PROJETO ASSEMBLY - MIPS (BATALHA NAVAL)[1]

José Natanael Santos Matos (estudante)[2]
Victor da luz lima (estudante)[3]
André Luis Meneses Silva (professor)[4]

Universidade Federal de Sergipe - Campus Professor Alberto Carvalho
DSI - Departamento de Sistemas de Informação

RESUMO

Este artigo tem como objetivo mostrar a implementação do tradicional jogo de batalha naval utilizando os conceitos de uma linguagem de programação de baixo nível (assembly - MIPS). Assim como, uma explicação sobre o processo de aprendizagem e implementação nos pontos que consideramos mais importantes e mais delicados para a compreensão e para a própria aplicação ao projeto.

Palavras-chave: Jogos;Assembly; Batalha naval

ABSTRACT

This article aims to show the implementation of the traditional naval battle game using the concepts of a low level programming language (assembly - MIPS). As well as, an explanation about the learning process and implementation in the points that we consider more important and more delicate for the understanding and for the application itself to the project.

Key-words: Games;Assembly; Naval battle

---

[1] *Este artigo foi apresentado como avaliação da disciplina Organização e Arquitetura de Computadores (2020.1)*

[2] *Bacharel em Sistemas de Informação (UFS). E-mail: nathanaelsantos15@gmail.com.*
[3] *Bacharel em Sistemas de Informação (UFS). E-mail: victorlima2017@academico.ufs.br*
[4] *Doutor em Engenharia Elétrica (USP), Mestrado em Ciências da Computação (UFPE) e Graduação em Ciência da Computação (UFS). E-mail: andreluis.ms@gmail.com*

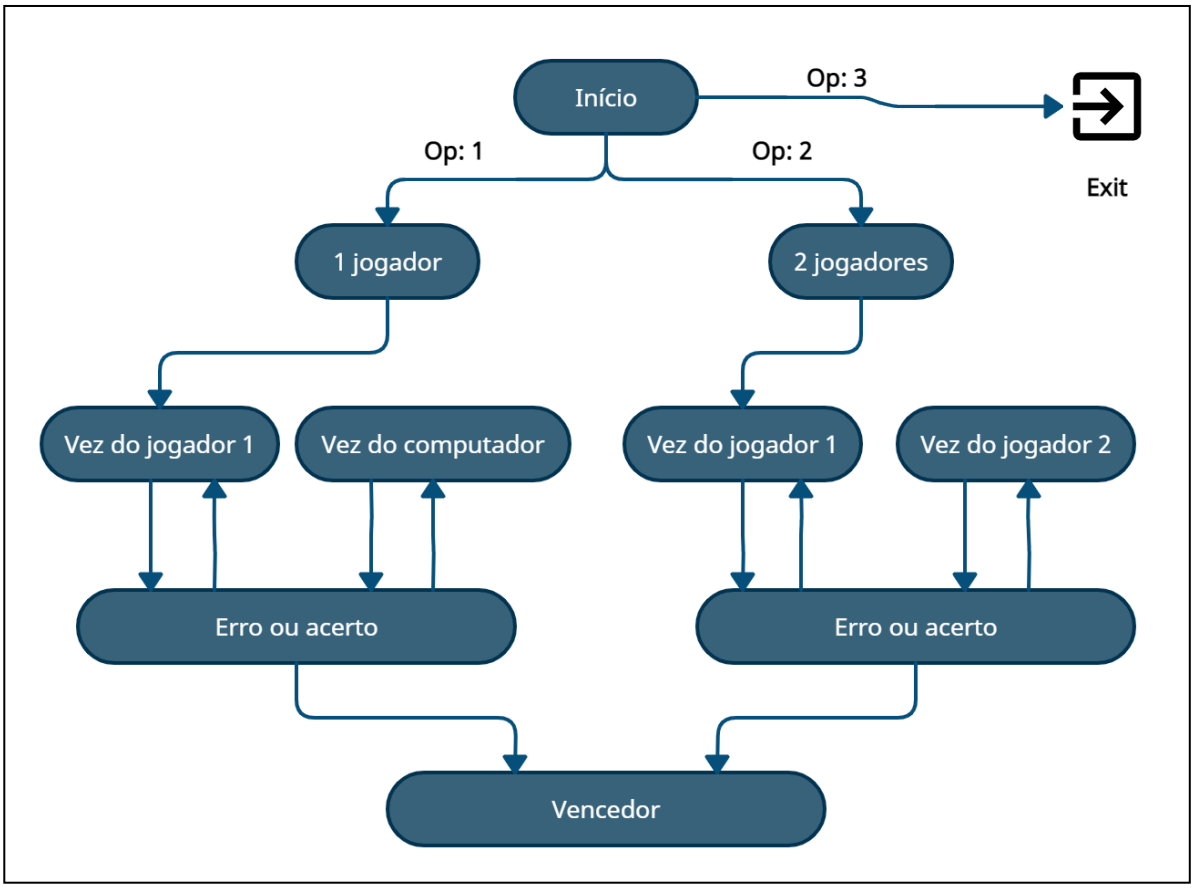
1. INTRODUÇÃO

Com o advento do avanço da tecnologia, houve grandes avanços tecnológicos, o surgimento de sistemas embarcados, IoT e desenvolvimento de firmware. Portanto, ter um conhecimento de montagem é essencial para tais áreas, bem como ter uma boa compreensão da linguagem de máquina e de como as coisas acontecem em baixo nível.

Este trabalho teve por objetivo desenvolver o tradicional jogo de batalha naval utilizando os conceitos de uma linguagem de programação de baixo nível (assembly - MIPS) fornece-nos uma base para a compreensão de alguns conceitos fundamentais, como cálculos, por exemplo. Assim como, uma explicação sobre o processo de aprendizagem e implementação nos pontos que consideramos mais importantes e mais delicados para a compreensão e para a própria aplicação ao projeto.

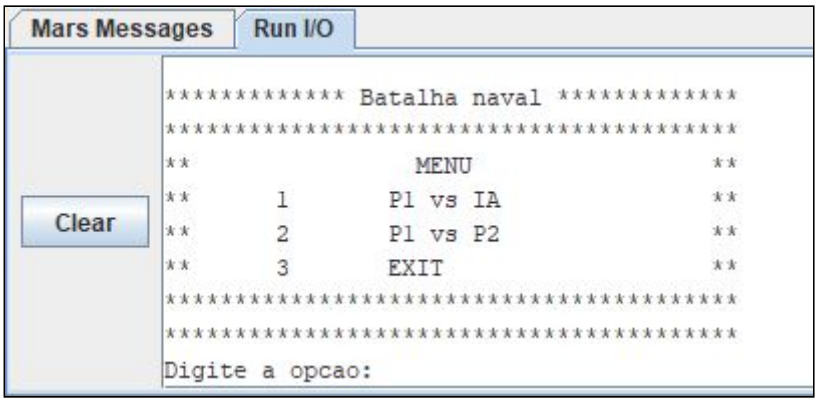
Para explicar como funciona a aplicação da melhor forma possível, é necessário compreender o fluxo do jogo e seus principais objetivos e como suas funcionalidades foram implementadas, tendo um panorama geral para assim entendermos como um todo. Abaixo está uma imagem que mostra este fluxo. Posteriormente, cada uma das funções será melhor explicada.

Fluxograma da aplicação



Op: *(Opção)*

De início, tem-se o menu, onde há três opções. A primeira, um jogador irá jogar contra uma “IA”. Na segunda opção, serão dois jogadores. Por fim, a terceira opção é responsável por encerrar o jogo. Ver a imagem do menu abaixo.



Fonte: MARS V4.5 (Simulated MIPS console input and output)

2. FUNÇÕES DO JOGO

Para a funcionalidade da aplicação foram escritas diversas funções, a maioria exclusiva para determinada funcionalidade. Iremos mostrar cada função de acordo com a que julgamos as mais relevantes e explicar o funcionamento de cada uma delas.

- **Menu principal**

```
menu_game:
    la $a0, txt_menu
    li $v0, 4
    syscall

    li $v0, 5   #Opcao ditigitada
    syscall

    move $k0, $v0

    beq $v0, 3, exit_game

    sge $a0, $v0, 4
    beq $a0, 1, msg_menu

    jal placar
    jal jogada_player1
```

```
        jr $ra
```

- **Gerador de linhas e colunas para a interface usando o Bitmap Display**

  a. **Linhas**

```
# ==============  LINHAS DO TABULEIRO  ==============
# ==================== linha 0 =======================
row_0x0:
    beq $t6,$s5,linha_s5x0x0
    beq $t6,$s6,linha_s6x0x0
    beq $t6,$s4,linha_s4x0x0
    beq $t6,$s7,linha_s7x0x0
        bne $t6,$s5,p_0x0
        p_0x0:bne $t6,$s6,p_0x0x0
        p_0x0x0: bne $t6,$s4,p_0x0x0x0
        p_0x0x0x0: bne $t6,$s7,quad_0x0
jr $ra
linha_s5x0x0:
       beq $s5,0,quad_0x0color
       bne $s5,0,quad_0x0
jr $ra
linha_s6x0x0:
       beq $s6,0,quad_0x0color
       bne $s6,0,quad_0x0
jr $ra
linha_s4x0x0:
       beq $s4,0,quad_0x0color
       bne $s4,0,quad_0x0
jr $ra
linha_s7x0x0:
       beq $s7,0,quad_0x0color
       bne $s7,0,quad_0x0
jr $ra
```

  a. **Colunas**

```
# ===================================================================
# ============================ culuna 0 =============================
```

```
    quad_0x0:
    lw $a2, 132($t2)
        bne $a2, $s2, possivel_jogada_0x0
        possivel_jogada_0x0:
            sw $s2, 132($t2)
            sw $s2, 136($t2)
            sw $s2, 260($t2)
            sw $s2, 264($t2)

            beq $a2, $s2, jogada_mesma_coordenada
    jr $ra
    acertou_0x0:
        lw $a2, 132($t2)
         beq $a2, $s1, poss_jogada_0x0 #Se a posicao estiver em azul, e
possivel jogar
            poss_jogada_0x0:
                addi $fp, $zero,1
                addi $s2, $zero, 0x17FD04  #Amarela
                sw $s2, 132($t2)
                sw $s2, 136($t2)
                sw $s2, 260($t2)
                sw $s2, 264($t2)
                addi $s2, $zero, 0xff3333 #VERMELHA

             bne $a2,$s1,jogada_mesma_coordenada #Se a cor diferente de
azul ou verde nao e possivel jogar na posicao
    jr $ra

    quad_0x1:
    lw $a2, 516($t2)
        bne $a2, $s2, possivel_jogada_0x1
        possivel_jogada_0x1:
            move $fp, $zero
            sw $s2, 516($t2)
            sw $s2, 520($t2)
            sw $s2, 644($t2)
            sw $s2, 648($t2)
        beq $a2, $s2, jogada_mesma_coordenada
    jr $ra
    acertou_0x1:
        lw $a2, 516($t2)
        bne $a2, $s1, poss_jogada_0x1
```

```
            poss_jogada_0x1:
                addi $fp, $zero,1
                addi $s2, $zero, 0x17FD04  #Amarela
                sw $s2, 516($t2)
                sw $s2, 520($t2)
                sw $s2, 644($t2)
                sw $s2, 648($t2)
                addi $s2, $zero, 0xff3333 #VERMELHA
            bne $a2,$s1,jogada_mesma_coordenada
    jr $ra
```

3. PRINCIPAIS FUNÇÕES

- **Função principal (main)**

Visão geral da função

```
    main:

        jal cores
        jal define_fundo
        jal desenha_tabuleiro
        jal titulo_jogo
        jal menu_game

        cores:
            addi $s1, $zero, 0x0040ff #Azul
            addi $s2, $zero, 0xff3333 #VERMELHA
        jr $ra

        define_fundo:
            addi $t1, $zero, 1024   #mapa possui 1024 quadrados
            add $t2, $zero, $t1
              lui $t2, 0x1000      #posição inicial dos dados para serem
pintados
        jr $ra

        titulo_jogo:
            la $a0, titulo
            li $v0, 4
            syscall
```

```
        jr $ra

        menu_game:
            la $a0, txt_menu
            li $v0, 4
            syscall

            li $v0, 5   #Opcao ditigitada
            syscall

            move $k0, $v0

            beq $v0, 3, exit_game

            sge $a0, $v0, 4
            beq $a0, 1, msg_menu

            jal placar
            jal jogada_player1

        jr $ra

    jr $ra
```

A função principal é responsável por definir o fundo, ou seja, criar uma matriz com uma altura de 512 x 512 pixels, onde conterá 1024 quadrados de 16 x 16 pixels, Continuando, é responsável também por definir a cor de fundo (azul) da placa e a cor da bomba (vermelho).

- **Geração dos destroyers**

Visão geral da geração do destroyer 1.

```
    #gera destroyers 1
    jal coordenada_inicial
    loop:
        beq $a2,16, saiDoLoop
        sw $a0, Destroyer_1($a2)
```

```
        addi $a2, $a2, 4
        addi $a0, $a0,1
        j loop
    saiDoLoop:
```

Inicialmente, fazemos uso da instrução **jal** (jump and link) para acessar a função **coordenada_inicial.** Veja a imagem da função abaixo.

```
    coordenada_inicial:
       li $v0, 42  # 42 código de chamada de sistema para gerar int
       li $a1, 7   # $a1 limite (0 <= [int] <[limite superior])
       syscall     # gera o número e coloca em $a0
    jr $ra
```

A função **coordenada_inicial**, gerará um número inteiro maior ou igual a 0 e inferior a 7. Por conseguinte, utilizará esta função para gerar um número que se refere à primeira coluna do início do destroyer.

Vamos simular um exemplo que faz alusão às linguagens de alto nível, para maior clareza.

$a0 = return **coordenada_inicial:**

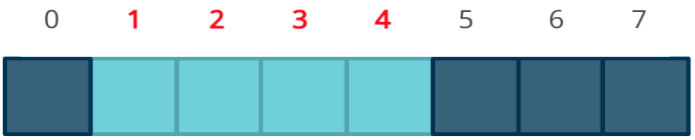
Valor obtido: 1
O valor obtido é colocado no registrador **$a0**

Assim que um número é armazenado no registo $a0, chamamos o procedimento de **loop:**

```
    loop:
        beq $a2,16, saiDoLoop        # loop 4x
        sw $a0, Destroyer_1($a2)
        addi $a2, $a2, 4             # $a2++
        addi $a0, $a0,1              # $a0++
        j loop
    saiDoLoop:
```

Este procedimento "pegará" o valor do registo $a0 e defini-lo-á como o primeiro valor da matriz. Este laço será executado 4 (quatro) vezes, incrementando cada vez o valor

armazenado em $a0 por 1 (um). Desta forma, temos as colunas que formam um destroyer de 4 (quatro) posições.



Uma vez feito isto, precisamos escolher uma linha aleatoriamente para o destroyer. Mais uma vez, utilizaremos a função **coordenada_inicial**. Agora já não para especificar uma coluna, mas sim uma linha.

Quando o programa começa, as linhas onde haverá destroyers são definidas aleatoriamente. O procedimento **while:** é invocado. A função **coordenada_inicial** é chamada e o valor gerado é armazenado num registo, neste caso $s6. Portanto, o valor armazenado corresponde a linha onde há um destroyer.

```
# ========== Gera coordenadas das linhas ==========

while:
    #gera coordenada linha destroyers 1
    jal coordenada_inicial
    move $s6, $a0
    #gera coordenada linha destroyers 2
    jal coordenada_inicial
    move $s5, $a0

    beq $s6, $s5, while

    #gera coordenada linha destroyers 3
    jal coordenada_inicial
    move $s4, $a0

    beq $s4, $s6, while
    beq $s4, $s5, while

    #gera coordenada linha destroyers 4
    jal coordenada_inicial
    move $s7, $a0

    beq $s7, $s6, while
```

```
        beq $s7, $s5, while
        beq $s7, $s4, while

        j exit_while
    j while
    exit_while:
```

Assim, em geral, inicialmente são armazenadas as coordenadas das linhas onde haverá destroyers, e durante o curso do programa são criadas as coordenadas iniciais das colunas que formam os destroyers de 4 (quatro) posições.

- **Função responsável pela jogada do jogador 1**

Visão geral da função responsável pela jogada do jogador 1

```
jogada_player1:

    jal on_player_1
    la $a0, linha
    li $v0, 4
    syscall
        jal quebra_linha
    la $a0, txt_player1
    syscall
        jal quebra_linha

    la $a0, txt_jogada_H
    li $v0, 4
    syscall

    li $v0, 5   #Le coluna
    syscall
    move $t5,$v0
        sge $a3,$t5,10
        beq $a3, 1, text_alerta_big_valor_coluna

    alerta_valor_linha: la $a0, txt_jogada_V
    li $v0, 4
```

```
        syscall

        li $v0, 5   #Le linha
        syscall
        move $t6,$v0
            sge $a3,$t6, 9
            beq $a3,1,text_alerta_big_valor_linha

        jal conta
        jal conta2
        jal conta3
        jal conta4

        jal get_coluna


        beq, $fp,1,player1_acertou


            #Se escolhida a opcao 1
            beq $k0,1,maquina_escolhe_jogada
            jal jogada_player2 #SE NAO

    jr $ra
```

Inicialmente a função mostra alguns textos no console. Ex:

```
    linha: .asciiz "\n"
    txt_jogada_H: .asciiz "Coluna: "
    txt_jogada_V: .asciiz "Linha: "
    txt_player1: .asciiz "Player 1, em qual posição deseja jogar? "
```

Em seguida faz a leitura da coordenada:

Coluna

```
        li $v0, 5   #Le coluna
        syscall
        move $t5,$v0
```

Linha

```
        li $v0, 5   #Le linha
```

```
        syscall
        move $t6,$v0
```

Há algumas verificações para os valores introduzidos para a coluna e para a linha.

```
move $t5,$v0
            sge $a3,$t5,10
            beq $a3, 1, text_alerta_big_valor_coluna
move $t6,$v0
            sge $a3,$t6, 9
            beq $a3,1,text_alerta_big_valor_linha
```

Logo em seguida, é chamado algumas instruções de **jal** (jump and link) para algumas funções.

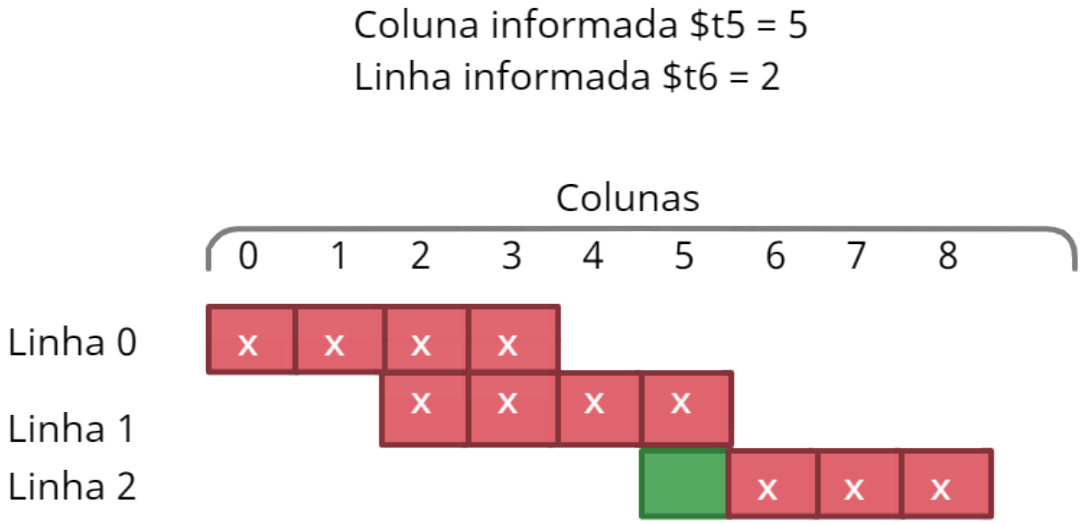
```
        jal conta
        jal conta2
        jal conta3
        jal conta4
```

Mencionaremos aqui como funciona utilizando a função **conta**. Nota: as outras seguem o mesmo procedimento.

```
    conta:
        move $a2, $zero
        print:
            beq $a2, 16, exit
            li $v0,1
            lw $a0, Destroyer_1 ($a2)
                beq $t5,$a0,posicao
                addi $a2, $a2, 4
        j print
    exit:jr $ra
```

A função **conta:** detém de um procedimento **print:** sendo responsável por mostrar se na coordenada informada há uma parte de um destroyer. A verificação é feita através de 4 voltas, onde é lido todos os vetores contendo as posições de todos os destroyer e, se o valor

que está no registrado **$t5** (responsável pelo valor da coluna) for igual a alguma posição e esta estiver em uma linha válida, logo temos uma posição válida. Temos um exemplo para ficar mais claro.



- **Procedimento para verificar o vencedor**

Visão geral do procedimento de verificação do vencedor.

```
verifica_vencedor:
    #Verifica empate
    beq  $v1, $t8, batalha_naval_empate

    #Verifica vencedor
    sgt $a0, $v1, $t8
    beq $a0, 1, vencedor_player1
    j vencedor_player2

    batalha_naval_empate:

        la $a0, msg_empate
        li $v0, 4
        syscall
```

```
        li $v0, 10      #Encerra o jogo
        syscall

    vencedor_player1:
        la $a0, player1
        li $v0, 4
        syscall

        li $v0, 10
        syscall
    vencedor_player2:
        la $a0, player2
        li $v0, 4
        syscall

        li $v0, 10
        syscall
```

Cada vez que um jogador encontra uma coordenada válida, 1 é adicionado à sua pontuação e é feita uma verificação se os 4 (quatro) destroyers já tiverem sido encontrados. Exemplo com a função para verificar se o jogador 1 acertou.

```
player1_acertou:
    addi $v1,$v1,1                  # Add 1 à pontuação

    jal placar
    add $a0, $v1, $t8
    beq $a0, 16, verifica_vencedor    # Faz a verificação

    jal jogada_player1
jr $ra
```

Uma vez que existem 4 destruidores, ou 16 pontos, quando esta pontuação é atingida pelo mesmo jogador ou dividida por ambos, o procedimento de verificação do vencedor é invocado.

```
    verifica_vencedor:
        #Verifica empate
        beq  $v1, $t8, batalha_naval_empate
```

Inicialmente, verifica-se se houve um empate, se for verdade, o procedimento **batalha_naval_empate** é chamado.

```
        batalha_naval_empate:

            la $a0, msg_empate
            li $v0, 4
            syscall

            li $v0, 10  # 10 código de chamada de sistema para exit
(terminate execution)
            syscall
```

Se não houver empate, o vencedor é verificado.

```
        #Verifica vencedor
        sgt $a0, $v1, $t8
        beq $a0, 1, vencedor_player1
        j vencedor_player2
```

Se o valor (pontos) contido no registrador $v1(jogador 1) for superior ao valor contido no registrador $t8 (jogador 2), então o procedimento **vencedor_player1** é chamado.

```
        vencedor_player1:
            la $a0, player1
            li $v0, 4
            syscall

            li $v0, 10
            syscall
```

Caso contrário, o vencedor foi o jogador 2. Então o procedimento **vencedor_player2** é chamado.

```
    vencedor_player2:
        la $a0, player2
        li $v0, 4
        syscall

        li $v0, 10
        syscall
```

- **“IA” jogando**

```
IA_acertou:
    addi $t8,$t8,1

    jal placar
    add $a0, $v1, $t8
    beq $a0, 16, verifica_vencedor


        jal on_player_2
        la $a0, maquina_jogando
        li $v0, 4
        syscall
        li $v0, 32
        li $a0, 4000
        syscall

        addi $t5,$t5,1

            sgt $a0,$t5,9
            beq $a0,1, IA_jogada_normal

        jal opcao_H
            move $a0,$t5
            li $v0, 1
            syscall
            jal quebra_linha
        jal opcao_V
```

```
            move $a0,$t6
            li $v0, 1
            syscall
        move $t6,$t6

        jal conta
        jal conta2
        jal conta3
        jal conta4

        jal get_coluna
            beq, $fp,1,IA_acertou
        jal jogada_player1
        jal  maquina_escolhe_jogada
jr  $ra

IA_jogada_normal:
        jal on_player_2
        la $a0, maquina_jogando
        li $v0, 4
        syscall
        li $v0, 32
        li $a0, 4000
        syscall

        jal opcao_H
        jal jogada_horizontal
            move $t5,$a0
            jal quebra_linha
        jal opcao_V
        jal jogada_vertical
            move $t6,$a0

            jal conta
            jal conta2
            jal conta3
            jal conta4

            jal get_coluna
                beq, $fp,1,IA_acertou
            jal jogada_player1

        jal  maquina_escolhe_jogada
```

```
    jr $ra
```

- **Jogada na mesma posição**

```
    quad_0x0:
    lw $a2, 132($t2)
        bne $a2, $s2, possivel_jogada_0x0
        possivel_jogada_0x0:
            sw $s2, 132($t2)
            sw $s2, 136($t2)
            sw $s2, 260($t2)
            sw $s2, 264($t2)

            beq $a2, $s2, jogada_mesma_coordenada
    jr $ra
    acertou_0x0:
        lw $a2, 132($t2)
        beq $a2, $s1, poss_jogada_0x0 #Se a posicao estiver em azul, e
possivel jogar
            poss_jogada_0x0:
                addi $fp, $zero,1
                addi $s2, $zero, 0x17FD04  #Amarela
                sw $s2, 132($t2)
                sw $s2, 136($t2)
                sw $s2, 260($t2)
                sw $s2, 264($t2)
                addi $s2, $zero, 0xff3333 #VERMELHA

            bne $a2,$s1,jogada_mesma_coordenada #Se a cor diferente de
azul ou verde nao e possivel jogar na posicao
    jr $ra
```

Todas as vezes que for feita uma jogada, é verificada a cor da posição.

```
    quad_0x0:
    lw $a2, 132($t2)
        bne $a2, $s2, possivel_jogada_0x0
```

É feito o acesso à memória (**LW**) referente a coordenada, e consequentemente verificada se a cor armazenada em **$a2** é igual a cor no registrador **$s2**. Caso a jogada não seja uma coordenada válida. Onde a cor armazenada em $s2 e vermelha.

```
        cores:
            addi $s1, $zero, 0x0040ff #Azul
            addi $s2, $zero, 0xff3333 #VERMELHA
        jr $ra
```

Se a coordenada for uma coordenada válida, é feito uma verificação se a cor armazenada no registrador **$a2** é igual a **$s1** (cor azul).

```
acertou_0x0:
        lw $a2, 132($t2)
        beq $a2, $s1, poss_jogada_0x0 #Se a posicao estiver em azul, e
possivel jogar
            poss_jogada_0x0:
                addi $fp, $zero,1
                addi $s2, $zero, 0x17FD04  #Amarela
                sw $s2, 132($t2)
                sw $s2, 136($t2)
                sw $s2, 260($t2)
                sw $s2, 264($t2)
                addi $s2, $zero, 0xff3333 #VERMELHA

            bne $a2,$s1,jogada_mesma_coordenada #Se a cor diferente de
azul ou verde nao e possivel jogar na posicao
    jr $ra
```

4. MELHORIAS FUTURAS

Dado o que foi feito, acreditamos que a aplicação ainda é capaz de receber futuras atualizações, tais como uma melhor modularização em parte do código ou mesmo a automatização da criação de linhas e colunas para a interface gráfica de uma forma mais compacta, reduzindo assim drasticamente a quantidade de linhas do código total.

Para uma melhor utilização da montagem, poderíamos implementar funções de montagem nativas, referenciá-las e criar uma biblioteca para a sua utilização, e depois chamá-las dentro da linguagem de programação C (manual do utilizador Hi-Tech C, 1989). E finalmente, devido à dificuldade encontrada na geração de movimentos mais inteligentes pela máquina e na geração de números aleatórios para posicionar destroyers em todo o mapa, juntamente com a dificuldade encontrada para gerir a utilização de registradores de uma forma mais eficiente. Não foi possível até agora, a implementação de rotinas que pudessem gerar destroyers de modo a posicioná-los verticalmente sobre o mapa.

5. CONCLUSÃO

Como demonstrado até agora, a aplicação foi implementada de forma a satisfazer os requisitos necessários para que o jogo funcionasse de forma semelhante ao que já é conhecido tradicionalmente. Também utilizamos todas as técnicas e instruções mostradas na aula e, apesar das limitações, acreditamos que tudo foi implementado de uma forma satisfatória. Além disso, de acordo com o que foi dito e demonstrado, o principal objetivo deste projeto foi alcançado, demonstrar num projeto a implementação prática de uma aplicação utilizando linguagem de montagem (assembly- MIPS).

REFERÊNCIAS

SYSCALL functions available in MARS. Disponível em: http://courses.missouristate.edu/kenvollmar/mars/help/syscallhelp.html. Acesso em 7 de janeiro, 2021.

IoT faz linguagem Assembly retomar popularidade momentânea. Disponível em: https://computerworld.com.br/plataformas/iot-faz-linguagem-assembly-retomar-popularidade-momentanea/. Acesso em 21 de janeiro, 2021.