# Projeto Aplicativo - Snake -

Gabriel Fernandes Carvalho - 17/0142698[*] Marina Pinho Garcia - 17/0110702[†]
Rafael Fernandes Barbosa - 17/0163857[‡] João Vítor Arantes Cabral - 17/0126048[§]

UnB - Organização e Arquitetura de Computadores – 2020/0 - Prof. Marcelo A. Marotta

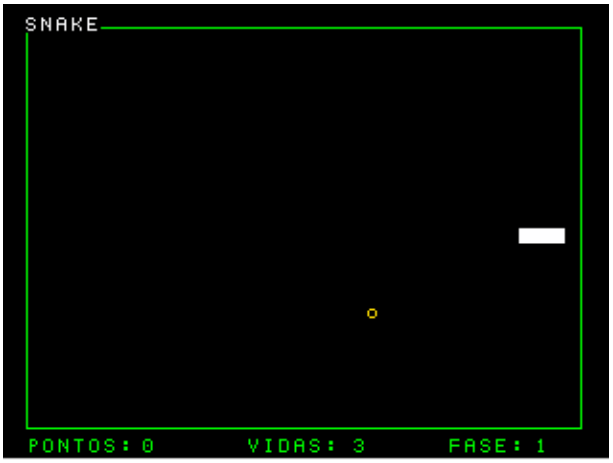


Figura 1: Início do jogo Snake

**RESUMO**

Este trabalho consiste no desenvolvimento de uma releitura do jogo SNAKE, lançado pela primeira vez em 1976. O jogo foi implementado usando a linguagem assembly RISC-V e executado no processador Pipeline RISC-V ISA RV32IMF na placa de desenvolvimento FPGA DE1-SoC.

**Palavras-chave:** Snake, RISC-V, processador, FPGA, RARS.

## 1 INTRODUÇÃO

Nesse relatório serão apresentadas imagens, exemplos, trechos de código e explicações referentes ao desenvolvimento do jogo SNAKE e sua implementação na FPGA DE1-SoC. Além disso, os resultados finais obtidos serão exibidos e comentados para uma melhor avaliação geral do projeto.

O jogo SNAKE consiste em uma figura divida em blocos (geralmente representada com gráficos simples) controlada pelo jogador que representaria uma cobra, daí o nome do jogo; o objetivo do game é conseguir a maior quantidade possível de pontos ao mesmo tempo que a cobra cresce e a dificuldade do jogo aumenta (na maioria dos casos pelo aumento da velocidade de execução). Para obter tais pontos, o jogador deve alcançar outros objetos os quais representam o alimento da cobra que, ao obtê-los, aumenta de tamanho.

Já existem milhares de iterações do jogo que variam de acordo com a criatividade de quem produziu, entretanto o modelo mais comum é aquele no qual se perde uma vida quando a cobra ou atinge a parede ou a si mesma. O SNAKE normalmente começa com 3 vidas e o jogador mantém sua pontuação acumulada até que perca todas suas vidas.

Um dos desafios que diferenciam esse projeto dos demais envolvendo o SNAKE, é o fato de que neste caso o jogo deve ser implementado em uma DE1-SoC, o que gera alterações no código do programa e problemas de compatibilidade que devem ser resolvidos. Outro aspecto notável é que o jogo será executado por um processador RISC-V instalado na FPGA, o que limita o processamento e a quantidade de memória que o projeto pode utilizar.

Por último vale citar que uma versão customizada da IDE RARS foi utilizada para escrever, montar e testar o código, assim como gerar o código de máquina final. Para passar o código para a placa, usou-se o software Quartus Prime 18.0, no qual foi compilado o processador RISC-V pipeline alterando sua rotina de execução normal para executar o jogo.

## 2 FUNDAMENTAÇÃO TEÓRICA E TÉCNICA (TRABALHOS CORRELATOS)

O jogo desenvolvido foi dividido em etapas para facilitar a implementação de todos os seus elementos, as etapas citadas serão descritas abaixo:

- Interface com o usuário:
  - Menu Inicial: nesta tela o usuário terá as opções de iniciar o jogo, fechar o programa e voltar para o menu. Foi a primeira tela a ser implementada no trabalho;
  - Game Over: nesta tela o usuário terá a opção de voltar ao menu inicial.
- Elementos do Jogo:
  - Snake: controlada pelo usuário, utilizando as teclas w (cima), a (esquerda), s (baixo), d (direita). Sempre ini-

[*]e-mail: gabrielfc2102@hotmail.com
[†]e-mail: jvarantescabraltrab@gmail.com
[‡]e-mail: rafaelfbarbosa7@gmail.com
[§]e-mail: NinaPGarcia@gmail.com

cia com tamanho igual a 3 e na mesma posição, e a cada comida que pegar seu tamanho aumenta em 1;

  - Comida: gerada de forma aleatória dentro da delimitação das paredes e de forma que não apareça no mesmo lugar em que está a snake;
  - Paredes: as paredes formam um retângulo no qual se a snake encostar o usuário perde uma vida;
  - Vidas: o usuário começa o jogo com 3 vidas, toda vez que a snake atingir uma parede ou ela mesma, o usuário perde uma vida e a snake volta ao seu tamanho e posição inicial e o jogo volta para a primeira fase (a pontuação não reinicia), quando as vidas do usuário acabarem o jogo vai para a tela de Game Over.

- Ambiente
  - A tela durante o jogo possui um retângulo (as paredes) e fora dele, na parte inferior da tela estão as informações do jogo: pontuação, vidas e fase. Durante o jogo, o usuário pode utilizar a tecla "0" para voltar ao Menu Inicial.

## 3 METODOLOGIA

A ideia inicial do jogo era se basear em uma máquina de estados para definir as interações do usuário com a execução do programa, assim como a interferência da mudança constante de posição da cobra nos atributos do jogo. Pensando nisso, fez-se um esboço inicial de como seria essa máquina de estados, cuja tradução em um fluxograma pode ser observada abaixo:

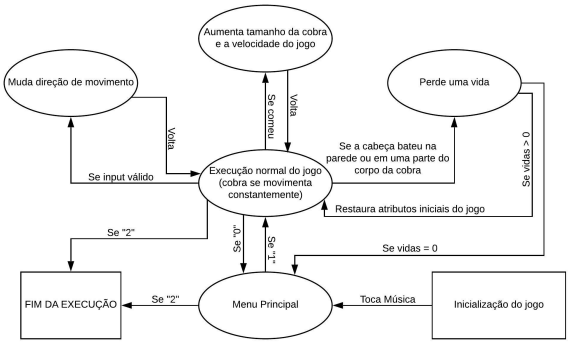


Figura 2: Fluxograma da máquna de estados do jogo

Ao criar a lógica do programa, a principal preocupação na implementação dos procedimentos em assembly RISC-V foi a otimização da execução do jogo, uma vez que à medida que a cobra cresce, mais instruções são executadas e consequentemente mais lenta fica a atualização de atributos e display. Diante disso, implementou-se algumas soluções para resolver este problema, dentre elas:

- Utilização eficiente da memória (half word e byte substituindo word quando possível);
- Imprimir somente a cabeça da cobra a cada loop e apagar o rabo, dessa forma não importa o tamanho da cobra, o mesmo número de ecalls será utilizado, diminuindo substancialmente o número de instruções executadas por loop;
- Utilizar o mínimo de ecalls possível e substituir ao máximo o uso de procedimentos constantemente por lógicas mais simples.

Vale ressaltar que tais preocupações só foram evidentes quando percebeu-se que o jogo apresentava quedas de FPS à medida que a pontuação do jogador aumentava e até travamentos do jogo (os quais requeriam o uso do gerenciador de tarefas para encerrar o RARS) que ocorriam a partir dos 300 pontos.

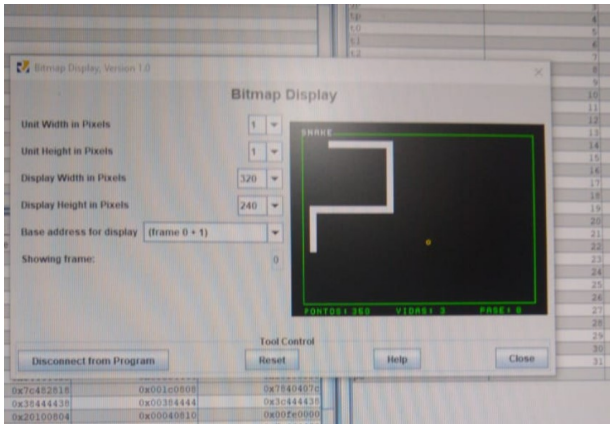


Figura 3: Exemplo de travamento em teste não otimizado do jogo, destaque para a pontuação de 350 que aparece na tela

Sobre a pontuação, decidiu-se que tal atributo não iria aumentar de acordo com o tempo de sobrevivência mas sim proporcional ao número de alimentos consumidos pela cobra, dessa forma ao obter um alimento o jogador consegue 10 pontos.

Outro aspecto importante foi a decisão de manter a pontuação do jogador mesmo após perder uma vida, o que permite ao usuário comparar seus recordes anteriores e sempre buscar uma pontuação máxima maior; entretanto, não se implementou um recurso de pausa devido à simplicidade do jogo snake que geralmente tem sessões de jogo rápidas, no final das contas isso acaba requisitando uma atenção maior do player enquanto estiver jogando.

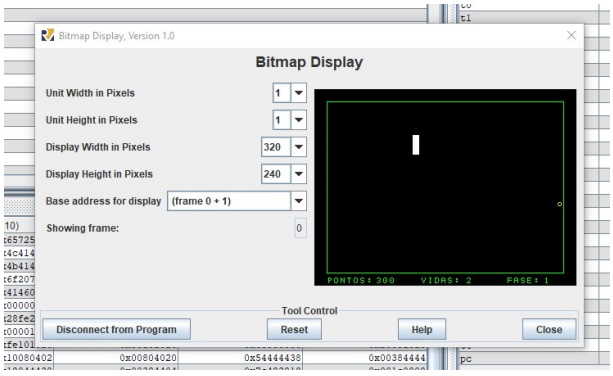


Figura 4: Pontuação se mantém mesmo após perder uma vida

Seguindo nas decisões criativas por trás do jogo, optou-se por disponibilizar 3 vidas ao jogador, visto que é o valor mais encontrado em jogos clássicos inclusive o snake. Ao perder uma vida o jogo reseta seus atributos para os valores iniciais (mantendo-se somente a pontuação e subtraindo uma vida) e o jogador precisa obter os alimentos para aumentar o tamanho da cobra e a velocidade do game novamente.

Como era de se esperar, ao perder 3 vidas uma tela de game over aparece e todos os atributos do jogo são resetados e o jogador retorna ao menu principal após apertar "0", onde deve escolher se deseja continuar jogando ou encerrar a execução do aplicativo.

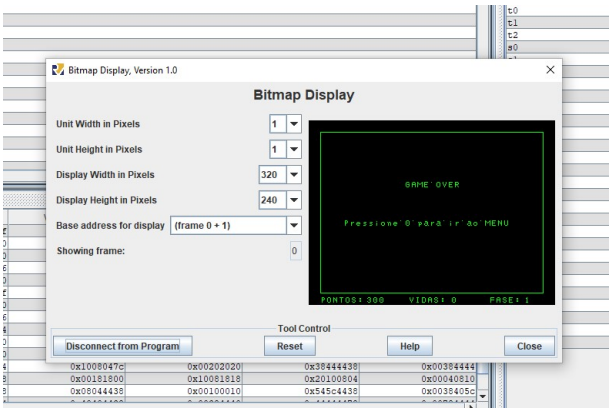


Figura 5: Tela de Game Over com 0 vidas

O método para obtenção de dados do usuário escolhido para o jogo foi o polling, no qual a cada ciclo de execução normal do programa confere-se se uma tecla foi pressionada e tal input é tratado para ver se trata-se de uma tecla válida; caso afirmativo, é mudada a direção do movimento da cobra ou a opção correspondente do menu é executada.

```
# Pegar dado do teclado e armazenar em t2
        li t1,0xFF200000                # carrega o endereco de controle do KDMMIO
        lw t0,0(t1)                     # Le bit de Controle Teclado
        andi t0,t0,0x0001               # mascara o bit menos significativo


# Se nao ha tecla pressionada, realiza estado anterior
        beq t0,zero, estado_anterior    # Se nao ha tecla pressionada entao volta para o loop

        lh t2,4(t1)                     # Armazena proxima tecla em t2
        mv s10, t2
```

Figura 6: Polling com máscara do bit menos significativo

```
# Tratamento para cada tecla pressionada
        li t0,48                        # t0 = 0 -> sai
        beq  t2,t0, MENU_PRINCIPAL

        la t0, teclado_anterior         #pega a ultima tecla direcional pressionada
        lb t6, 0(t0)

testa_esquerda:
        li t0, 100
        beq t6, t0, testa_direita       # se tava indo pra direita nem compara
        li t0, 97                       # t0 = a
        beq t2,t0, esquerda

testa_direita:
        li t0, 97
        beq t6, t0, testa_baixo         #se tava indo pra esquerda nem compara
        li t0,100                       # t0 = d
        beq  t2,t0, direita

testa_baixo:
        li t0, 119
        beq t6, t0, testa_cima          #se tava indo pra baixo nem compara
        li t0,115                       # t0 = s
        beq  t2,t0, baixo

testa_cima:
        li t0, 115
        beq t6, t0, INVALIDO            #se tava indo pra baixo nem compara
        li t0,119                       # t0 = w
        beq  t2,t0, cima

INVALIDO:
        mv t2, t6
        li t0, 100
        beq t6, t0, direita
        li t0, 97
        beq t6, t0, esquerda
        li t0, 119
        beq t6, t0, cima
        li t0, 115
        beq t6, t0, baixo
```

Figura 7: Tratamento da tecla pressionada para testar se é válida para o movimento atual da cobra

O tratamento da tecla pressionada pelo usuário é necessário pois é preciso impedir que a cobra tente voltar pelo caminho que ela percorreu, e ao mesmo tempo, teclas indesejadas pressionadas pelo usuário devem ser ignoradas. Neste caso a execução normal do movimento deve ser mantida, como mostra a imagem acima.

Para imprimir a cobra na tela, optou-se por imprimir um caractere branco com um fundo branco, dessa forma o código ficou mais simples mas ao mesmo tempo o desempenho ficou um pouco abaixo do máximo possível com a lógica implementada. Entretanto, como o número de impressões se mantém o mesmo sem importar o tamanho da cobra, o impacto gerado não foi significativo.

```
# Imprime cabeca da cobra
        li a7, 111
        li a3, 0xffffffff
        li a4, 0
imprime_snake:
        li a0, 111
        lh a1, 0(t1)                    # Pega x
        lh a2, 2(t1)                    # Pega y

        ecall
```

Figura 8: Fragmento do código que imprime a cabeça da cobra a cada loop

```
######################################################################## InicioAPAGA_ULTIMO
APAGA_ULTIMO:

        addi t0, s11, -1                # Tamanho - 1 -> Usaremos para acessar o ultimo elemento
        slli t0, t0, 2                  # Obtendo o quando o percorredor pularÃ¡ na pilha para chegar ao ultimo elemento
        add  t0, t0, sp                 # t0 e o endereco inicial do ultimo elemento

        lh t3, 0(t0)                    # Acessa o elemento de x
        lh t4, 2(t0)                    # Acessa o elemento de y

        li a7, 111
        li a0, 111
        mv a1, t3
        mv a2, t4
        li a3, 0x0
        li a4, 0
        ecall
        ret
```

Figura 9: Fragmento do código que apaga a última posição da cobra a cada loop

Uma outra abordagem para essa situação seria criar um novo procedimento que altera o valor no endereço de memória dos bits que aparecem na tela especificamente para a cobra, o que otimizaria mais ainda a execução do programa; mas isso poderia aumentar a complexidade do código e poderia custar muito tempo para implementar então decidiu-se pela solução mais trivial.

Neste projeto também foi implementado efeitos sonoros durante a execução do programa com o uso do Syscall do RARS ”MidiOut”, responsável por emitir os sons. Em nosso jogo foi implementado quatro diferentes sons, os quais depende da situação em que o jogo se encontra:

- Som ao inicializar o jogo:

```
98  # Musica inical
99          li a0,50            # define a nota
100         li a1,1000          # define a duraï¿½ï¿½o da nota em ms
101         li a2,1             # define o instrumento
102         li a3,127           # define o volume
103         li a7,33            # define o syscall
104         ecall
105
106         li a0,55            # define a nota
107         li a1,1000          # define a duraï¿½ï¿½o da nota em ms
108         li a2,1             # define o instrumento
109         li a3,127           # define o volume
110         li a7,33            # define o syscall
111         ecall
112
113         li a0,50            # define a nota
114         li a1,1000          # define a duraï¿½ï¿½o da nota em ms
115         li a2,1             # define o instrumento
116         li a3,127           # define o volume
117         li a7,33            # define o syscall
118         ecall
119
120         li a0,45            # define a nota
121         li a1,1500          # define a duraï¿½ï¿½o da nota em ms
122         li a2,1             # define o instrumento
123         li a3,127           # define o volume
124         li a7,33            # define o syscall
125         ecall
```

Figura 10: Fragmento do código que toca a música inicial

- Som quando a cobra come

```
339 # Faz barulho quando come
340         li a7, 31
341         li a0, 70                   # Nota
342         li a1, 150                  # Duracao
343         li a2, 80                   # Instrumeto
344         li a3, 127                  # Volume
345         ecall
```

Figura 11: Fragmento do código que faz emitir som quando a cobra come

- Som quando a cobra morre

```
309         li a0,30            # define a nota
310         li a1,1200          # define a duraï¿½ï¿½o da nota em ms
311         li a2,1             # define o instrumento
312         li a3,127           # define o volume
313         li a7,33            # define o syscall
314         ecall
```

Figura 12: Fragmento do código que emite som quando a cobre morre

- Som quando o jogador perde as 3 vidas

```
266         li a0, 40                   # define a nota
267         li a1,1000          # define a duraï¿½ï¿½o da nota em ms
268         li a2,1             # define o instrumento
269         li a3,127           # define o volume
270         li a7,33            # define o syscall
271         ecall
272
273         li a0, 30                   # define a nota
274         li a1,2000          # define a duraï¿½ï¿½o da nota em ms
275         li a2,1             # define o instrumento
276         li a3,127           # define o volume
277         li a7,33            # define o syscall
278         ecall
```

Figura 13: Fragmento do código que toca música quando o jogador perde as 3 vidas

Vale ressaltar que primeiramente os sons foram implementados para o RARS, funcionando como o esperado, porém ao executar o programa na FPGA houve uma mudança na sonoridade deles e, portanto, foi necessário alterá-los. Descobrimos que isso se deve ao fato de o RARS aceitar diferentes instrumentos enquanto a FPGA não possui esta funcionalidade, alterando apenas as notas.

## 4 Resultados Obtidos

O funcionamento do jogo foi como o esperado tanto ao ser executado no simulador RARS com o auxílio do Bitmap Display quanto no microprocessador:

- Tela inicial: A Figura 14 mostra a primeira tela do jogo ao executarmos o programa.

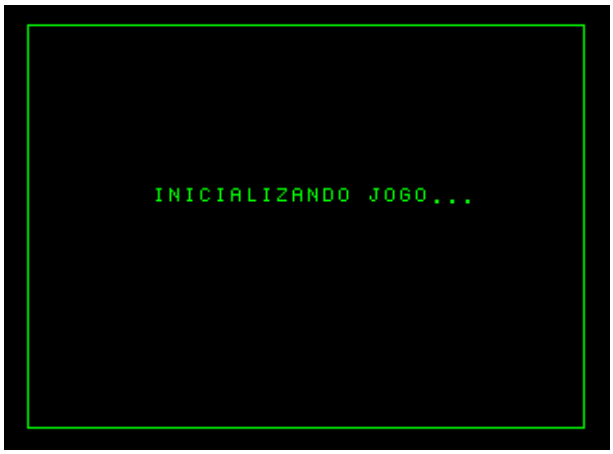


Figura 14: Tela Inicial

- Interface com o usuário: as telas de interface com o usuário são o Menu Inicial do jogo e a tela de Game Over, representadas abaixo:

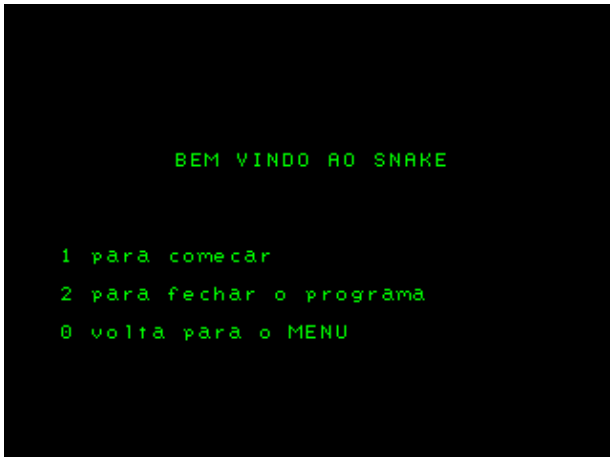


Figura 15: Menu Inicial


Figura 16: Tela de Game Over

- Jogo: a Figura 17 mostra a tela do jogo em si, a snake (controlada pelo usuário) o ambiente, delimitado pelas linhas verdes, a comida e as informações (pontos, vidas e fase).

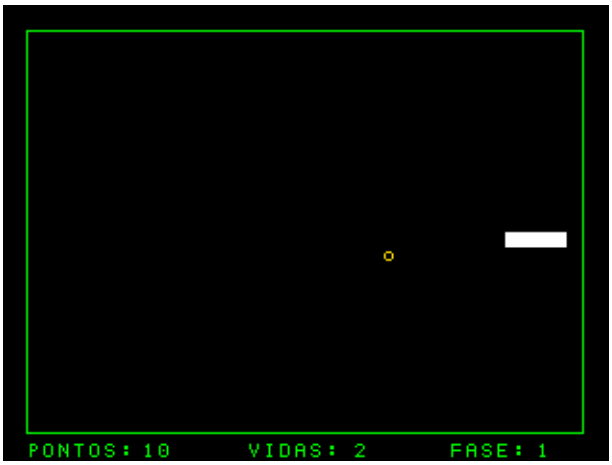


Figura 17: Tela de um momento do jogo

- Execução no microprocessador: do arquivo "snake.asm", foram gerados os dois arquivos ".mif " (data e text) que, utilizando o programa Quartus Prime v18.0, o jogo foi executado no microprocessador. Foi utilizado um teclado PS2 para a movimentação da snake e o funcionamento do jogo foi o esperado.

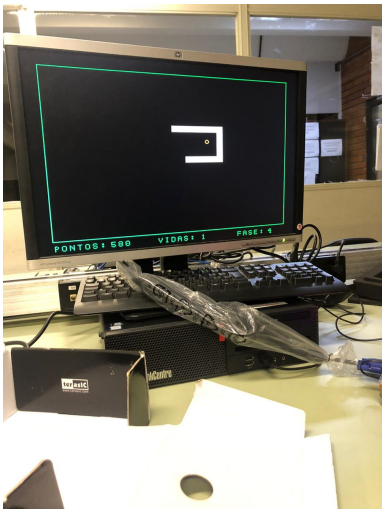

Figura 18: Execução no microprocessador

## 5 Conclusões e Trabalhos futuros

Este trabalho foi importante para a familiarização e aprofundamento da linguagem Assembly do RISC-V e para melhor compreensão do funcionamento dos processadores. Além disso, no desenvolvimento do jogo vimos que era necessária preocupação com o uso da memória, processamento e interrupções, para que a performance do jogo não fosse prejudicada. Por fim, tivemos que fazer alterações para garantir a compatibilidade com as funções disponíveis no processador pipeline usado.

## Referências Bibliográficas

[1] P. D. Hennessy, John L. *Computer Organization and Design: the hardware/software interface*. Morgan Kaufmann Publishers, fifth edition, 2014.

[2] M. V. Lamar. Slides: Lab1-rars-io. Aprender UnB, 2020.

[3] P. D. Waterman, Andrew. *RISC-V Um guia prático*. Strawberry Canyon, primeira edition, 2017.