Projeto Aplicativo: SNAKE

Gabriel Guimarães Almeida de Castro[1]* Lucas Emanuel Silva Ferreira [2]† Rodrigo Alves de Araujo [3] ‡
Stephanie Cardoso Guimarães[4] §

[1]Universidade de Brasília, Departamento de Ciência da Computação -CIC
Cód: 116394 Organização e Arquitetura de Computadores Turma: A - 0/2020
Professores: Marcelo A. Marotta e Marcus Vinucius Lamar

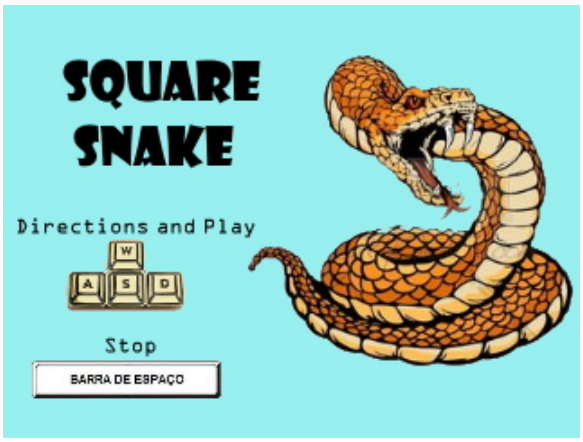

Figure 1: Jogo Square Snake!

ABSTRACT

Desenvolver o clássico jogo Snake. Utilizando o RARS como IDE, linguagem de programação assembly e o microprocessador RISC-V (ISA RV32IMF) implementado no kit de desenvolvimento DE1-Soc. O jogo possui compatibilidade para rodar na FPGA do kit DE1-Soc, utilizando um teclado PS2 para controlar os movimentos da cobra, efeitos sonoros e um monitor para vizualização do jogo. O RARS com a versão SYSTEMv17b.s, tambem possui todas as ferramentas necessárias para rodar e testar o jogo.

Keywords: Snake, Rars, RISC-V, DE1-Soc, Jogo, Teclado PS2
.

1 INTRODUÇÃO

1.1 Square Snake

O *Square Snake*,[3] jogo criado nesse projeto, tenta recriar e renovar o jogo da cobrinha sem perder suas principais caracteristicas, que o tornam um clássico. Um jogo, cujo objetivo é sempre alimentar a cobra, uma vez que esta consegue comer, seu corpo cresce e velocidade de movimentação da cobrinha aumenta. Portanto, a dificuldade do jogo está em continuar fazendo a cobra comer, mesmo quando seu corpo já está bem extenso, cada vez que a cobra come a fruta ou popularmente, a maça, a pontuação *(score)* e a velocidade *(speed)* do jogador é incrementada.

*e-mail: gabriel1997.castro@gmail.com
†e-mail: lucaoemanuel@gmail.com
‡e-mail: rodrygoalvys@gmail.com
§e-mail: stephanieguimaraes7@gmail.com

O nome do jogo, ”Square Snake” (traduzido literalmente como, cobra quadrada), se deve ao fato de que a cobrinha é formada de pequenos blocos quadrados. Esses blocos possuem o tamanho de 8x8 *pixels*, a cobrinha inicia sempre com o tamanho de três blocos dispostos horizontalmente, a maçã possui o tamanho de um único bloco e a cobrinha cresce adiconando-se ao tamanho de seu corpo o equivalente a um bloco sempre que come uma maçã.

Os obstáculos são os limites da área de movimentação da cobra e seu próprio corpo. O ”jogo da cobrinha” ficou bastante popularizado por ser um jogo dos primeiros telefones celulares, por sua simplicidade e desafiar o jogador a querer sempre pontuar mais.

1.2 RISC-V

O Risc-V é uma arquitetura usada para a implementação do jogo, por ser *assembly*, é o nível de programação mais próximo da máquina. Sua utilização se dá por meio de atribuições diretas aos registradores, ou acessos a memória. Ambiente em que será implementado o processador, seja ele, uniciclo, multiciclo ou *pipeline*.

1.3 kit de Desenvolvimento DE1-Soc

O kit consiste em uma placa FPGA, com um processador ARM integrado, memória, entradas de áudio, vídeo e SP2, para teclado ou *mouse*. No entanto, o foco na disciplina, é desenvolvermos um aplicativo para a arquitetura RISC-V, que será implantado na memoria da FPGA, assim como o jogo, através dos arquivos *’.data’* e *’.text’*, escritos em *assembly* no RARS.

1.4 RARS

RARS: ”The RISC-V assembler, Simulator, and Runtime”. Possui a capacidade de montar e simular programas da linguagem *assembly* RISC-V.

Para o caso específico desse projeto é preciso ressaltar que, várias *ecalls* foram acrescentadas ao RARS, na versão denominada

SYSTEMv17b.s, criadas pelo professor orientador da matéria, com o objetivo de otimizar a criação de novos códigos e aplicações, a figura 3, exemplifica um conjunto de ecalls adicionadas.

Foram utilizadas intruções de todos os tipos, R, I, S, B, U e J. Instruções que diferem umas das outras, quanto ao número de bits existentes em cada um dos campos. A figura abaixo exibe a divisão de bits por instrução.

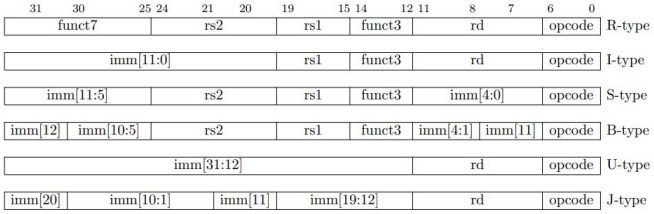

Figure 2: Tipos de instrução

Serviço	a7	Argumentos	Resultados
print integer	101	a0=inteiro a1=coluna a2=linha a3=cores a4=frame	Imprime o número inteiro complemento de 2 a0 na posição (a1,a2) da frame a4 com as cores a3={0...0BBGGGRRRbbgggrrr} sendo BGR fundo e bgr frente
print float	102	fa0=float a1=coluna a2=linha a3=cores a4=frame	Imprime na frame a4 o número float em fa0 na posição (a1,a2) com as cores a3
print string	104	a0=endereço string a1=coluna a2=linha a3=cores a4=frame	Imprime na frame a4 a string terminada em NULL presente no endereço a0 na posição (a1,a2) com as cores a3
print char	111	a0=char (ASCII) a1=coluna a2=linha a3=cores a4=frame	Imprime na frame a4 o caractere a0 (ASCII) na posição (a1,a2) com as cores a3
print int hex	134	a0=inteiro a1=coluna a2=linha a3=cores a4=frame	Imprime na frame a4 em hexadecimal o número em a0 na posição (a1,a2) com as cores a3

Figure 3: Ecalls relacionadas ao bitmap.

Outras ferramentas de fundamental importância para a criação do projeto, foram, o *bitmap* e o *Keyboard*.

O *Bitmap Display* (figura 4) é responsável pelo mapeamento da memória de vídeo VGA, com duas frames de vídeo, 0 e 1, possibilitando a visualização de imagens mais elaboradas. Para o uso do *bitmap*, foram disponibilizadas informações sobre os *pixels*, forma de endereçamento, código de cores e etc.

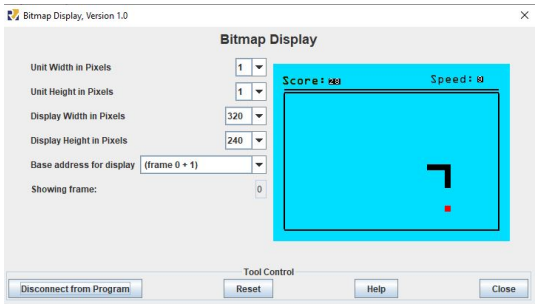

Figure 4: Imagem do jogo no bitmap.

O *keyboard*, ferramenta de *I/O hardware* (figura 5), permite a captura de valores do teclado. O *keyboard* foi usado nesse projeto para fazer o controle dos movimentos da cobrinha, usando as letras ”a” esquerda, ”s” baixo, ”d” direita, ”w” para cima e ”barra de espaço” para pausa.

O RARS também disponibiliza diversas outras ferramentas que possibilitam, a medição da quantidade de instruções executadas

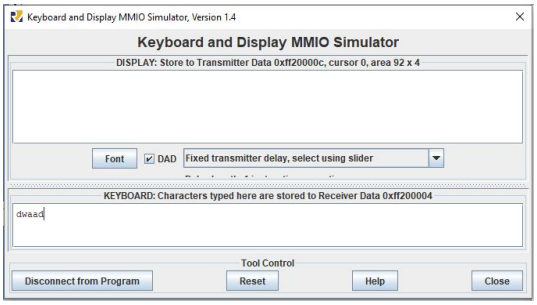

Figure 5: Ferramenta de captura do teclado.

incluindo os tipos de instruções que foram usados no código e o *”timer tool”* que contabiliza o tempo.

1.5 Paint.net

O Paint.net (figura 6) é um *software* gratuito e *open-source*, é util para diversas aplicações de edição de imagens e fotografia. Esse *software* foi usado para criar todas as imagens (.bmp) do jogo. O Paint.net é um *software* de fácil compreenssão e bastante intuitivo, possibilita editar e criar imagens a nível de pixel e formatar as cores em hexadecimal pelo código RGB, facilitando a compatibilidade entre o código em *assembly* e as imagens criadas.

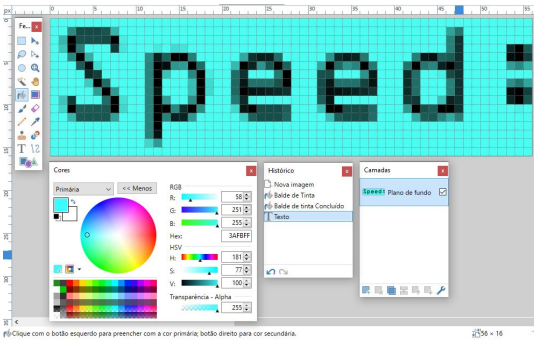

Figure 6: Ferramenta de captura do teclado.

2 Fundamentação Teórica

Para o desenvolvimento do projeto, foi necessário conhecimento sobre a arquitetura RISC-V e sua ISA [1], que foi adquirido ao longo do curso de Organização e Arquitetura de Computadores na Universidade de Brasília em que foram desenvolvidos outros projetos envolvendo a utilização da ISA RV32IMF e a organização processador RISC-V.

Os processadores são desenhados para que as instruções seja codificadas em bits.[4] Programas são armazenados na memória para serem lidos da mesma forma que os dados.A figura 2 mostra como o processador lida com as instruções do programa.As Instruções são buscadas na memória do endereço armazenado no registrador PC : Program Counter e colocadas no registrador IR : Instruction Register, os bits do registrador IR controlam as ações subsequentes necessárias à execução da instrução.Busca a próxima instrução e continua .

2.1 Pipeline

Um dos processos mais usados é o que conhecemos na indústria como linha de montagem (”pipeline”)[1], no qual o processador se divide em várias partes funcionais distintas (estágios), cada uma correspondendo a uma determinada atividade. A idéia básica num

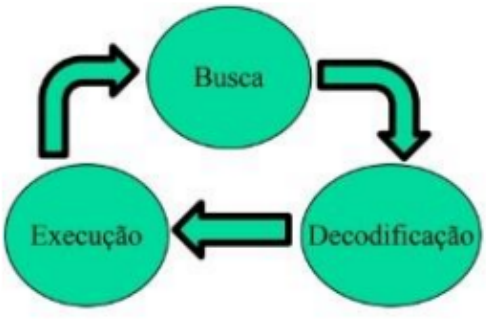

Figure 7: Ciclo de Busca e Execução.

pipeline de instruções é a de novas entradas serem aceitas, antes que as entradas aceitas previamente tenham terminado. Este conceito assume que uma instrução tem vários estágios, como mostra a figura a 8[2]:

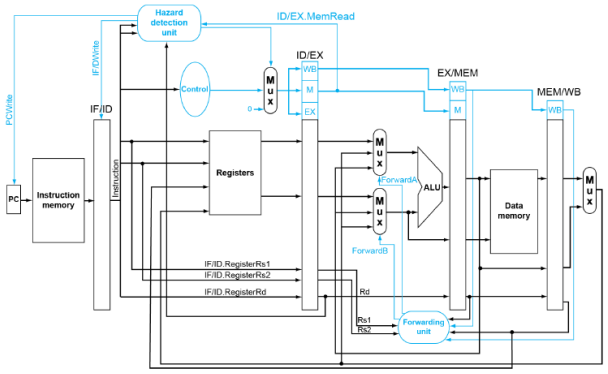

Figure 8: Caminho de Dados do Pipeline .

Em que IF: Instruction Fetch e PC Increment ID: Instruction Decode / Register Fetch EX: Execution MEM: Memory Stage WB: Write Back[5]

Neste projeto foi utilizado a ISA Risc-V[6], que chamaremos de RV32-IMF, para o pipeline, pois há suporte a instruções do tipo inteiros, multiplicação, divisão e ponto flutuante de precisão simples.

3 METODOLOGIA

A técnica utilizada para a realização desse projeto esta, em todos seus aspectos, intimamente relacionada as características da principal ferramenta utilizada para a implementação do jogo, o RARS.

Pensando nas diversas ferramentas e suporte ao programador que o RARS possui, foram tomadas várias decisões e princípios de trabalho que definiram a filosofia do projeto. Por exemplo, na primeira etapa de criação do projeto, foi definido que seria sempre usado um bloco ou unidade fundamental constituida de 8x8 pixeis, responsável por definir o corpo da cobrinha, o tamanho da maçã e o tamanho que ela crescera quando comer.

A programação do aplicativo foi dividida em etapas. Em que, os procedimentos foram criados em arquivos ".asm" independentes e depois chamados pela main ou por outros procedimentos secundarios, tornando comum o uso das funções, "jal", "j", "ret" e da pilha, todas esses procedimentos foram adcionados a main usando-se o ".include". Essa forma de organização foi adotada para facilitar a programação em grupo, permitindo a cada membro trabalhar em diferentes partes do projeto paralelamente, usando de boas práticas de programação para facilitar a junção dos códigos. O *github* tambem foi amplamente utilizado para facilitar a atualização das novas versões do código, que eram criadas por cada membro simultaneamente.

A função 'main' chama as demais funções. Os registradores 'S' salvam constantes como, as teclas de comando, o tempo de pausa, pontuação, última tecla lida e o vetor que armazena toda cobra. A figura 15 foi retirada do arquivo readme, gerado para facilitar as boas práticas de programação em grupo.

Registradores	Uso
s0	ascii value to 'a'
s1	ascii value to 'd'
s2	ascii value to 's'
s3	ascii value to 'w'
s4	Vector with the coordinates of the snake
s5	Coordinate of the fruit
s6	Time of Sleep function
s7	Score
s11	The last key pressed

Figure 9: Lista de registradores salvos.

Há a função tela de fundo, responsável pela cor. Para a implementação do Score, foi usado uma imagem da palavra 'score', que foi criado na função 'score_image' e ao lado, em que a potuação, de fato, acontece, é resultado da função score, resultado este que é incrementado de 7 em 7 pontos.

A função 'frutinha' gera o alimento da cobra em pontos aleatórios dentro do campo de movimentação da cobra, na realidade o alimento é um conjunto de bits que formam o ponto vermelho. Para comandar a cobra foi usado o algoritmo do 'keypoll' disponibilizado pelo professor, programa roda em *loop* até que uma tecla seja apertada, técnica também conhecida como *polling*.

há bordas prédefinidas, se o vetor que define a cobra receber o endereço de memória de algum item da borda significa que a cobra tocou no limite de sua movimentação, então ocorre o 'game over' e o jogador perde. Se endereço de memória da cobra entrar um endereço que também está no vetor cobra, significa que a cobra comeu a si mesma, então acaba o jogo. Quando ocorre o 'game over' é mostrado uma imagem característica de fim de jogo.

Com auxílio das chamadas ao sistema, é possível incluir som, dessa forma, quando a cobra come ou quando bate num obstáculo é emitido um som. Cada nota musical tem seu código, dessa maneira foi póssivel implementar sons específicos para cada situação.

Os procedimentos denominados "PontoBaixo.asm", "PontoDireita.asm", "PontoEsquerda.asm" e "PontoSobe.asm", são simplesmente blocos de 8x8 *pixels* e são chamadas em *loop* para formar o corpo da cobrinha, figura 10. Essas funções eram originalmente uma única função denominda "Ponto.asm", mas devido a forma como o bloco era criado (criando oito linhas paralelas e horizantais de comprimento 8), prejudicava a estética do movimento uniforme e contínuo da cobrinha para as outras direções (com exceção da direita), por isso foram criadas todas essas versões apenas mudando a forma como as linhas são escritas dentro do bloco.

4 RESULTADOS

A aplicação *Square Snake* está plenamente funcional, abordando todos os requisitos exigidos para a execução desse projeto.

O jogo inicia com uma imagem inicial que permanece estática até o usuário apertar uma das teclas de direção, iniciando assim o jogo de fato. Durante o jogo é possível vizualizar a pontuação

```
Main.asm | Ponto.asm* | PontoDireita.asm | PontoEsquerda.asm | PontoSobe.asm | PontoBaixo.asm

Ponto:
slli a0, a0, 3 #Converte pontos de 40x30 para 320x240
li t3, 0x0000FFFF
li t4, 0xFFFF0000
li t6, 7
addi t0, a0, 0    # t0 recebe a coordenada
addi t1, a1, 0    # t1 recebe a cor
li t5, 0
and a1, t0, t3    # a0 recebe os 16 primeiros bits
and a0, t0, t4    # a1 recebe os 16 menos significativos
srli a0, a0, 16   #

addi a2, a0, 8
addi a3, a1, 0
addi a4, t1, 0
```

Figure 10: Trecho do código do procedimento ”Ponto.asm” .

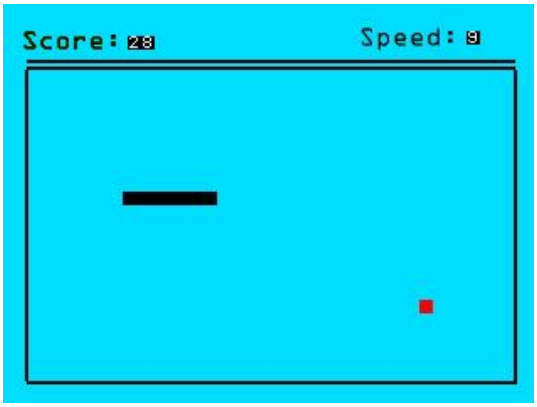

Figure 12: Square Snake sendo jogado.

(quantidade de maçãs multiplicadas por 7) e a velocidade da cobrinha (aumenta sempre que come uma maçã), a velocidade e o crescimento da cobrinha são dois fatores que aumentam a dificuldade do jogo gradativamente.

O *game over* pode ocorrer quando a cobrinha bate em uma das bordas que limitam o campo do jogo ou quando ela encontra seu próprio corpo, nesse caso em que se perde o jogo, uma tela final de *game over* com uma imagem surge, indicando a derrota (figura 11). Após a derrota e o surgimento da imagem de game over, a imagem inicial volta a aparecer dando a opção de uma nova partida ao jogador.

Figure 11: Imagem de Game Over.

As imagens de inicio e de fim de jogo são printadas na frame 1, sendo elas as unicas informações ocupando essa frame, enquanto a frame 0 roda todo o jogo. Durante a criação dessas imagens foi preciso fazer ajustes para diminuir a quantidade de memória usada, devido a limitação de memória da FPGA, para resolver esse problema somente a imagem de game over foi modificada, mudando sua proporção de 320x240 para 100x100 e preenchendo o resto da tela, usando as ecalls para pintar a tela de preto e para printar a frase ”Game Over”, tornando assim o projeto com um tamanho de memória viável.

A frame 0 é onde roda todo o jogo como mencionado anteriormente, nesta frame as duas imagens printadas juntamente com a funçaõ que define a tela de fundo (procedimento ”TelaFundo”) são as palavras ”Score:” e ”Speed:”, essas imegens são pequenas o suficiente para não ultrapassar o limite de memória juntamente com o código figura 12.

Na frame 0, os números inteiros que mostram os valores da pontuação e da velocidade printados na tela, são feitos usando a ecall que executa essa tarefa. No entanto, inicialmente houve uma tentativa de printar esses números usando-se imagens .bmp, figura 13 apenas para melhorar a estética, mas a dificuldade adicional para implementar esta ideia apenas para uma melhoria praticamente insignificante, torno-a inviável, optando-se por simplesmente usar uma ecall.

Figure 13: Modelo inicial para os números do score e speed.

Embora o projeto tenha sido implementado de forma fragmentada, foram necessárias algumas funções mais robustas como as funções ”Directions” e ”Directions.Long” responsáveis por todo o controle da cobrinha. A figura **??** mostra a quantidade de instruções geradas durante uma partida, essa quantidade aumenta de acordo com o tempo em que o jogo fica rodando. Mas podemos ver as proporções em que cada tipo de instrução foi usada, esses dados podem ser úteis para estudos mais aprofundados e futuras melhorias no código.

Há um *bug* no jogo, que só foi descoberto no último teste, não foi possível corrigir o erro devido ao prazo de entrega do projeto. A posição da maça é sorteada aleatóriamente, a cada vez que é comida, porém a maçã pode ser sorteada numa posição em que já está o corpo da cobra, dessa forma o jogador não conseguirá comer a maçã. Um *bug* simples de corrigir, basta sortear o alimento da cobra em um *loop* de verificação, em que avalia se a posição sorteada, é ou não, igual a algum valor dentro do vetor que armazena a cobra.

Foi feito a filmagem do jogo para exemplificar o modo de utilização e apresentar o o jogo em plena operação que pode ser visto através do vídeo no link: https://youtu.be/yogM9r2ar64

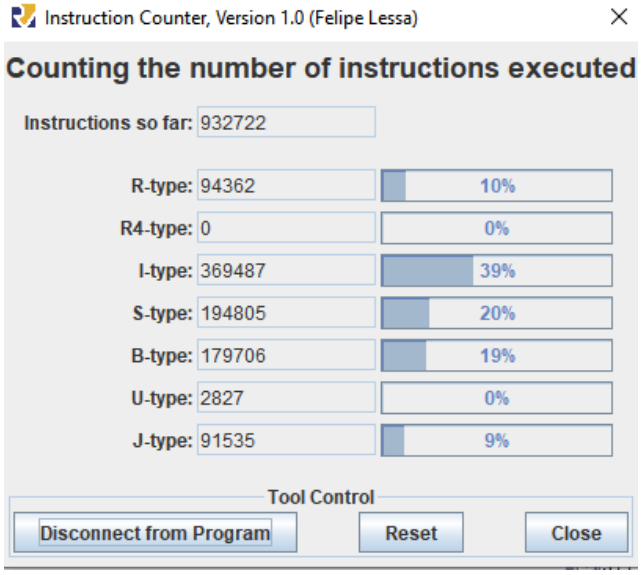

Figure 14: Número aproximado de Intruções do Jogo.

gravado utilizando a placa DE1-SoC.

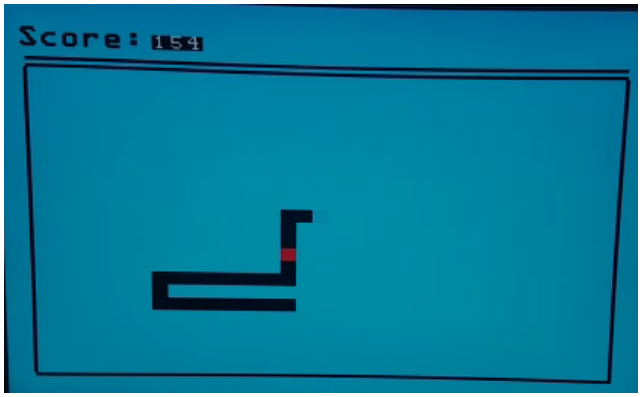

Figure 15: tela do jogo projetada pela FPGA.

5 Conclusões

O 'Snake' implementado apresentou certo nível de complexidade, principalmente ao fazer curvas consecutivas, no entanto, as dificuldades foram sanadas, assim como o *bug* que havia, quando a cobra encostava em uma borda, esta era tratada como o alimento da cobra, devido isto, a cobra comia a borda ao invés de encerrar o jogo. inserir imagens como no inicio e no *Game Over* apresentaram problemas no início, entretanto, todos estes bugs foram verificados e corrigidos.

Para trabalhos futuros, espera-se a correção do erro quando a maçã é sorteada para uma posição já ocupada pelo corpo da cobra. Além disso, espera-se aumentar a dificuldade do jogo ao se alcançar determinada pontuação. Por exemplo, o surgimento de outros obstáculos dentro do campo de movimentação da cobra.

References

[1] L. Coelho. *ARQUITETURA DE COMPUTADORES Unidade Central de Processamento – U C P*. IFRN, Instituto Federal do Rio Grande do Norte, January 2013.

[2] D. A. P. e John L. Hennessy. *Computer Organization and Design: The Hadware/Software Interface*. Morgan Kaufmann Publishers, risc-v edition, 2017.

[3] James. History of nokia part 2: Snake.

[4] M. V. Lamar. *Aula 16 Implementação RISC-V Pipeline - Conceitos*. UNB, Universidade de Brasília, January 2016.

[5] M. V. Lamar. *Aula 17 Implementação RISC-V Pipeline – Unidade Operativa e Controle*. UNB, Universidade de Brasília, January 2016.

[6] M. V. Lamar. *Aula 4 Arquiteturas de Processadores*. UNB, Universidade de Brasília, January 2016.