



**UNIVERSIDADE FEDERAL DE SANTA MARIA (UFSM)**  
**CIÊNCIA DA COMPUTAÇÃO**

**Relatório de Projeto do Snake com assembly MIPS no MARS 4.5**

João Antonio Guerim Guasso

Leonardo Martins Brisolla

Santa Maria, RS  
2024

**João Antonio Guerim Guasso**

**Leonardo Martins Brisolla**

## **Relatório de Projeto de Organização de Computadores**

Relatório de projeto apresentado à disciplina de Organização de Computadores, do curso de Ciência da Computação, da Universidade Federal de Santa Maria (UFSM).

Professor: Giovani Baratto

Santa Maria, RS  
2024

## LISTA DE FIGURAS

Figura 1 - Display Snake.....	8
Figura 2 - Colisão com o próprio corpo .....	8
Figura 3 - Colisão com a borda .....	9
Figura 4 - mover_cobra (parte 1).....	12
Figura 5 - mover_cobra (parte 2).....	12
Figura 6 - mover_cobra (parte 3).....	13
Figura 7 - retirar_do_fim.....	13
Figura 8 - move_restante_cobra .....	14
Figura 9 - coordenada_aleatoria .....	15
Figura 10 - verificar_derrota .....	16

## SUMÁRIO

<b>1</b>	<b>INTRODUÇÃO .....</b>	<b>5</b>
1.2	OBJETIVO GERAL .....	5
1.3	OBJETIVOS ESPECÍFICOS .....	5
<b>2</b>	<b>FERRAMENTAS .....</b>	<b>6</b>
2.1	MARS 4.5 .....	6
2.2	ASSEMBLY MIPS .....	6
2.3	BITMAP DISPLAY E KEYBOARD AND DISPLAY MMIO SIMULATOR .....	6
<b>3</b>	<b>METODOLOGIA.....</b>	<b>7</b>
3.1	EXPERIMENTO .....	7
3.2	DESAFIOS .....	9
<b>4</b>	<b>CÓDIGO DESENVOLVIDO.....</b>	<b>9</b>
<b>5</b>	<b>CONCLUSÃO.....</b>	<b>16</b>
	<b>REFERÊNCIAS BIBLIOGRÁFICAS.....</b>	<b>17</b>

## **1) INTRODUÇÃO**

O desenvolvimento desse projeto foi realizado como forma de colocar em prática os conhecimentos adquiridos na disciplina de Organização de Computadores. Para isso, foi escolhido pelo grupo o jogo “Snake”, sendo projetado para funcionar dentro da IDE MARS 4.5 com assembly MIPS.

Para a criação da parte gráfica do jogo foram usados o Bitmap Display, para criar um jogo com um gráfico mais chamativo para o usuário, e o Keyboard and Display MMIO Simulator, para criar uma interação mais fluída com a cobra.

Ao longo desse relatório será abordado sobre quais são os conhecimentos necessários para criar o jogo, como utilizar a ferramenta para executar ele e qual o código implementado para cada funcionalidade utilizada.

### **1.1) OBJETIVOS GERAIS**

O objetivo desse projeto é mostrar como pode ser criado o jogo Snake no MARS 4.5, servindo como uma espécie de manual para os interessados em reproduzir e fornecendo um código capaz de funcionar de maneira interativa com o jogador, da mesma forma que o jogo na sua forma convencional. A criação de uma interface gráfica foi realizada como uma forma de estimular o usuário a ter interesse pelas ferramentas usadas.

### **1.2) OBJETIVOS ESPECÍFICOS**

Considerando todas as funcionalidades implementadas no jogo desse projeto, pode-se dizer que tivemos como objetivos:

- Implementar o jogo Snake com a linguagem assembly MIPS.
- Entender a IDE MARS 4.5.
- Compreender o funcionamento do Bitmap Display e do Keyboard and Display MMIO Simulator.
- Realizar integração de ferramentas gráficas com o código da linguagem selecionada.

## **2) FERRAMENTAS**

Como dito anteriormente, para o desenvolvimento do jogo foi necessário o uso de conhecimentos em assembly MIPS, MARS 4.5, Bitmap Display e Keyboard and Display MMIO Simulator, ferramentas que serão abordadas com mais detalhes nesse tópico. Como o objetivo principal do relatório é mostrar como criamos o jogo a partir dessas ferramentas, o enfoque será dado ao uso delas dentro do nosso projeto.

### **2.1) MARS 4.5**

Basicamente, o MARS 4.5 é um ambiente de desenvolvimento integrado (IDE) designado para podermos programar em assembly MIPS, sendo muito usado em ambientes acadêmicos para aprender essa linguagem. É dentro dele que editamos o código, usamos o Bitmap Display e o simulador de leitor de caracteres (para não precisar apertar “enter” a cada comando do teclado). Dentro dele temos acesso ao editor de código e aos registradores na aba “Edit”, assim como, na aba de execução “Execute”, vemos as instruções e os dados usados na implementação.

### **2.2) ASSEMBLY MIPS**

Essa linguagem de programação de baixo nível, feita para ser próxima do hardware, possibilita o controle sobre a arquitetura do computador. Além disso, permite que possamos ver com mais detalhes como o processador executa as instruções e gerencia os recursos da memória e registradores durante a execução do jogo. Durante a criação do jogo foi evitada a utilização de muitos pseudocódigos, priorizando comandos de instruções básicas.

### **2.3) BITMAP DISPLAY E KEYBOARD AND DISPLAY MMIO SIMULATOR**

O Bitmap Display do MARS 4.5 é um simulador de tela gráfica que permite a criação de gráficos para o jogo criado, isso sendo feito a partir da impressão de pixels individuais na tela do usuário. No “Snake”, ele foi usado para mostrar um fundo branco, a cobra em movimento e as maçãs que o jogador deve pegar com a cobra. O código base do Bitmap Display foi alterado para possibilitar a tela com 16x16 e melhorar a experiência do jogador.

O KEYBOARD AND DISPLAY MMIO SIMULATOR foi usado para simular a interação do teclado com o programa em Assembly MIPS. A leitura do teclado gera uma exibição da saída em uma interface de texto. No jogo do projeto, as teclas que o usuário deve digitar, dentro desse simulador, são W, A, S, D, permitindo a movimentação da cobra como no jogo convencional.

### **3) METODOLOGIA**

Para o desenvolvimento do jogo foram criados diversos procedimentos que se adequaram a cada parte do jogo, sendo testados ao longo da produção do código, começando pelo desenho do movimento da cobra e terminando pela produção da maçã em diferentes posições da tela. A integração na produção do código foi feita pela ferramenta Github, permitindo acesso ao código atualizado por outro membro do grupo. O sistema do jogo foi planejado baseado principalmente nas estruturas de apresentação dos componentes do jogo (cobra e maçã), movimentação da cobra, detecção de qualquer colisão, manipulação de entrada do usuário e pelo sistema de pontuação.

#### **3.1) EXPERIMENTO**

Para iniciar o jogo, o usuário deve seguir o seguinte passo a passo para iniciar o jogo:

- Abrir Bitmap Display e o KEYBOARD AND DISPLAY MMIO SIMULATOR.
- Clicar em “Connect to MIPS” dentro dessas duas abas abertas.
- Clicar na chave de fenda da parte superior para reiniciar andamento do código (deve fazer isso toda vez que quiser reiniciar o jogo também).
- Deve configurar o Bitmap Display para a seguinte configuração:
  - Unit Width in Pixels: 16.
  - Unit Height in Pixels: 16.
  - Display Width in Pixels: 512.
  - Display Height in Pixels: 512.
- Redimensionar o Bitmap Display para a tela preta atingir seu tamanho total.
- Clicar em “Run the current program”.
- Clicar com o cursor do mouse dentro do KEYBOARD AND DISPLAY MMIO SIMULATOR (vai ser onde serão captados os comandos).

- Sempre que for reiniciar o jogo aperte em “Reset” dentro das duas abas abertas no primeiro passo, clicar na chave de fenda e dar “Run” novamente.

Após seguir esses passos, a interface do usuário estará pronta para jogar.

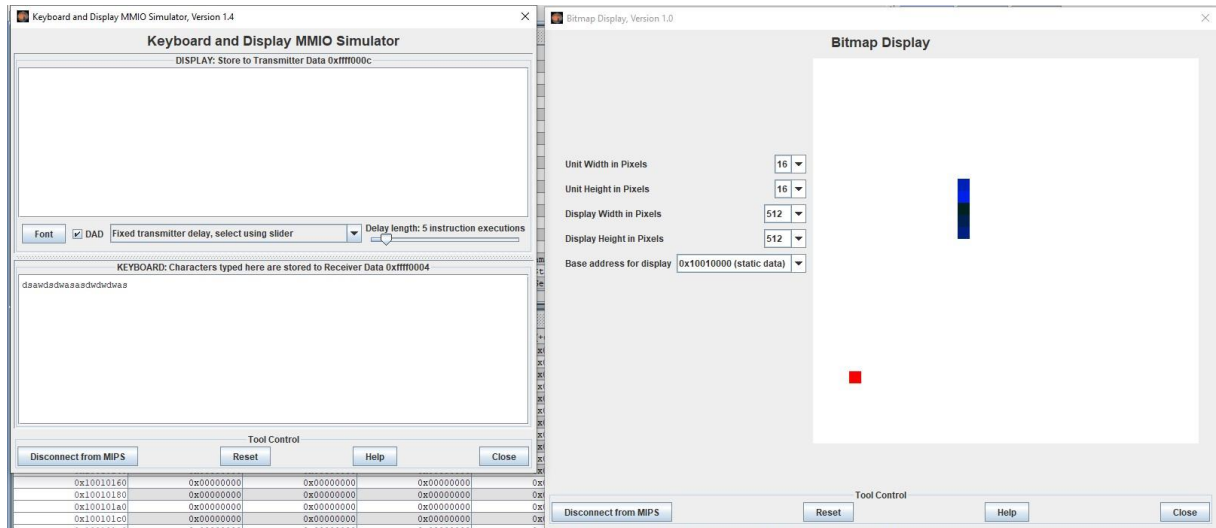


Figura 1 - Display Snake

O jogo funciona da mesma forma que o “Snake” basta decidir a direção que a cobra vai caminhar usando W, A, S, D e pegar as maçãs que vão ser espalhadas pelo mapa. A coleta de maçãs faz a cobra crescer. Para o jogo acabar basta a cobra bater no próprio corpo ou bater na borda do Bitmap Display.

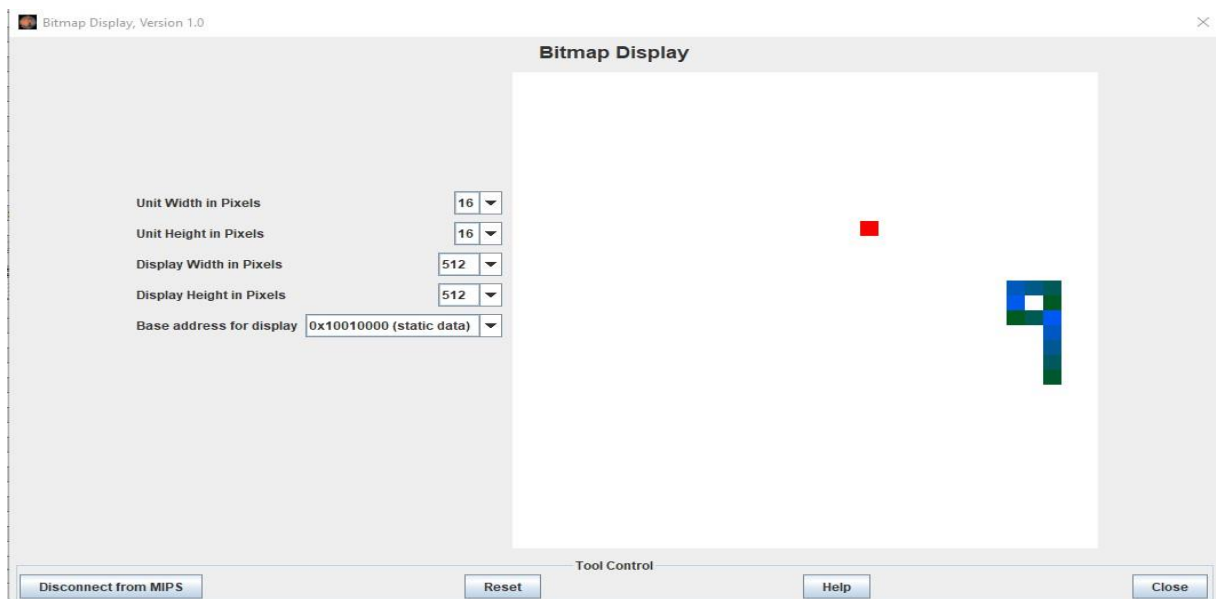


Figura 2 - Colisão com o próprio corpo

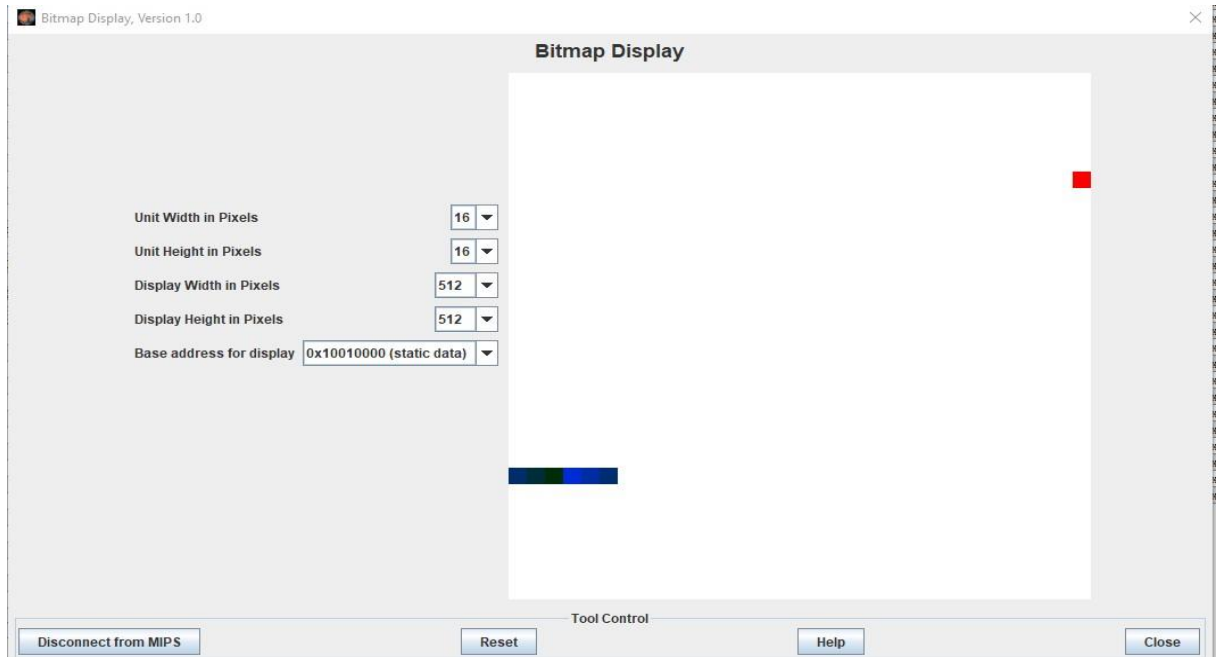


Figura 3 - Colisão com a borda

### 3.2) DESAFIOS

Ao longo do desenvolvimento do projeto foram encontrados alguns desafios pelos membros do grupo, entre eles podemos destacar a dificuldade de entendimento do código feito por outro membro, aliando-se ao desafio de integrar o código Assembly ao Bitmap Display para tornar o jogo com interações gráficas.

Um desafio encontrado durante a criação do jogo foi a biblioteca do Bitmap Display, "include display\_bitmap.asm", na qual estava interferindo com as inicializações de variáveis ".data". Para resolver esse problema, a função "init" foi realocada para ficar acima dessa biblioteca, resolvendo esse problema encontrado. Outro empecilho enfrentado pela equipe foi a limitação do leitor de teclas da ferramenta Keyboard Simulator onde, caso seja pressionado qualquer tecla (fazendo com que ela se repita várias vezes), o Bitmap Display acaba travando e apresentando uma queda de frames no jogo. Fizemos tentativas de tratamento para isso, porém acreditamos ser uma limitação da própria ferramenta para essa situação.

## 4) CÓDIGO DESENVOLVIDO

Inicialmente foram feitas as inicializações de cada variável no procedimento **init**, dividido na parte .text e .data, sendo elas em .text: S0 recebendo 16 como coordenada inicial x e S1 como y também, ambas coordenadas iniciais da cobra, a formação da seed para gerar um

número aleatório dentro de S2, S3 sendo a direção da cobra e começando com mExit (impossibilitando que ande inicialmente), S4 é a cor da cobra (que sofre alterações), S5 sendo usado como x da maçã e S6 como y, S7 foi usado como comprimento da cobra e inicializado como 1. S4, S5, S6 não precisaram ser inicializados no init. Após essas inicializações, chama-se a função `init_graph_test` para printar o fundo do jogo e a função `gera_maca` para gerar as coordenadas da maçã. Com isso feito, chamamos a `main` para começar o jogo. O `.include "display_bitmap.asm"` ficou abaixo das inicializações para evitar incompatibilidade do código (estava gerando problemas). Na seção `.data` foram inicializados, com `.space`, `x` e `y` com tamanho de 2048, pois  $32 \times 32 = 1024$  e cada word precisa de 2 bytes então usamos 2048. Também foi inicializada a `seed`, com `.word` e números primos (para a maçã não ser previsível ao jogador). Para finalizar o programa foi usado um procedimento chamado `finit` que coloca o valor de retorno da `main` em `a0`, depois coloca o serviço 17 em `v0`, usado como forma de trancar o funcionamento do programa e usa `syscall` para fazer uma chamada de sistema com o serviço 17 identificado em `v0` e passado para `a0`.

Após isso, na **main** foi feito um loop `lacoP` (chamado com um `j` até acabar o jogo) que chama, respectivamente, os procedimentos a seguir: `verifica_pontuacao` (vê se a maçã foi comida), `desenha_maca` (desenha a maçã), `mover_cobra` (movimento da cobra) e `verificar_derrota` (bateu no próprio corpo). Esses procedimentos chamados pelo comando `jal` representam o funcionamento do jogo, terminando quando o `finit` é chamado por `jr $t0`, onde `t0` recebe `finit` com a instrução `la`.

Por conseguinte, todos os procedimentos de funcionamento e apresentação do jogo, que serão explicados a partir de agora, apresentam uma padrão de código inserido neles. Esse “esqueleto” de procedimento torna a função dividida em, respectivamente, em prólogo, corpo do procedimento e epílogo. O padrão usado está no prólogo e no epílogo. O prólogo é usado para preparar o ambiente para execução de um procedimento, reservando 4 bytes na pilha (sendo `$sp` o ponteiro da pilha que recebe os 4 bytes necessários para armazenar `$ra`) com `addi $sp, $sp, -4` e armazenando o endereço de retorno na pilha, salvando o estado atual dela, a partir do `sw $ra, 0($sp)`. O epílogo serve para restaurar o endereço de retorno com `lw $ra, 0($sp)`, restaurar a pilha original com `addiu $sp, $sp, 4` e retornar ao procedimento chamador do procedimento atual com `jr $ra`. O corpo do procedimento contém as instruções e a lógica principal da função, sendo criada para desempenhar um papel específico do programa.

Feito isso, para desenhar o fundo branco foi usado o procedimento **init\_graph\_test**, função que coloca a cor WHITE em a0 e, por conseguinte, chama utilizando a instrução jal a função set\_background\_color, que envia a cor branca selecionada e um jal para screen\_init2 que inicializa a tela gráfica. Ambas as funções chamadas são ferramentas disponíveis na biblioteca “display\_bitmap.asm”.

O desenho da cobra é feito no procedimento **desenha\_cobra**, no qual cria um efeito de mudança de cor na cobra, usando addi \$s4, \$s4, 50 (lembrando que \$s4 é a cor dela) como forma de alterar gradualmente a cor da cobra e andi \$s4, \$s4, 0x00FFFFFF para aplicar uma máscara binária no valor de \$s4 (fazendo com que permaneça em um intervalo válido para uma cor RGB). A cor da cobra presente em \$s4 é enviada para set\_foreground\_color que configura a cor de desenho correspondente. Para desenhar o pixel, com a cor já selecionada, foram enviados o x (\$s0) e o y (\$s1) como argumentos para o procedimento put\_pixel que desenha um pixel na tela conforme as coordenadas fornecidas.

Agora vamos abordar, da forma mais sucinta possível, sobre o procedimento **mover\_cobra**, responsável por fazer o movimento da cabeça da cobra. Primeiramente, foi implementada a leitura de caractere pelo Keyboard and Display MMIO Simulator pegando o endereço do RDR (Receiver Data Register) 0Xffff0004 e colocando em \$t0 que passa o caractere dele para \$t2 (caractere digitado no terminal). Após isso, foi feito um limitador de velocidade da cobra a partir de um syscall 32 que recebe como argumento um a0 com 100 (tempo de espera em milissegundos). Para impossibilitar o usuário de fazer movimentos proibidos no jogo snake, como tentar ir para baixo com a cobra indo para cima, foi feito um tratamento onde se \$s3 está indicando a direção para mCima ou mBaixo, então pula para mHorizontal, ou se indica a direção para mEsquerda e mDireita, então pula para mVertical. No mVertical ele confere se a tecla digitada (\$t2) é igual a w, W, s ou S (maiúsculos tratados) para ir ao pequeno procedimento auxiliar da direção correspondente (mCima ou mBaixo), da mesma forma acontece no mHorizontal com d, D, a ou A (mDireita ou mEsquerda). Caso não venha um input válido ele retorna \$s3 e mantém direção. Qualquer direção válida escolhida tem verificação de colisão que chama o finit (caso ocorra) e, também, possui o chamamento dos procedimentos de retirar\_do\_fim (remove último segmento da cobra) e do move\_restante\_cobra (move segmentos intermediários). No movimento mCima é carregada a coordenada x (\$s0), verificada colisão com borda superior (\$zero), x é decrementado (subida) e atualiza \$s3 com a direção atual (mCima). Acontece a mesma coisa para mBaixo mas a coordenada x é incrementada em 1 (descida) e verifica a colisão com a borda inferior (31). No

movimento mDireita é carregada a coordenada y (\$s1), verificada colisão com a borda (31), y é incrementado (direita) e atualiza \$s3 com a direção atual (mDireita). Da mesma forma com mBaixo, porém a coordenada x é decrementada em 1 (esquerda) e verifica colisão com a borda da esquerda (\$zero).

```

151 # Procedimento que move a cabeça da cobra(parte dianteira)
152 #####
153 mover_cobra:
154     # prólogo
155     addiu $sp, $sp, -4      # ajustamos a pilha
156     sw $ra, 0($sp)         # armazenamos o endereço de retorno na pilha
157     #Recebe o caracter digitado no Keyboard and Display MVTIO Simulator
158     la $t0, 0xFFFF0004     # endereço do RDR (Receiver Data Register)
159     lw $t2, 0($t0)         # $t2 <- caracter do terminal
160
161     #Syscall de sleep de 100 milissegundos (0.1 segundos)
162     #Para limitar a velocidade da cobra
163     li $v0, 32 #Número da Syscall
164     li $a0, 100 #Tempo de espera em milissegundos
165     syscall
166
167     #Para evitar movimentações erradas, como o usuario se movimentar contra o próprio corpo da cobra
168     #Caso a cobra esteja se movendo no eixo VERTICAL
169     la $t3, mCima
170     beq $t3, $s3, mHorizontal
171     la $t3, mBaixo
172     beq $t3, $s3, mHorizontal
173     #Caso a cobra esteja se movendo no eixo HORIZONTAL
174     la $t3, mEsquerda
175     beq $t3, $s3, mVertical
176     la $t3, mDireita
177     beq $t3, $s3, mVertical
178
179     #Movimentações possíveis para o eixo vertical
180     #Tratamento para os caracteres maiúsculos a partir de seu código ASCII
181     mVertical:
182     beq $t2, 0x77, mCima #W
183     beq $t2, 0x57, mCima #V
184     beq $t2, 0x73, mBaixo #S
185     beq $t2, 0x53, mBaixo #5
186
187     #Caso nenhum caractere valido seja verificado
188     #Se a cobra estiver em movimento, continua na mesma direção
189     #Se a cobra estiver parada, verificasse o eixo horizontal
190     la $t3, mExit

```

Figura 4 - mover\_cobra (parte 1)

```

190     la $t3, mExit
191     bne $s3, $t3, mElse
192
193
194
195     #Movimentações possíveis para o eixo horizontal
196     #Tratamento para os caracteres maiúsculos a partir de seu código ASCII
197     mHorizontal:
198     beq $t2, 0x44, mDireita #D
199     beq $t2, 0x64, mDireita #d
200     beq $t2, 0x41, mEsquerda #A
201     beq $t2, 0x61, mEsquerda #a
202
203
204     mElse:
205     #Continua o mesmo movimento caso NENHUM caracter valido tenha sido digitado
206     jt $s3
207
208     #Movimentação para cima
209     mCima:
210     #Retira o fim da cobra
211     jal retirar_do_fim
212
213     #Altera o resto das coordenadas da cobra
214     jal move_restante_cobra
215
216     lw $t0, 0($s0) # $t0 <- x(cabeça da cobra)
217     beq $t0, $zero, finit # Caso a cobra bata na parte superior da tela
218     addi $t0, $t0, -1 #ajusta a posição da coordenada x da cabeça
219     sw $t0, 0($s0) #salva o ajuste
220     la $s3, mCima #salva a direção que a cobra está indo
221     j mExit #pula pro final
222     mBaixo:
223     #Retira o fim da cobra
224     jal retirar_do_fim
225
226     #Altera o resto das coordenadas da cobra
227     jal move_restante_cobra
228     lw $t0, 0($s0) # $t0 <- x(cabeça da cobra)
229     beq $t0, 31, finit # Caso a cobra bata na parte inferior da tela

```

Figura 5 - mover\_cobra (parte 2)

```

225
226      #Altera o resto das coordenadas da cobra
227      jal move_restante_cobra
228      lw $t0, 0($s0) # $t0 <- x(cabeça da cobra)
229      beq $t0, 31, finit # Caso a cobra bata na parte inferior da tela
230      addi $t0, $t0, 1 #ajusta a posição da coordenada x da cabeça
231      sw $t0, 0($s0) #salva o ajuste
232      la $s3, mBaixo #salva a direção que a cobra está indo
233      j mExit #pula pro final
234
235      mDireita:
236      #Retira o fim da cobra
237      jal retirar_do_fim
238
239      #Altera o resto das coordenadas da cobra
240      jal move_restante_cobra
241      lw $t1, 0($s1) # $t1 <- y(cabeça da cobra)
242      beq $t1, 31, finit # Caso a cobra bata na parte a direita da tela
243      addi $t1, $t1, 1 #ajusta a posição da coordenada y da cabeça
244      sw $t1, 0($s1) #salva o ajuste
245      la $s3, mDireita #salva a direção que a cobra está indo
246      j mExit #pula pro final
247
248      mEsquerda:
249      #Retira o fim da cobra
250      jal retirar_do_fim
251
252      #Altera o resto das coordenadas da cobra
253      jal move_restante_cobra
254      lw $t1, 0($s1) # $t1 <- y(cabeça da cobra)
255      beq $t1, $zero, finit # Caso a cobra bata na parte a esquerda da tela
256      addi $t1, $t1, -1 #ajusta a posição da coordenada y da cabeça
257      sw $t1, 0($s1) #salva o ajuste
258      la $s3, mEsquerda #salva a direção que a cobra está indo
259
260      mExit:# fim
261      # epílogo
262      lw $ra, 0($sp) # restauramos o endereço de retorno
263      addiu $sp, $sp, 4 # restauramos a pilha
264      jr $ra # retornamos ao procedimento chamador
265

```

Figura 6 - mover\_cobra (parte 3)

O procedimento **retirar\_do\_fim**, citado anteriormente, serve para retirar o último pixel da cobra (simular movimento). Inicialmente, ele lê a cor branca e chama a função `set_foreground_color` para setar essa cor. Feito isso, salvamos a posição x e y da cobra e o seu comprimento, para poder reduzir em 1 o comprimento da cobra. Com o valor do novo comprimento em uma variável temporária `$t2`, pegamos esse índice e multiplicamos por 4 (cada coordenada tem 4 bytes) e adicionamos esse deslocamento dentro de uma variável temporária de x e y. Enviamos esse x-1 e y-1 para o `put_pixel` para printar. Lembrando que isso não modifica o `$s7` (comprimento da cobra), mantendo o tamanho da cobra original.

```

409 #####
410 # Retira a ultima célula da cobra para ela se movimentar
411 #####
412 retirar_do_fim:
413     # Pro?o
414     addiu $sp, $sp, -4 # Ajustamos a pilha
415     sw $ra, 0($sp) # Armazenamos o endereço de retorno na pilha
416     # Corpo do procedimento
417     #Seleciona a cor branca(cor do fundo)
418     li $a0, WHITE
419     jal set_foreground_color
420
421     move $t0, $s0#$t0<-X
422     move $t1, $s1#$t1<-Y
423     move $t2, $s7#$t2<-Comprimento
424
425     #calcula o final da cobra
426     addi $t2, $t2, -1#$t2 <- $t2 - 1 | (Comprimento - 1)
427     sll $t2, $t2, 2 # $t2 <- $t2 * 4 | (Comprimento - 1)*4
428     add $t0, $t2, $t0#$t0 <- $t2 + $t0 | $t0 = x[Comprimento-1](endereço)
429     add $t1, $t2, $t1#$t1 <- $t2 + $t1 | $t1 = y[Comprimento-1](endereço)
430
431     #Carrega as coordenadas do final da cobra
432     lw $a0, 0($t0) # $a0 <- X[Comprimento-1](valor)
433     lw $a1, 0($t1) # $a1 <- Y[Comprimento-1](valor)
434     jal put_pixel #Coloca o pixel
435
436     # Ep?o
437     lw $ra, 0($sp) # Restauramos o endereço de retorno
438     addiu $sp, $sp, 4 # Restauramos a pilha
439     jr $ra # Retornamos ao procedimento chamador
440
441
442
443 #####

```

Figura 7 - retirar\_do\_fim

O procedimento **move\_restante\_cobra**, também chamado anteriormente pelo **mover\_cobra**, é responsável por atualizar o restante da cobra. Com **\$s7** pegamos o comprimento total da cobra, decrementado por 1 (cabeça está na posição 0 e não deve ser contada), para multiplicar por 4 e termos o deslocamento total em bytes. Pegamos as últimas posições nos arrays **x** (**\$t0**) e **y** (**\$t1**) somando a posição de cada um pelo deslocamento total. Verificamos se o endereço da cabeça é igual ao endereço final de **x**, se sim acaba ele pula para o fim do procedimento. Para o movimento das células da cobra existir foi feito um laço onde foram carregados os valores do **x**(**\$t0**) e do **y**(**\$t1**) da célula anterior e inseridos na célula atual. Para a próxima etapa de dentro do loop foram feitos ajustes que movem as coordenadas **x** e **y** 4 bytes para trás. O laço continua enquanto **\$t0** não chegar ao início da cobra **\$s0**.

```

268 #####
269 move_restante_cobra:
270     # Pr?o
271     addiu $sp, $sp, -4      # Ajustamos a pilha
272     sw    $ra, 0($sp)      # Armazenamos o endere?de retorno na pilha
273     # corpo do procedimento
274     #for(i=c;i>0;i--)
275     #Comprimento - 1 pois a cabeça está na posição 0
276     addi $t0, $s7, -1 # $t0 <- Comprimento - 1
277     sll $t0, $t0, 2 # $t0 <- 4*(Comprimento - 1)
278     #Seleciona a ultima posição de ambos arrays
279     add $t1, $t0, $s1 # $t1 <- Y[Comprimento - 1]
280     add $t0, $t0, $s0 # $t0 <- X[Comprimento - 1]
281
282     #Se o endereço do inicio for igual ao final, pule para o final
283     beq $t0, $s0, mrExit
284     #laço para mover todas as coordenadas
285     mrFor:
286         #X[Comprimento - i - 1] = X[Comprimento - i - 2]
287         lw $t3, -4($t0) #célula anterior(x)
288         sw $t3, 0($t0) #salva a célula anterior na próxima(x)
289         #Y[Comprimento - i - 1] = Y[Comprimento - i - 2]
290         lw $t3, -4($t1) #célula anterior(y)
291         sw $t3, 0($t1) #salva a célula anterior na próxima(y)
292
293         #anda uma célula para trás
294         addi $t0, $t0, -4
295         addi $t1, $t1, -4
296
297         #Se a célula for diferente da inicial, continue o laço
298         bne $t0, $s0, mrFor
299
300     # Ep?o
301     mrExit:
302         lw    $ra, 0($sp)      # Restauramos o endere?de retorno
303         addiu $sp, $sp, 4      # Restauramos a pilha
304         jr    $ra              # Retornamos ao procedimento chamador
305 #####

```

Figura 8 - move\_restante\_cobra

O **desenha\_maca** é um procedimento com a função de desenhar a maçã na tela. Para isso, é enviada a cor **RED** para **set\_foreground\_color**. Com a cor definida, são enviadas as coordenadas **x** (**\$s5**) e **y** (**\$s6**) da maçã para o **put\_pixel** desenhar o pixel dela. O **gera\_maca** serve para determinar a posição da nova maçã, chamando o procedimento **coordenada\_aleatoria** para pegar o **x**(**\$s5**) e **y**(**\$s6**) da nova maçã. O **coordenada\_aleatoria** gera valores pseudoaleatórios a partir de um gerador linear congruente (LCG), gerando assim as coordenadas da maçã. Iniciamos carregando a seed atual, a constante multiplicativa e a constante de incremento para fazer o cálculo. Para o cálculo usamos a multiplicação da seed

pela constante multiplicativa e depois somamos pela constante de incremento. Com o cálculo pronto, fizemos o tratamento para que o número gerado esteja entre 0 e 31 (tamanho máximo do grid) a partir da definição de um limite em \$t1 com 32 e com o cálculo do resto com rem, sendo feito entre o valor gerado e o 32. Após isso, basta enviarmos a seed ao \$s2 com o endereço da maçã e retornarmos esse valor gerado. Para verificar se a cabeça da cobra comeu a maçã (cabeça na mesma coordenada da maçã) usamos o procedimento **verifica\_pontuacao**. Ele carrega as coordenadas x (\$s0) e y (\$s1) da cobra e compara com a posição x (\$s5) e y (\$s6), caso um deles não for igual o programa pula para o final da função. Caso contrário, o programa irá adicionar 1 ao tamanho total da cobra (\$s7) e chama o procedimento gera\_maca para criar outra.

```

352 #####
353 # Gera as coordenada Pseudo-aleatória a partir de uma seed, e de constantes de multiplicação e incremento
354 #####
355 coordenada_aleatoria:
356     # Pr?o
357     addiu    $sp, $sp, -4          # Ajustamos a pilha
358     sw      $ra, 0($sp)          # Armazenamos o endereço de retorno na pilha
359     # Corpo do procedimento
360     lw      $t1, 0($s2)          # $t1 <- seed
361     lw      $t2, 4($s2)          # $t2 <- (a) constante multiplicativa
362     lw      $t3, 8($s2)          # $t3 <- (c) constante de incremento
363     mul     $t0, $t1, $t2        # $t0 <- seed * a
364     add     $t0, $t0, $t3        # $t0 <- ($t0 + c)
365
366
367     # Limita? do intervalo
368     li      $t1, 32              # Define o limite superior
369     rem     $t0, $t0, $t1        # $t0 <- $t0 % 32
370
371     sw      $t0, 0($s2)          # seed <- $t0
372     move    $v0, $t0            # o retorno é $t0
373
374     # Ep?o
375     lw      $ra, 0($sp)          # Restauramos o endereço de retorno
376     addiu   $sp, $sp, 4          # Restauramos a pilha
377     jr      $ra                  # Retornamos ao procedimento chamador
378
379
380 #####

```

Figura 9 - coordenada\_aleatoria

Além das verificações de derrota ocasionadas pela cabeça da cobra colidir com algum lado da tela, também foi necessário criar um procedimento **verificar\_derrota** que verifica se a cobra colidiu com o próprio corpo. Para verificar essa situação basta ver se a cabeça ocupa o mesmo espaço que outra parte do corpo. Primeiramente pegamos o comprimento da cobra (\$s7) e colocamos em uma variável temporária com esse valor decrementado em 1 (para não contar a cabeça). Caso a cobra tenha tamanho 1 (sem corpo) saímos do procedimento. Calculamos o deslocamento total com sll para acessar a coordenada final x (\$s0) e y (\$s1) da cobra, possibilitando acesso a qualquer parte da cobra. Dessa maneira, iniciamos um laço que compara a cabeça da cobra na posição x (\$s0) e y(\$s1) com a posição \$t0 (posição x da célula

atual) e \$t1 (posição y da célula atual). Caso \$s0 for igual a \$t0, assim como \$s1 for igual a \$t1, o programa faz o comando j finit para encerrar o jogo. Caso uma das coordenadas for diferente, os ponteiros \$t0 e \$t1 são somados por -4 para ir para próxima posição deles (posição de célula anterior). No final, quando \$t0 chegar ao valor do \$s0, finaliza a função.

```

448 verificar_derrota:
449     # Ep?o
450     addiu $sp, $sp, -4      # Ajustamos a pilha
451     sw $ra, 0($sp)         # Armazenamos o endereço de retorno na pilha
452     # Corpo do procedimento
453     #acha as coordenadas finais do procedimento
454     addi $t0, $s7, -1 # $t0 <- Comprimento - 1
455
456     beqz $t0, derExit
457     sll $t0, $t0, 2 # ($Comprimento - 1)*4
458     add $t1, $t0, $s1 # Y+($Comprimento - 1)*4
459     add $t0, $t0, $s0 # X+($Comprimento - 1)*4
460     lw $t4, 0($s0) # $t4 <- X[0]
461     lw $t5, 0($s1) # $t5 <- Y[0]
462     #Laço para verificar a cobra inteira
463     verFor:
464         #Verifica a posição
465         lw $t3, 0($t0) # $t3 <- X[Comprimento - i - 1]
466         #Se X[0] != X[Comprimento - i - 1] então vá para verElse
467         bne $t4, $t3, verElse
468         lw $t3, 0($t1) # $t3 <- Y[Comprimento - i - 1]
469         #Se Y[0] == Y[Comprimento - i - 1] então vá para verElse
470         bne $t5, $t3, verElse
471         #Se ambas coordenadas forem iguais o jogo acaba
472         j finit
473
474         #Se pelo menos uma coordenada for diferente continua o laço
475     verElse:
476         #Retorna uma posição
477         #i = i-1
478         addi $t0, $t0, -4
479         addi $t1, $t1, -4
480
481         #Quando i==0 a função acaba
482         bne $t0, $s0, verFor
483     derExit:
484     # Ep?o
485     lw $ra, 0($sp)         # Restauramos o endereço de retorno
486     addiu $sp, $sp, 4      # Restauramos a pilha
487     jr $ra                 # Retornamos ao procedimento chamador

```

Figura 10 - verificar\_derrota

## 5) CONCLUSÃO

A realização desse projeto serviu de maneira valiosa para a aprendizagem de novos conhecimentos sobre o assembly MIPS, ajudando nosso grupo a colocar em prática nossos conhecimentos adquiridos. A possibilidade de manipular a parte gráfica com o Bitmap Display e outras ferramentas do MARS 4.5 tornou o projeto ainda mais interessante e possibilitou, com sucesso, a criação do jogo Snake. O aprendizado gerado por esse projeto resultou no aprimoramento geral do grupo, podendo ser desenvolvidas novas aplicações que explore essa ferramenta proposta na disciplina.

## **REFERÊNCIAS BIBLIOGRÁFICAS**

- [1] **PATTERSON, David A.; HENNESSY, John L.** Organização e projeto de computadores: a interface hardware/software. 5. ed. Rio de Janeiro: Elsevier, 2017.
- [2] **BARATTO, Giovani.** Notas de aula da disciplina de Organização de Computadores. Santa Maria: UFSM, 2024. Notas de aula.