UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
DEPARTAMENTO DE SISTEMAS DE INFORMAÇÃO

Matheus Delazeri Souza e Wederson Fagundes

SNAKE GAME:
UMA IMPLEMENTAÇÃO EM ASSEMBLY MIPS

Santa Maria, RS

SUMÁRIO

1 INTRODUÇÃO

O presente trabalho apresenta o desenvolvimento do clássico "Jogo da Cobra"(*Snake Game*) utilizando a linguagem Assembly MIPS, destacando os desafios e peculiaridades da programação em baixo nível. O *Snake Game*, originalmente concebido por Gremlin Industries e lançado em 1976 como parte do jogo arcade Blockade, tornou-se amplamente popular em várias plataformas ao longo das décadas. Sua popularidade foi impulsionada especialmente nos anos 1990, quando versões do jogo passaram a ser incorporadas em celulares Nokia, consolidando-se como um marco na história dos videogames.

O jogo possui uma mecânica simples, porém viciante: o jogador controla uma cobra que se movimenta em um tabuleiro, coletando itens (normalmente representados como frutas ou blocos) que aumentam o tamanho da cobra. O desafio reside em evitar colisões com as paredes do tabuleiro ou com o próprio corpo da cobra, o que resulta no término da partida. Apesar de sua simplicidade, o *Snake Game* é um excelente exemplo de como conceitos básicos de lógica, controle de fluxo e manipulação de dados podem ser aplicados para criar experiências interativas envolventes.

Neste trabalho, exploramos a implementação do *Snake Game* em MIPS Assembly, detalhando as estratégias para manipulação de memória, controle do fluxo de execução e interação com o usuário, a fim de recriar a funcionalidade clássica do jogo em um ambiente de programação de baixo nível.

1.1 COMO JOGAR

1.1.1 Requisitos

Antes de iniciar, certifique-se de que possui os seguintes itens configurados em seu ambiente de trabalho:

- **Simulador MIPS Mars**: O software Mars deve estar instalado em seu computador. Ele pode ser baixado em https://dpetersanderson.github.io/.
- **Arquivos do Jogo**: Faça o download dos arquivos `main.asm` e `display_bitmap.asm`, disponíveis no repositório: https://github.com/matheus-delazeri/snake.

1.1.2 Etapas para Execução

1. Abra o Arquivo no Mars: Inicie o software Mars e abra o arquivo `main.asm` utilizando o menu de navegação do simulador.

2. Configure o *Bitmap Display* No menu do Mars, selecione *Tools* > *Bitmap Display* e configure o display gráfico de acordo com as especificações abaixo:

```
Unit Width in pixels: 8
Unit Height in pixels: 8
Display Width in pixels: 512
Display Height in pixels: 512
```

Após realizar as configurações, clique em *Connect to MIPS* para conectar o *Bitmap Display* ao programa.

3. Configure o Simulador de Teclado: No menu do Mars, acesse *Tools* > *Keyboard and Display MMIO Simulator*. Em seguida, pressione o botão *Connect to MIPS* para associar o teclado ao programa.

4. Inicie o Jogo: Com as ferramentas configuradas, compile e execute o programa utilizando os botões *Assemble* e *Run*, disponíveis na interface do Mars.

1.1.3 Controles do Jogo

A movimentação da cobra no jogo é controlada pelas teclas do teclado. As funções de cada tecla são descritas a seguir:

- **W**: Mover para cima.
- **S**: Mover para baixo.
- **A**: Mover para a esquerda.
- **D**: Mover para a direita.
- **Q**: Sair do jogo.

1.1.4 Considerações

Para garantir o funcionamento adequado do jogo, certifique-se de que o *Bitmap Display* e o Simulador de Teclado estão conectados ao Mars antes de iniciar a execução.

Caso a partida termine devido a uma colisão (com as bordas ou com o próprio corpo da cobra), será necessário repetir a etapa de execução para reiniciar o jogo.

2 OBJETIVOS

Este trabalho tem como objetivo principal implementar o clássico *Snake Game* utilizando a linguagem Assembly MIPS, explorando as particularidades da programação em baixo nível e implementando técnicas otimizadas para o fluxo do jogo. Além disso, busca-se desenvolver uma compreensão prática de conceitos como gerenciamento de memória, uso de registradores e controle de fluxo em um ambiente restrito.

3 REVISÃO BIBLIOGRÁFICA

Para o desenvolvimento do *Snake Game* em Assembly MIPS, foram analisadas diversas implementações *open source*, com o objetivo de identificar boas práticas e estratégias eficientes. Observou-se que, em grande parte, as implementações seguem estruturas similares, utilizando abordagens padronizadas para o controle do corpo da cobra e a detecção de colisões.

Entretanto, algumas técnicas otimizadas destacaram-se nesse contexto. Um exemplo é o jogo desenvolvido por Medeiros (2018), que introduziu a prática de armazenar a direção atual de um segmento da cobra diretamente no endereço correspondente da memória. Essa abordagem, detalhada posteriormente na seção de Implementação, foi incorporada ao projeto como uma solução inovadora para simplificar o gerenciamento da movimentação da cobra.

4 METODOLOGIA

4.1 PLANEJAMENTO DO JOGO

O desenvolvimento do projeto começou com a definição clara das regras básicas do jogo. Esta etapa foi essencial para orientar a implementação e evitar ambiguidades durante o processo de desenvolvimento. Também foi realizada uma revisão bibliográfica preliminar, onde implementações semelhantes foram analisadas para fundamentar o planejamento técnico.

4.2 CONFIGURAÇÃO DO AMBIENTE

Optou-se pelo uso do simulador MIPS Mars, que inclui ferramentas como Bitmap Display e Keyboard Simulator. Essa escolha baseou-se na praticidade do ambiente para emular gráficos e capturar entradas do teclado, recursos fundamentais para o funcionamento do jogo.

4.3 IMPLEMENTAÇÃO DO CÓDIGO

Com o planejamento e o ambiente definidos, deu-se início ao desenvolvimento do código. O processo foi realizado de forma iterativa, com testes frequentes para validar o funcionamento dos principais componentes, como a movimentação da cobra, a geração de maçãs e a detecção de colisões. As soluções empregadas foram ajustadas conforme os resultados obtidos durante os experimentos.

5 IMPLEMENTAÇÃO

A implementação foi planejada de forma a aproveitar ao máximo os recursos limitados da linguagem, equilibrando simplicidade e eficiência. Nesta seção, destacamos dois aspectos técnicos fundamentais para o funcionamento do jogo: o armazenamento da direção dos segmentos da cobra diretamente nos endereços do *bitmap* e o uso de operações *bitwise* para detecção de colisões. Esses elementos foram essenciais para garantir um desempenho adequado e simplificar a lógica de movimentação e colisão.

5.1 ARMAZENAMENTO DE DIREÇÃO NOS ENDEREÇOS DO *BITMAP*

Uma das técnicas utilizadas foi a codificação da direção de movimento dos segmentos da cobra diretamente nos valores utilizados para representar os segmentos no *bitmap*, como definido no projeto de Medeiros (2018). Isso foi realizado através de *flags* de direção, codificadas junto aos valores de cor, conforme demonstrado no trecho de código a seguir:

```
# Direction Flags (encoded with color value)
# 00 = Right, 01 = Down, 02 = Left, 03 = Up
.eqv SNAKE_RIGHT, 0x0000FF00
.eqv SNAKE_DOWN,  0x0100FF00
.eqv SNAKE_LEFT,  0x0200FF00
.eqv SNAKE_UP,    0x0300FF00
```

Cada segmento da cobra possui um valor associado que representa sua direção atual. Por exemplo, um segmento com o valor *0x0100FF00* está se movendo para baixo. Esse valor é armazenado no mesmo endereço de memória utilizado para desenhar o segmento no *bitmap*. Dessa forma, não é necessário manter uma estrutura de dados separada para registrar as direções dos segmentos e, ao remover a cauda antiga da cobra ao locomover-se, a nova posição da cauda pode ser facilmente atualizada.

5.2 DETECÇÃO DE COLISÃO COM OPERAÇÕES *BITWISE*

Outro elemento crítico do jogo é a detecção de colisões, tanto com as bordas da tela quanto com o próprio corpo da cobra. A implementação utiliza operações *bitwise* para isolar e analisar componentes dos endereços de memória. A seguir, está um exemplo da lógica utilizada para detecção de colisões horizontais e verticais:

```
.eqv DISPLAY_MEMORY_BASE 0x10010000

check_wall_collision:
    move $t0, $a0 # Snake head
    lw $t1, 0($s7) # Snake current direction

    # Check horizontal boundaries (X)
    andi $t2, $t0, 0XFF # Isolate the last 8 bits
    srl $t3, $t2, 2     # Convert to screen coordinate
    beq $t1, SNAKE_LEFT, check_left_wall
    beq $t1, SNAKE_RIGHT, check_right_wall

    # Check vertical boundaries (Y)
    sub $t2, $t0, $t2 # Keep only Y changes
    subi $t2, $t2, DISPLAY_MEMORY_BASE
    bltz $t2, game_over # Check if out of top boundary
    srl $t3, $t2, 8     # Convert to screen coordinate
    bge $t3, SCREEN_HEIGHT, game_over
```

A técnica se baseia na estrutura do espaço de memória utilizada pelo *display*, que pode ser verificada abaixo:

```
.eqv UNIT_WIDTH     8 # width unit in pixels
.eqv UNIT_HEIGHT    8 # height unit in pixels
.eqv DISPLAY_WIDTH  512 # width of the bitmap display
.eqv DISPLAY_HEIGHT 512 # height of the bitmap display
.eqv SCREEN_WIDTH   64 # DISPLAY_WIDTH/UNIT-WIDTH
.eqv SCREEN_HEIGHT  64 # DISPLAY_HEIGHT/UNIT_HEIGHT
```

Como o jogo foi desenvolvido para este tamanho fixo de *display*, podemos extrair dos endereços as posições relativas de **X** e **Y**, presentes nos *bits* menos significativos. Considerando que cada nova linha no *display* ocorre de 256 em 256 bits:

```
Y_PACE = SCREEN_WIDTH * WORD_SIZE
256 = 64 * 4
```

Podemos definir que os 8 primeiros bits ($2^8 = 256$) do endereço são preenchidos com a variação **X** relativa ao *display* do jogo. Tendo isso em vista, basta isolarmos os 8 últimos *bits* do endereço, utilizando:

```
 andi $t2, $t0, 0XFF # Isolate the last 8 bits
```

Para identificarmos se a posição atual de **X** ultrapassa a largura máxima, especificada por **SCREEN_WIDTH**, o que causaria uma colisão horizontal. Algo semelhante é feito para as colisões verticais, onde decrementa-se de **DISPLAY_MEMORY_BASE** o endereço atual, sem os bits de variação **X**, e verifica-se se ultrapassam os limites impostos.

Por exemplo, considere o endereço de memória **0x10010208**: seus 8 *bits* menos significativos (**0x08**) representam a posição horizontal do segmento (**X** = 8 pixels ou **X** = 1 unidade) enquanto os *bits* seguintes (**0x0200**), já sem os *bits* de posição horizontal, indicam a posição vertical (**Y** = 512 pixels ou **Y** = 64 unidades).

6 RESULTADOS

O projeto resultou em um jogo funcional, implementado inteiramente na linguagem de montagem MIPS, que é capaz de reproduzir uma versão simples e eficiente do clássico jogo da cobra. O jogo atinge os principais objetivos estabelecidos, como a movimentação fluida da cobra, a interação responsiva com os controles e a detecção precisa de colisões. Essas funcionalidades foram testadas em um ambiente simulado utilizando o MIPS Mars, e o comportamento do jogo se mostrou estável em todos os cenários propostos.

Além disso, as técnicas de codificação otimizadas para o espaço de memória do *display* foram implementadas com sucesso, como o armazenamento da direção nos bits de cor do bitmap e o uso de operações *bitwise* para a detecção de colisões. Tais abordagens não apenas simplificaram o código, mas também reduziram o consumo de memória e processamento, o que é crítico em sistemas de arquitetura reduzida. Por fim, o desempenho do jogo foi considerado satisfatório, sem quedas perceptíveis de performance mesmo em situações de maior complexidade, como a cobra atingindo tamanhos maiores.

7 DISCUSSÃO

A implementação do projeto revelou tanto desafios quanto aprendizados significativos. Uma das principais dificuldades foi adaptar a lógica do jogo para uma arquitetura de baixo nível, onde os recursos de abstração são limitados e cada operação deve ser explicitamente detalhada. No entanto, esse desafio foi enfrentado com estratégias que maximizam a eficiência, como a escolha de representar a direção da cobra diretamente nos valores dos endereços de memória. Essa abordagem não apenas simplificou o controle da movimentação, mas também reduziu a necessidade de estruturas de dados adicionais, demonstrando como soluções criativas podem superar limitações arquiteturais.

8 CONCLUSÕES

O desenvolvimento deste projeto permitiu explorar de maneira prática as possibilidades e desafios da linguagem de montagem MIPS, bem como as particularidades de arquiteturas baseadas em controle direto de memória. O jogo final alcançou os objetivos propostos, ao mesmo tempo em que incorporou técnicas otimizadas para gestão de memória e processamento, demonstrando ser um exemplo funcional de programação eficiente em MIPS.

Do ponto de vista educacional, o projeto contribuiu para um aprendizado aprofundado sobre manipulação de memória, operações *bitwise* e a integração entre software e hardware em arquiteturas simples. Além disso, abriu caminhos para melhorias e extensões. Entre as perspectivas futuras, destaca-se a possibilidade de implementar níveis de dificuldade adicionais, incluindo variações na velocidade da cobra. Outra expansão seria a introdução de modos de jogo, como um modo competitivo para dois jogadores ou variações nos tipos de alimentos coletados pela cobra, cada um com efeitos diferentes no jogo.

REFERÊNCIAS

MEDEIROS, J. **MIPSnake**. 2018. Acesso em 20 nov. 2024. Disponível em: <https://github-.com/JoMedeiros/MIPSnake>.