Desenvolvimento de Jogo em Assembly RISC-V: Metroid (NES)

Luana Cruz Silva [*] Lucas de Oliveira Silva [†]

Universidade de Brasília, 12 de setembro de 2024

Figura 1: *Metroid*

RESUMO

Este projeto consiste no desenvolvimento de um jogo temático baseado em Metroid (NES) utilizando a linguagem Assembly RISC-V e implementado em uma FPGA. O objetivo é aplicar os conhecimentos adquiridos nas disciplinas de Organização e Arquitetura de Computadores (OAC) e programação em Assembly. O jogo contém mecânicas de movimentação, ataque e interação com itens, além de múltiplos ambientes e inimigos.

Palavras-chave: OAC · Assembly IRSC-V · Metroid NES

1 INTRODUÇÃO

O presente trabalho visa consolidar os conhecimentos adquiridos no curso de OAC, utilizando o RISC-V para a criação de um jogo temático inspirado em Metroid. O projeto envolve desde a programação de baixo nível em Assembly até a integração com periféricos externos.

Este tipo de atividade prática é essencial para o desenvolvimento de habilidades em design de hardware e software, especialmente no contexto de sistemas embarcados.

2 FUNDAMENTAÇÃO TEÓRICA E TRABALHOS RELACIONADOS

2.1 LINGUAGEM ASSEMBLY E ARQUITETURA RISC-V

A linguagem Assembly é um dos meios mais diretos de comunicação entre software e hardware, permitindo o controle granular do processador. Em arquiteturas RISC (Reduced Instruction Set Computing), como a RISC-V, a programação em Assembly torna-se especialmente eficiente devido à simplicidade e regularidade de suas instruções, o que facilita tanto o desenvolvimento quanto o ensino de conceitos fundamentais de organização de computadores.

Segundo Patterson e Hennessy no livro "Computer Organization and Design: The Hardware/Software Interface"(2011) [1], a arquitetura RISC foi desenvolvida para reduzir a complexidade das instruções, proporcionando um desempenho superior em sistemas embarcados e de tempo real, onde a eficiência de processamento e energia é essencial. O RISC-V é um exemplo moderno desta abordagem, oferecendo um conjunto de instruções abertas e modulares, permitindo a inclusão de extensões conforme a necessidade, como operações de ponto flutuante (ISA RV32IMF) ou segurança avançada.

O design simplificado do RISC-V permite que os alunos compreendam como o hardware interage com o software, como o gerenciamento de registradores e operações de controle, facilitando a criação de jogos como o proposto neste projeto.

2.2 O JOGO METROID (NES)

O Metroid (1986), desenvolvido pela Nintendo para o NES, é um clássico no gênero de ação-aventura, introduzindo um dos primeiros exemplos de jogabilidade não-linear em consoles.

A história se passa no planeta Zebes, onde Samus Aran tenta recuperar os Metroids, que são organismos parasitas roubados pelos Space Pirates, que planejam a replicação desses organismos expondo-os a raios beta e depois utilizá-los como armas biológicas para destruir Samus e todos os seres que se opõem a eles.

Em Metroid, o jogador controla Samus Aran, navegando por uma série de áreas conectadas, enfrentando inimigos e coletando itens que permitem o acesso a novas regiões do mapa. [2] [3]

3 METODOLOGIA

O projeto foi desenvolvido em Assembly RISC-V, arquitetura de CPU multiciclo RV32IM, com a intenção de ser implementado em uma FPGA, utilizando periféricos como monitor, teclado e caixa de som. O jogo foi projetado para atender a vários requisitos, incluindo a implementação de música e efeitos sonoros, ataques do jogador, movimentação e animação de personagens, itens interativos, informações sobre a vida e equipamentos, múltiplas salas distintas, inimigos com IA, e um background móvel.

O desenvolvimento do jogo foi dividido nas seguintes etapas inicialmente:

[*] 202033543@aluno.unb.br
[†] 200022857@aluno.unb.br

- **Planejamento do Jogo**: Definimos o design do jogo e fizemos o levantamento dos requisitos principais, conforme as especificações do projeto (Movimentação, IA, Sons, Itens).
- **Programação em Assembly**: Desenvolvimento do código em Assembly RISC-V, utilizando o conjunto de instruções RV32IM, focando na movimentação do personagem principal e nas interações com o ambiente.
- **Implementação na FPGA**: O código Assembly deveria ser testado no hardware da FPGA, conectando os periféricos como monitor, teclado e caixa de som para garantir a jogabilidade e a interação física do jogo.
- **Desenvolvimento da IA dos Inimigos**: Focar também na implementação de inimigos diferentes com IA (número de inimigos em aberto), sendo um deles um chefão, tendo comportamentos inteligentes, incluindo padrões de ataque e movimentação.
- **Testes e Debugging**: Testar para corrigir erros de código e otimizar a performance do jogo na FPGA.

A implementação das instruções de controle e uso das habilidades no jogo foram implementadas para seguir a seguinte forma:

- **Movimentação**:
  - **A**: Move o personagem (Samus) para a esquerda.
  - **D**: Move o personagem (Samus) para a direita.
- **Ações**:
  - **W**: Faz o personagem (Samus) pular.
  - **E**: Realiza um ataque padrão (tiro) com a arma equipada.
- **Armas Especiais**:
  - **F**: Após adquirir a arma de gelo, ativa o tiro de gelo, que congela inimigos e pode impactar o ambiente.
  - **R**: Após adquirir o míssil, dispara um míssil que causa dano significativo aos inimigos e pode destruir certos obstáculos.

3.1 Música e Efeitos Sonoros

A música no jogo foi implementada utilizando instruções em para enviar parâmetros de som através de chamadas de sistema (syscalls). Cada nota musical é definida em termos de frequência, duração, instrumento e volume

Listing 1: *Trecho da implementação de música e efeitos sonos*

```
    li a0,100    # define a nota
  li a1,100    # define a dura  oo da nota em
      ms
  li a2,127    # define o instrumento
  li a3,127    # define o volume
  li a7,33     # define o syscall
  ecall
```

3.2 Ataques do Jogador

O jogador possui a habilidade de disparar diferentes tipos de ataques, incluindo o tiro de gelo. Parte da implementação principal para essa habilidade:

Listing 2: *Trecho da implementação do tiro de gelo*

```
ICE_SHOT_LADO_M3:
  la t5,LAST_DIREC
  lw t5,0(t5)
  li t1,'a' # tecla 'a'
  beq t5,t1,ICE_SHOT_ESQ_M3 # disparo para a
      esquerda

  li t1,'d' # tecla 'd'
  beq t5,t1,ICE_SHOT_DIR_M3 # disparo para a
      direita
```

O tiro de gelo é uma habilidade de arma de combate. Ele congela os inimigos, tornando-os vulneráveis por um tempo e também aumenta o dano causado, atingindo 2 de dano por tiro, em comparação com o dano padrão de 1.

Já o míssil é uma arma que causa 4 de dano ao atingir um inimigo. Além de seu alto poder de ataque, o míssil também permite ao jogador acessar novas áreas no jogo, que não podem ser destruídos com tiros normais.

Listing 3: *Trecho da implementação do míssel*

```
START_SHOT_RIDLEY:
  la a0, RIDLEY_POS
  lw a1, 16(a0)
  addi a1, a1, 1 # Incrementa o intervalo que
       Ridley est  atirando, depois de 300
      (301)
  sw a1, 16(a0)

  # Setar posi  o do primeiro tiro
  lw a1, 0(a0)
  lw a2, 4(a0)
  addi a1, a1, -5
  addi a2, a2, 20

  la t0, RIDLEY_HIT
  sw a1, 0(t0)
  sw a2, 4(t0)

  # Setar posi  o do segundo tiro
  addi a2, a2, -6

  sw a1, 8(t0)
  sw a2, 12(t0)

  # Setar posi  o do terceiro tiro
  addi a2, a2, -7

  sw a1, 16(t0)
  sw a2, 20(t0)

  ret
```

3.3 Movimentação e Animação dos Personagens

A movimentação e animação dos personagens são geridas através de funções que atualizam o estado dos personagens com base em suas ações e eventos do jogo. Para Samus, a animação é atualizada quando ela dispara um tiro:

Listing 4: *Trecho da implementação da atualização da Samus*

```
  la a0,FRAME_SAMUS # Atualiza a anima  o
      da Samus
  sh zero,0(a0)
```

Ridley também possui animações para pulo e ataque, configuradas e atualizadas conforme ele realiza essas ações:

Listing 5: *Trecho da implementação da atualização da Samus*

```
START_JUMP_RIDLEY:
  la a0,RIDLEY_POS
  sw zero,20(a0) # Configura o frame de pulo
```

3.4 Itens e Acessos

O jogo inclui dois itens com efeitos distintos:

- **Item 1: Tiro de Gelo** - Se o jogador pegar esse item, ele é capaz de congelar inimigos e criar plataformas temporárias. Impacta o combate ao permitir a manipulação do ambiente e a neutralização de inimigos.

- **Item 2: Míssel** - Este item pode ser usado para acessar novas áreas, proporcionando acesso a novas salas ou caminhos no jogo.

Listing 6: *Trecho da implementação que verifica se a Samus pegou o item de Gelo*

```
    PICK_ITEM_1: # verifica se a samus
        pegou o item
    la t0,MAP_POS # se o item n o est na
        tela, pula
    lw t0,0(t0)
    li t1,350
    bgt t0,t1,DANO_SAMUS_ZOOMER
    la t0,ITENS # se o item ja foi pego, pula
    lh t0,0(t0)
    bnez t0,DANO_SAMUS_ZOOMER
```

Listing 7: *Trecho da implementação que verifica se a Samus pegou o item de Míssel*

```
    PICK_ITEM_2: # verifica se a samus
        pegou o item
    la t0,MAP_POS # se o item n o est na
        tela, pula
    lw t0,0(t0)
    li t1,208
    bgt t0,t1,DANO_SAMUS_RIPPER
    la t0,ITENS # se o item ja foi pego, pula
    lh t0,2(t0)
    bnez t0,DANO_SAMUS_RIPPER
```

3.5 Informações sobre a Vida e Equipamentos

A vida e o equipamento da Samus são geridos por variáveis específicas na implementação do código. Por exemplo, a vida de Samus pode ser monitorada conforme ela toma dano e os itens que a personagem pega durante o jogo são mostrados.

Listing 8: *Trecho da implementação sobre vida*

```
########## IMPRIMIR STATUS NA TELA, VIDA E
    ETC
    la a0,statusfull #INICIA O REGISTRADOR
        COM A IMAGEM DO MENU
    li a1,64 # LARGURA DA IMAGEM
    li a2,32# ALTURA DA IMAGEM
    mv a3,s0 # alterna o frame em que
        trabalhamos, definir o frame atual na
         verdade
    call PRINT
```

A vida de Samus é armazenada na variável LIFE-SAMUS. Para exibir a vida no jogo, o valor dessa variável é carregado no registrador t1. Em seguida, o valor é processado para extrair os dois dígitos da vida. Isso é feito por meio de operações de divisão e módulo. O dígito das unidades é obtido utilizando a operação de módulo por 10, enquanto o dígito das dezenas é obtido dividindo o valor total por 10. Esses dígitos são então comparados com valores de 0 a 9 para determinar a representação visual da vida no jogo.

Listing 9: *Trecho da implementação sobre vida*

```
########## IMPRIMIR O PRIMEIRO DIGITO DA VIDA
    li a1,89 # LARGURA DA IMAGEM
    li a2,32# ALTURA DA IMAGEM
    mv a3,s0 # definir o frame atual
    call PRINT
```

3.5.1 Detecção de Dano de Samus pelo Ridley

A função DANO-SAMUS-RIDLEY é responsável por verificar se Samus está colidindo com o Ridley e, se estiver, aplicar o dano correspondente à vida de Samus. O processo é dividido em:

- **Delay de Detecção de Dano:** Para evitar que Samus receba dano contínuo e instantâneo ao colidir com Ridley, é implementado um atraso (delay). Isso é feito carregando o valor de um temporizador da variável DELAY e aplicando uma operação de módulo para verificar se o intervalo de tempo é suficiente para permitir a detecção de dano apenas uma vez por período especificado. Se o tempo decorrido desde o último dano for menor que 100 unidades de tempo, a função salta para DANO-SHOTS-M3 para evitar a aplicação de dano imediato.

- **Verificação de Colisão:** Se o atraso for suficiente, a função continua obtendo as posições e áreas de colisão de Samus e Ridley. As posições são carregadas das variáveis CHAR-POS (para Samus) e RIDLEY-POS (para Ridley). A função VERIFICA-HIT-BOX é chamada para verificar se as áreas de colisão de Samus e Ridley se sobrepõem. Os parâmetros passados para esta função definem as áreas de detecção de colisão para ambos os personagens.

- **Aplicação de Dano:** Caso a função VERIFICA-HIT-BOX indique que houve uma colisão (verificando o registrador a6 que não é zero), o dano é aplicado a Samus. O valor da vida de Samus é carregado da variável LIFE-SAMUS, reduzido em 5 unidades e atualizado de volta na variável.

  Controle de Fluxo: Se não houver colisão detectada, a execução salta para DANO-SHOTS-M3, onde outras ações podem ser tratadas, evitando a redução da vida.

Listing 10: *Trecho da implementação sobre dano*

```
DANO_SAMUS_RIDLEY: # verifica se a samus
    est  encostada no ridley
    la a0,DELAY
    lh a1,0(a0)
    li a0,100
    rem a1,a1,a0
    bnez a1,DANO_SHOTS_M3

    la a1,CHAR_POS
    lw a0,0(a1)
    lw a1,4(a1)
    li a2,24
```

```
    li a3,32

    la t1,RIDLEY_POS
    lw t0,0(t1)
    lw t1,4(t1)
    li t2,32
    li t3,40
    call VERIFICA_HIT_BOX

    beqz a6,DANO_SHOTS_M3

    # se a colis o foi detectada, a samus
        perde vida
    la a0,LIFE_SAMUS
    lw a1,0(a0)
    addi a1,a1,-5
    sw a1,0(a0)
```

Essas etapas garantem que Samus só receba dano quando há uma colisão real com Ridley, e evita a aplicação de dano contínuo ou excessivo.

3.6 Salas Distintas

O jogo possui três salas distintas, cada uma representando um ambiente separado. As transições entre essas salas são geridas por portas ou áreas de passagem:

- **Sala 1:** Área inicial do jogo, onde o jogador começa e pode explorar os primeiros desafios.

Listing 11: *Trecho da implementação da sala 1*

```
    la a0,cenario1 #INICIA O REGISTRADOR
        COM A IMAGEM DO cenario
    li a1,0 # LARGURA DA IMAGEM
    li a2,0 # ALTURA DA IMAGEM
    mv a3,s0 # alterna o frame em que
        trabalhamos, definir o frame
        atual na verdade
    call PRINT_MAPA
```

- **Sala 2:** Ambiente intermediário com novos inimigos e obstáculos, separado por uma porta ou outro mecanismo de transição.

Listing 12: *Trecho da implementação da sala 2*

```
    la a0,cenario2 #INICIA O REGISTRADOR
        COM A IMAGEM DO cenario
  li a1,32 # LARGURA DA IMAGEM
  li a2,0 # ALTURA DA IMAGEM
  mv a3,s0 # alterna o frame em que
      trabalhamos, definir o frame atual
      na verdade
  call PRINT_MAPA_M2
```

- **Sala 3:** Sala final ou um estágio avançado do jogo, possivelmente com um chefe ou desafio final.

Listing 13: *Trecho da implementação da sala 3*

```
    la a0,cenario3 #INICIA O REGISTRADOR
        COM A IMAGEM DO cenario
  li a1,0 # LARGURA DA IMAGEM
  li a2,0 # ALTURA DA IMAGEM
  mv a3,s0 # alterna o frame em que
      trabalhamos, definir o frame atual
      na verdade
  call PRINT_MAPA_M3
```

3.7 Inimigos e IA

O jogo apresenta pelo menos três tipos de inimigos diferentes, cada um com sua própria IA.

- **Inimigo 1:** Inimigo básico com comportamento de patrulha simples.
- **Inimigo 2:** Inimigo que persegue o jogador e ataca com um padrão mais complexo.
- **Chefe:** Inimigo chefe com um comportamento avançado e múltiplas fases de ataque, oferecendo um desafio significativo.

Quanto a implementação dos inimigos, a função START-JUMP-RIDLEY inicializa a animação de pulo do Ridley e atualiza o intervalo e a posição de Ridley durante o pulo.

Listing 14: *Trecho da implementação da função START-JUMP-RIDLEY*

```
  START_JUMP_RIDLEY:
  la a0, RIDLEY_POS
  sw zero, 20(a0)  # coloca o frame de pulo
      dele

  lw a1, 12(a0)
  addi a1, a1, 1  # incrementa o intervalo do
      pulo
  sw a1, 12(a0)

  lw a1, 4(a0)
  addi a1, a1, 2
  sw a1, 4(a0)

  ret
```

Função START-SHOT-RIDLEY configura a posição dos tiros disparados por Ridley e atualiza os registros relacionados ao ataque de Ridley.

Listing 15: *Trecho da implementação da função START-SHOT-RIDLEY*

```
  START_SHOT_RIDLEY:
  la a0, RIDLEY_POS
  lw a1, 16(a0)
  addi a1, a1, 1  # incrementa o intervalo do
      tiro
  sw a1, 16(a0)

  # Define a posi   o dos tiros
  lw a1, 0(a0)
  lw a2, 4(a0)
  addi a1, a1, -5
  addi a2, a2, 20

  la t0, RIDLEY_HIT
  sw a1, 0(t0)
  sw a2, 4(t0)

  addi a2, a2, -6
  sw a1, 8(t0)
  sw a2, 12(t0)

  addi a2, a2, -7
  sw a1, 16(t0)
  sw a2, 20(t0)

  ret
```

MOVE-RIDLEY controla o movimento vertical de Ridley, alternando entre subir e descer com base no intervalo de tempo.

Listing 16: *Trecho da implementação da função MOVE-RIDLEY*

```
MOVE_RIDLEY:
la a0, RIDLEY_POS
lw a1, 12(a0)
li a2, 200
blt a1, a2, MOVE_RIDLEY_RET

addi a1, a1, 1
sw a1, 12(a0)

li a2, 230
ble a1, a2, MOVE_RIDLEY_BAIXO

# Move Ridley para cima
lw t1, 4(a0)
addi t1, t1, 2
sw t1, 4(a0)

j MOVE_RIDLEY_END

MOVE_RIDLEY_BAIXO:
# Move Ridley para baixo
lw t1, 4(a0)
addi t1, t1, -2
sw t1, 4(a0)

MOVE_RIDLEY_END:
li a2, 258
bne a1, a2, MOVE_RIDLEY_RET

# Zera a anima  o de pulo
li a1, 1
sw a1, 20(a0)  # coloca o frame no ch o
sw zero, 12(a0)

MOVE_RIDLEY_RET:
ret
```

A função responsável por atualizar a posição dos tiros de Ridley e faz com que eles se movam até desaparecerem é a SHOT-RIDLEY-ANIMA

Listing 17: *Trecho da implementação da função SHOT-RIDLEY-ANIMA*

```
SHOT_RIDLEY_ANIMA:
la a0, RIDLEY_POS
lw a2, 16(a0)
li a1, 150
blt a2, a1, SHOT_RIDLEY_RET

addi a2, a2, 1
sw a2, 16(a0)

# Atualiza a posi  o dos tiros
la t0, RIDLEY_HIT
lw t1, 0(t0)
addi t1, t1, -3
sw t1, 0(t0)

lw t1, 8(t0)
addi t1, t1, -3
sw t1, 8(t0)

lw t1, 16(t0)
addi t1, t1, -3
sw t1, 16(t0)

li a1, 165
bgt a2, a1, SHOT_RIDLEY_BAIXO

# Tiros subindo
lw t1, 4(t0)
addi t1, t1, -2
sw t1, 4(t0)

lw t1, 12(t0)
addi t1, t1, -3
sw t1, 12(t0)

lw t1, 20(t0)
addi t1, t1, -4
sw t1, 20(t0)

j SHOT_RIDLEY_END

SHOT_RIDLEY_BAIXO:
# Tiros descendo
lw t1, 4(t0)
addi t1, t1, 2
sw t1, 4(t0)

lw t1, 12(t0)
addi t1, t1, 3
sw t1, 12(t0)

lw t1, 20(t0)
addi t1, t1, 4
sw t1, 20(t0)

SHOT_RIDLEY_END:
li a1, 180
bne a2, a1, SHOT_RIDLEY_RET

# Zera a anima  o de tiro
la a0, RIDLEY_POS
lw a2, 16(a0)
sw zero, 16(a0)

SHOT_RIDLEY_RET:
ret
```

3.8 Background Móvel

O background móvel é uma característica que acompanha o movimento da Samus, dando uma certa profundida e imersão na jogabilidade. O código para a atualização do background deve garantir que ele se mova horizontalmente ou verticalmente com base no movimento da Samus, criando uma sensação de continuidade no ambiente.

Listing 18: *Trecho da implementação do background móvel*

```
BACKGROUND_UPDATE:
  la a0,BACKGROUND_POS
  lw a1,0(a0)
  # Atualiza a posi  o do background com
      base na movimenta  o da Samus
  sw a1,NEW_POSITION(a0)
```

O código foi implementado para que o background se mova suavemente e sincronize com a movimentação da Samus, melhorando a experiência visual do jogo.

4 Resultados Obtidos

- **Música e Efeitos Sonoros**: A implementação de música e efeitos sonoros foi bem-sucedida. Há efeitos sonos de ataque, de dano e efeitos sonoros do Ridley.

- **Ataques do Jogador**: Os ataques foram implementados conforme o que foi solicitado nas especificações. A Samus pode usar diferentes tipos de ataques, incluindo o tiro de gelo, que foi programado para congelar inimigos e criar novos caminhos. Podemos ver que o sistema de ataque responde correta-

mente às entradas do jogador e afeta os inimigos e o ambiente conforme esperado.

Figura 2: *Tiro padrão*

Figura 3: *Tiro de gelo*

Figura 4: *Míssel*

- **Movimentação e Animação dos Personagens**: A movimentação e animação dos personagens, como a Samus e Ridley, foram implementadas com sucesso. A Samus possui animações para caminhada, pulo e ataques, enquanto Ridley tem animações para pular e atirar. As animações estão fluídas e acompanham as movimentações do jogados.

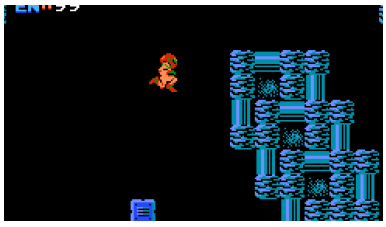

Figura 5: *Pulo Samus*

Figura 6: *Ridley voando*

- **Itens Interativos**: O jogo inclui dois tipos principais de itens: um que permite o acesso a novas áreas e outro que impacta diretamente o combate. O tiro de gelo, por exemplo, é um item que permite ao jogador congelar inimigos e abrir novos caminhos e o míssel pode ser usado para acessar novas salas. A implementação desses itens funcionam conforme o esperado.

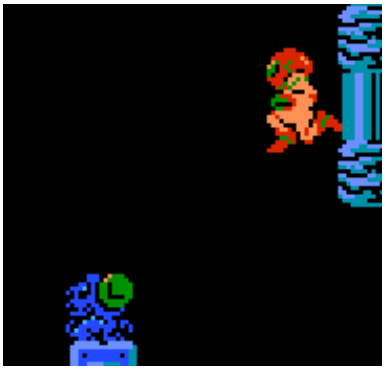

Figura 7: *Item de gelo*

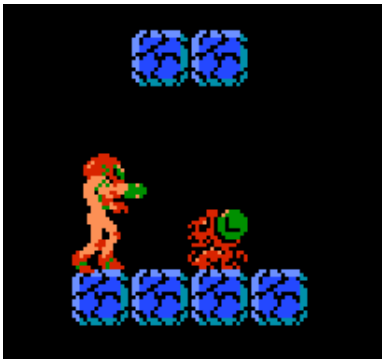

Figura 8: *Item Míssel*

- **Informações sobre Vida da Samus**: As informações sobre a vida da Samus é exibida corretamente no jogo. O sistema atualiza essas informações em tempo real, mostrando a condição atual de vida da personagem conforme ela toma dano.

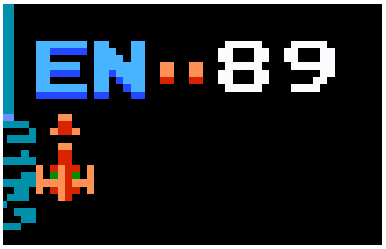

Figura 9: *Vida e item*

- **Salas Distintas**: Foram implementadas três salas distintas no jogo, cada uma com ambientes separados por portas, com a transição entre salas funcionando corretamente. As salas apresentam desafios e cenários fiés ao jogo.

Figura 10: *Sala 1*

Figura 11: *Sala 2*

Figura 12: *Sala 3*

- **Inimigos e Chefão**: O jogo inclui três tipos de inimigos com IA distintas, incluindo um chefão. A IA dos inimigos foi programada para criar padrões de ataque e movimentação consideravelmente desafiadores.

Figura 13: *Inimigo 1*

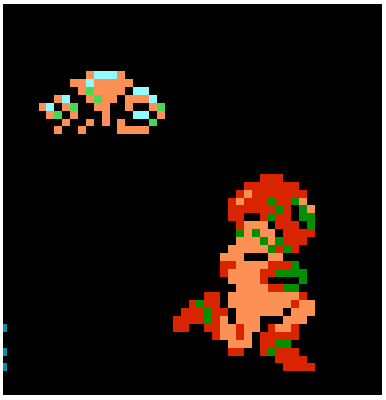

Figura 14: *Inimigo 2*

Figura 15: *Chefão*

- **Background Móvel**: O background móvel foi implementado para acompanhar o movimento da Samus, tanto horizontal quanto verticalmente, se movimentando de maneira sincronizada com os comandos do jogador e a personagem, criando uma sensação de profundidade e imersão no jogo que esperávamos.

5 Conclusões e Trabalhos Futuros

O projeto atende aos requisitos estabelecidos para um jogo funcional e envolvente. A música e os efeitos sonoros são implementados para enriquecer a experiência de jogo. Os ataques do jogador e a movimentação dos personagens estão bem definidos, permitindo uma jogabilidade satisfatória. Itens importantes são incluídos, a vida e o equipamento da Samus são geridos adequadamente, e o jogo apresenta várias salas distintas para exploração. Além disso, a IA dos inimigos e o background móvel contribuem para uma experiência de jogo mais rica.

O próximo passo seria realizar testes detalhados para garantir que o projeto funcione na FPGA, na qual não foi possível executar na implementação do jogo devido à quantidade de requisitos a serem cumpridos por uma dupla. Seria possível implementar outros aspectos para que o jogo ficasse mais fiel ao original, mas focamos em fazer pontualmente cada um dos itens solicitados na especificação do projeto aplicativo Metroid (NES).

Referências

[1] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 6th ed. Morgan Kaufmann, 2021.

[2] Nintendo of America, *Metroid Instruction Booklet*, USA: Nintendo of America, 1989. NES-MT-USA-1. Disponível em: https://www.nintendo.co.jp/clv/manuals/en/pdf/CLV-P-NAAQE.pdf

[3] B. Shoemaker, "The History of Metroid,"GameSpot, p. Metroid. Disponível em: https://web.archive.org/web/20131003050311/http://www.gamespot.com/gamespot/features/video/hist_metroid/p2_01.html