica por 2^2 e guarda em $t2.

```
add $t0,$t1,$t2
add $t0,$t0,$s0
lw $s7,0($t0)
```

1. soma os valores da linhas e colunas e guarda em t0, pra andar no board.
2.soma as cordenadas e localiza no board.
3. usa a instrução Lw(load word) pra guardar em $s7 o valor de $t0.

Ou seja, Calculam o endereço da célula adjacente na matriz board.

```
bne $s7, -2, continua
```

Utilizamos a instrução **bne**(**branch on not equal ou pule se não for igual)** para verificar se a célula adjacente já foi revelada. Ou seja, se o valor que tiver em $s7 não for igual a -2 ele sai do for.

```
int x = countAdjacentBombs(board, i, j); // Marks as revealed
board[i][j] = x;
```

Para traduzir esse trecho do código usamos:

```
#chamando a função countadjacentbombs
move $a1,$s3
move $a2,$s5
addi $sp, $sp,-4
sw $s0,0($sp)
move $a3,$t0
jal countAdjacentBombs
addi $sp, $sp,4
sw $v0,0($a3)
```

- **move $a1, $s3 e move $a2, $s5**:Move as coordenadas da célula adjacente para os argumentos da função countAdjacentBombs.
- **addi $sp, $sp, -4**

Esta instrução reserva espaço na pilha para armazenar temporariamente o valor de $s0 e preservar seu valor. O valor -4 é subtraído do ponteiro da pilha para criar espaço para armazenar uma palavra.

- **sw $s0, 0($sp)**: Utilizamos a instrução **sw (store word)** para armazenar o valor de $s0, que contém o endereço da célula adjacente na matriz board, na pilha. Isso é feito para preservar o valor de $s0 enquanto chamamos a função countAdjacentBombs

- **move $a3, $t0**: Esta instrução move o endereço da célula adjacente na matriz board, que está armazenado em $t0, para o registrador de argumento

$a3. Este argumento é necessário para passar o endereço da célula adjacente para a função countAdjacentBombs.

- **jal countAdjacentBombs**: Usamos essa instrução **jal (jump and link)** que no caso essa instrução faz uma chamada para a função countAdjacentBombs, que contará o número de bombas adjacentes à célula atual.

- **addi $sp, $sp, 4:** Esta instrução restaura o ponteiro da pilha, removendo o espaço reservado anteriormente para armazenar $s0. Isso libera o espaço na pilha usado temporariamente para preservar o valor de $s0.

- **sw $v0, 0($a3)**: Após a chamada da função countAdjacentBombs, o resultado da contagem (o número de bombas adjacentes) é armazenado na célula adjacente do board. Isso é feito escrevendo o valor retornado pela função ($v0) na posição de memória apontada por $a3, que contém o endereço da célula adjacente.

- Resumindo esse trecho chama a função countadjacentbombs e também gerencia o armazenamento temporário de valores necessários para a chamada da função na pilha, garantindo que o estado dos registradores seja preservado durante a execução.

```
move $a1, $s3
move $a2, $s5
```

move $a1, $s3 e move $a2, $s5: Move as coordenadas da célula adjacente para os argumentos da função revealNeighboringCells. Ou seja, tão colocando $s3 (linha) em $a1 e $s5 em $a2 respectivamente. Que no caso são o $a1 e $a2 são usados para passar um argumento para a função revealNeighboringCells.

```
if (!x)
    revealAdjacentCells(board, i, j);
```

Esse trecho traduzimos da seguinte forma:

```
bne $v0,$zero,continua
addi $sp, $sp,-4
sw $s0,0($sp)
jal revealNeighboringCells
addi $sp, $sp,4
```

Este trecho de código é responsável por verificar se o resultado retornado pela função countAdjacentBombs é diferente de zero. Se o resultado for diferente de zero, significa que há bombas adjacentes à célula atual, e portanto não é necessário continuar a revelar as células adjacentes. Se o resultado for zero, significa que não há bombas adjacentes, então ele continua a revelar as células adjacentes recursivamente.Ou seja,

- **bne(branch on not equal ou pule se não for igual):** Se o valor retornado pela função countAdjacentBombs (armazenado em $v0) for diferente de zero. Se $v0 for diferente de zero, ele pula o “**continua**”.

- **addi $sp, $sp,-4:** Esta instrução reserva espaço na pilha para armazenar temporariamente o valor de $s0 antes de chamar a função revealNeighboringCells.

- **sw $s0, 0($sp)**: Esta instrução armazena o valor de $s0, que contém o endereço da célula atual na matriz board, na pilha. Isso é feito para preservar o valor de $s0 enquanto chamamos a função revealNeighboringCells.

- **jal revealNeighboringCells**: Esta instrução chama a função revealNeighboringCells, que irá revelar as células adjacentes à célula atual. Se a célula atual não tiver bombas adjacentes, esta função será chamada recursivamente para revelar as células adjacentes a ela.

- **addi $sp, $sp, 4**: Esta instrução restaura o ponteiro da pilha, removendo o espaço reservado anteriormente para armazenar $s0. Isso libera o espaço na pilha usado temporariamente para preservar o valor de $s0.

```
continua:
        addi $s5,$s5,1
        j fordoj

finalfordoj:
        addi $s3,$s3,1
        j fordoi

finalfordoi:
        restore_context
        jr $ra
```

Por fim , as funções para marcar o final dos loops internos e externos, respectivamente.

Na função “**continua**” ela incrementa o j em 1 e pula para o for do j, continuando o for;
Na função **finaldofordoj** ela incrementa o i em 1 e pula para o for o i;
Na função **finaldofordoi** ela usa a função restore_context e sai da função **revealNeighboringCells.**.

Função CheckVictory:

```
int checkVictory(int board[][SIZE]) {
    int count = 0;
    // Checks if the player has won
    for (int i = 0; i < SIZE; ++i) {
        for (int j = 0; j < SIZE; ++j) {
            if (board[i][j] >= 0) {
                count++;
            }
        }
    }
    if (count < SIZE * SIZE - BOMB_COUNT)
        return 0;
    return 1; // All valid cells have been revealed
}
```

A Função CheckVictory é responsável para checar se o jogador venceu ou perdeu

Para traduzir essa função em C , para Assembly usamos:

```
move $s0, $a0

li $v0, 0 # count = 0
```

1. **move $s0, $a0:** Esta instrução move o endereço do tabuleiro (passado como argumento) para o registrador $s0.
2.**li $v0, 0:** Inicializamos a variável count com zero que será usada para contar as células reveladas.

```
for (int i = 0; i < SIZE; ++i) {
```

```
li $s3, 0 # i = 0
li $s4, SIZE
```

1.li $s3, 0: Inicializamos o contador i com zero para percorrer as linhas do tabuleiro.
2.**li $s4, SIZE:** Carregamos o tamanho do tabuleiro na variável $s4.
Ou seja, estamos contando o número de células reveladas (ou seja, não contendo bombas) em cada linha do tabuleiro para verificar se todas as células válidas foram reveladas, indicando assim a vitória do jogador.

```
fordoi:
        bge $s3, $s4, finalfordoi

        li $s5, 0
        li $s6, SIZE
```

1. o **“fordoi”** inicia o loop para percorrer no jogo.
2.**bge $s3, $s4, finalfordoi**: Verificamos se i ($s3) é maior ou igual ao tamanho do tabuleiro para sair do loop externo. (**branch on greater than or equal** ou pule se for maior ou igual).

3.**li $s5, 0:** Esta instrução inicializa o contador j com zero, o que significa que ele vai percorrer as colunas do tabuleiro a partir da primeira coluna.

4.**li $s6, SIZE**: Aqui, o tamanho do tabuleiro é carregado na variável $s6. O motivo de carregar o tamanho do tabuleiro em $s6 é garantir que o quantidade que o loop do for vai percorrer sobre as colunas seja executado exatamente SIZE veze. Ao fazer isso, garantimos que percorremos todas as colunas do tabuleiro, desde a primeira até a última, sem ultrapassar os limites do tabuleiro.

```
for (int j = 0; j < SIZE; ++j)
```

```
fordoj:
        bge $s5, $s6,finalfordoj
```

1. o **“fordoj”** inicia o loop para percorrer as colunas no jogo.
2.**bge $s5, $s6, finalfordoj:** Verificamos se j é **maior ou igual** ao tamanho do tabuleiro para sair do loop interno e pular para o “finalfordoj”.

```
if (board[i][j] >= 0) {
    count++;
}
```

```
sll $t1, $s3, 5 # t1 = i*32
sll $t2, $s5, 2 #
```

1.**sll $t1, $s3, 5**: Utilizamos o **sll** (shift left logical) paara calcular o deslocamento para acessar a linha i no tabuleiro (32 bits por linha).

2.**sll $t2, $s5, 2:** Calculamos o deslocamento para acessar a coluna j no tabuleiro (4 bytes por coluna). Ou seja, j*4.

```
add $t0, $t1, $t2
add $t0, $t0, $s0
```

1.add $t0, $t1, $t2: Esta instrução adiciona os valores contidos em $t1 e $t2 e armazena o resultado em $t0. Isso é feito para calcular o índice da célula no tabuleiro, onde $t1 contém o deslocamento necessário para acessar a linha atual (i * 32) e $t2 contém o deslocamento para acessar a coluna atual (j * 4).

2.add $t0, $t0, $s0: Adicionamos o endereço base do tabuleiro ao endereço da célula.

```
lw $s7, 0($t0)

bge $s7, 0, continua

addi $v0, $v0, 1    # Incrementa count
```

1.**lw $s7, 0($t0)**: Carregamos o valor da célula (i, j) no registrador $s7.

2.**bge $s7, 0, continua**: Verificamos se o valor da célula é maior ou igual a zero (ou seja, se a célula está revelada).

3.**addi $v0, $v0, 1:** Incrementamos o contador count (representando uma célula revelada).

```
continua:
        addi $s5, $s5, 1    # Incrementa j
        j fordoj
```

1.**continua**:: indica que o loop interno continuará.

2.**addi $s5, $s5, 1**: Incrementamos o contador j para avançar para a próxima coluna.

3. **j fordoj:** Esta instrução faz um salto de volta para o início do loop interno.

```
finalfordoj:
        addi $s3, $s3, 1        # Incrementa i
        j fordoi
```

1.**finalfordoj**:: Marca o fim do loop interno.

2.**addi $s3, $s3, 1:** Incrementa i para avançar para a próxima linha.

3. **j fordoi:** Salta de volta para o início do loop externo.

```
    if (count < SIZE * SIZE - BOMB_COUNT)
        return 0;
    return 1; // All valid cells have been revealed
}
```

```
finalfordoi:

        beq $v0,BOMB_COUNT, retorne1
        li $v0,0
        jr $ra
```

1.**finalfordoi**: iMarca o fim do loop externo.

2.**beq $v0, BOMB_COUNT, retorne1**: Esta instrução verifica se o valor contido em $v0 é igual a BOMB_COUNT. Se for verdadeiro, significa que o número de células reveladas (representado por $v0) é igual ao número total de células menos o número de bombas (BOMB_COUNT). Nesse caso, o jogo está ganho, então pulamos para o "**retorne1**"

3.**li $v0, 0**: Se o número de células reveladas não for igual ao número total de células menos o número de bombas, então isso significa que o jogo ainda não foi ganho. Então, atribuímos zero ao registrador $v0, indicando que o jogo não está ganho.

4.**jr $ra**: Retornamos à função chamadora, que é responsável por decidir o que fazer com o valor retornado por checkVictory.

```
retorne1:
        li $v0, 1
        restore_context
        jr $ra
```

1.**retorne1**:: Marca o início do código a ser executado se o jogador ganhar.

2.**li $v0, 1:** Esta instrução carrega o valor 1 no registrador $v0. Isso indica que o jogo foi vencido, pois chegamos à condição de vitória na função checkVictory.

3.**restore_contex**t:Restaura o contexto antes de retornar.. Isso garante que os registradores tenham os mesmos valores que tinham antes da execução da função checkVictory.

4.**jr $ra**: retornamos à função chamadora, que aguardará o valor retornado por checkVictory para tomar decisões adicionais no programa, como exibir uma mensagem de vitória ou continuar o jogo.

Por fim, finalizamos todas as implementações das funções que fará com que o jogo MineSweeper funcione sem nenhum problema. Essas funções são essenciais para a construção e funcionamento do jogo. O entendimento da lógica usada no códico em C também foi crucial para a tradução correta para a linguagem ASSEMBLY.