Assembler's Realm: Implementação de Engine de Jogo em Assembly RISC-V com Gerenciamento Otimizado de Hardware Simulado

Felipe G. V. Honda[1], **Vitor G. Duarte**[1], **Victor Wahrendorff V. de Azevedo**[1]

[1]Departamento de Ciência da Computação – Universidade de Brasília (UnB)
Brasília – DF – Brasil

{felipegvh, vitorgduarte, victorwahrendorff}@unb.br

Abstract. *This paper presents the technical development of Assembler's Realm, an action-adventure game developed entirely in RISC-V Assembly. The project was conceived as the final assignment for the Introduction to Computer Systems course, aiming to apply low-level concepts such as direct memory addressing, register allocation strategies, and MMIO control. The implementation features a custom engine supporting double buffering, asynchronous multi-channel audio via syscalls, and a tile-based collision system using bitwise operations for optimization. Code snippets regarding memory mapping and entity logic are discussed to demonstrate the control over the data flow.*

Resumo. *Este artigo descreve o desenvolvimento técnico de Assembler's Realm, um jogo de ação-aventura desenvolvido inteiramente em Assembly RISC-V. O projeto, parte da disciplina de Introdução aos Sistemas Computacionais, aplica conceitos de baixo nível como endereçamento direto de memória, estratégias de alocação de registradores e controle via MMIO. A implementação apresenta uma engine proprietária com suporte a double buffering, áudio multicanal assíncrono via syscalls e um sistema de colisão baseado em tiles otimizado por operações bit a bit. Trechos de código referentes ao mapeamento de memória e lógica de entidades são discutidos para demonstrar o controle sobre o fluxo de dados.*

1. Introdução

A programação em Assembly exige que o desenvolvedor transponha a barreira da abstração, gerenciando explicitamente o ciclo de vida de cada dado nos registradores da CPU. Este trabalho apresenta a implementação de *Assembler's Realm*, uma releitura de *The Legend of Zelda* (1986), desenvolvida para a arquitetura RISC-V no simulador RARS.

O desafio principal não residiu apenas na lógica do jogo, mas na construção de uma infraestrutura de software capaz de gerenciar renderização gráfica, física e áudio em um ambiente *single-thread* sem auxílio de bibliotecas externas. A Figura 1 ilustra o resultado visual alcançado através da manipulação direta do *Bitmap Display*.

Figura 1. Tela inicial renderizada pixel a pixel, demonstrando a capacidade gráfica da implementação.

2. Metodologia e Implementação

A arquitetura do sistema foi dividida em módulos (`.include`), separando a lógica (`main.s`) dos dados estáticos (sprites e mapas). O núcleo do sistema é um *Game Loop* que orquestra a temporização e execução das sub-rotinas.

2.1. Renderização e Acesso à Memória

Para a saída visual, o sistema interage com o *Bitmap Display* mapeado em memória. A ausência de funções de desenho nativas exigiu a implementação de um algoritmo de cálculo de endereço linear. Para otimizar o desempenho, substituiu-se a multiplicação por deslocamentos lógicos sempre que possível. A fórmula base utilizada é:

$$Endereço = Base + (y \times 320 + x) \times 4 \tag{1}$$

Para eliminar o *flickering*, implementou-se *Double Buffering*. O código alterna a escrita entre `0xFF000000` (Frame 0) e `0xFF100000` (Frame 1), utilizando uma operação `xori` no registrador de controle de frame ao final de cada ciclo.

2.2. Gerenciamento de Mapas e Portais

Para permitir a escalabilidade do mundo do jogo, desenvolveu-se uma estrutura de dados tabular para gerenciar transições de cena. A `PORTAL_TABLE` atua como um despachante dinâmico, associando IDs de colisão a endereços de memória de novos mapas e coordenadas de *spawn*. O trecho abaixo ilustra essa implementação:

```
PORTAL_TABLE:
    # ID, MapaEnd, ColisaoEnd, NovoX, NovoY, TileFundo
    .word 2, mapa2, mapa2_colisao, 16, 128, tile
    .word 3, mapa3, mapa3_colisao, 144, 192, tile2
    .word 5, dungeon, dungeon_colisao, 160, 208, preto
```

Quando o jogador colide com um tile especial (ex: ID 2), o sistema percorre esta tabela, carrega os novos endereços base nos registradores globais e reinicia o loop de renderização, permitindo transições instantâneas e sem carregamento perceptível.

2.3. Sistema de Áudio Assíncrono

A execução de áudio em sistemas monociclo apresenta o risco de bloquear a CPU. Para contornar isso, implementou-se um sequenciador baseado em tempo (`syscall 30`).

Uma decisão de design crucial foi a escolha da trilha *"Zelda's Lullaby"* (proveniente de *Ocarina of Time*). Além do apelo emocional e nostálgico que conecta o grupo à era clássica dos jogos, essa escolha teve um fundamento técnico: o compasso ternário e o andamento moderado da peça permitiram intercalar as instruções de envio de MIDI com os cálculos de física de forma mais eficiente do que o tema principal (que é rápido e denso), garantindo que a música tocasse sem "engasgar"a movimentação do Link.

2.4. Entrada e Física (MMIO)

A leitura do teclado é realizada via *Memory Mapped I/O* no endereço `0xFFFF0000`. Utiliza-se a técnica de *polling* não-bloqueante: se o bit de status for 0, a sub-rotina retorna imediatamente (‘ret‘), liberando ciclos para a IA dos inimigos.

A detecção de colisão utiliza uma matriz de *tiles*. Para converter coordenadas de pixel (0-320) para índices da matriz (0-20), utiliza-se a instrução `srai` (Shift Right Arithmetic Immediate) para dividir por 16, economizando ciclos em relação à instrução `div`.

3. Resultados e Discussão

O sistema final atingiu os requisitos funcionais com desempenho estável. A Figura 2 exibe o HUD e a renderização de múltiplos sprites simultâneos.

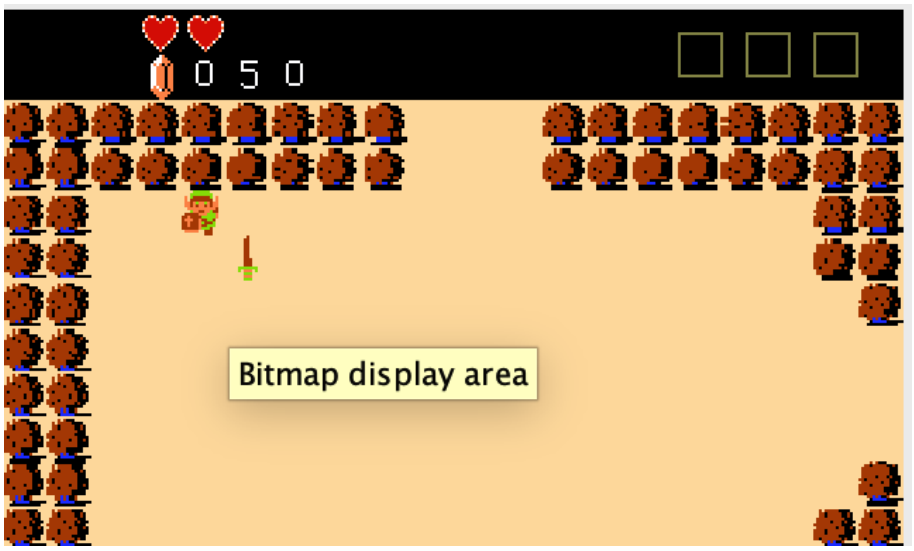

Figura 2. Gameplay no Deserto. O HUD exibe o estado dos registradores de vida e moedas em tempo real.

3.1. Inteligência Artificial e Colisão

Um desafio técnico relevante foi a colisão dos inimigos. Inicialmente, o inimigo ”Octorok”ficava preso em paredes ao mudar de direção. A solução foi implementar uma verificação de colisão ”inteligente”que testa o vértice do sprite correspondente à direção do movimento, e não apenas o centro. O código abaixo demonstra essa lógica para o movimento à direita:

```
CALCULAR_MOVIMENTO:
    # Se Vel > 0 (Direita), checamos o pixel X+15
    # Se Vel < 0 (Esquerda), checamos o pixel X
    mv  t5, a1              # t5 = X original
    bltz t4, CHECK_GRID_X   # Se esquerda, usa X
    addi t5, a1, 15         # Se direita, usa X + 15
CHECK_GRID_X:
    # ... chamada da rotina de colisão ...
```

Essa abordagem refinada garantiu que os inimigos navegassem pelo labirinto sem atravessar paredes ou ficarem estagnados.

3.2. Otimização: O Sistema de Rastro

Para viabilizar a execução no simulador, foi necessário reduzir o número de instruções de escrita na memória de vídeo (`sw`). Implementou-se um sistema de ”Limpeza de Rastro”. Em vez de redesenhar o mapa inteiro (background) a cada frame, o jogo salva as coordenadas anteriores das entidades (`OLD_POS`) e redesenha apenas os tiles de 16x16 pixels que foram ”sujos”pelo movimento. Isso reduziu a carga de renderização em aproximadamente 95%.

3.3. Economia e Persistência

O sistema de loja (Figura 3) manipula diretamente endereços de memória reservados para o inventário. A compra de itens verifica a colisão espacial e compara o valor no registrador de Rúpias, atualizando o HUD instantaneamente.

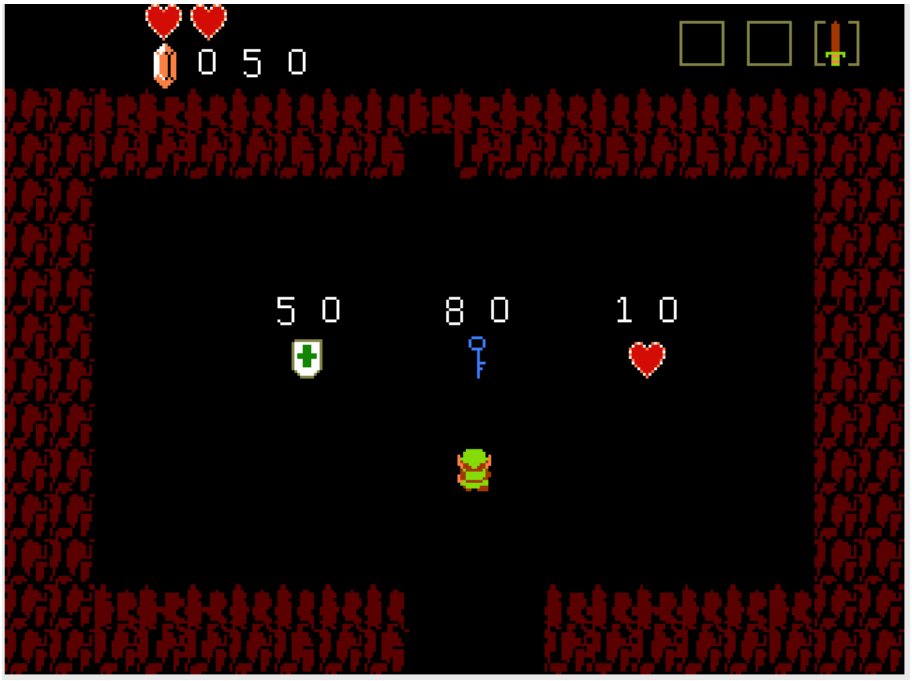

Figura 3. Interior da Loja. O sistema detecta a interação e manipula as flags de inventário na memória estática.

4. Conclusão

O desenvolvimento de *Assembler's Realm* permitiu a consolidação prática dos conhecimentos de arquitetura de computadores. A necessidade de gerenciar explicitamente a preservação de registradores na pilha (*stack*), calcular endereços efetivos e sincronizar periféricos demonstrou a complexidade oculta por trás das *engines* modernas.

Além do rigor técnico, o projeto promoveu um resgate histórico. A recriação de mecânicas da era 8-bits em baixo nível aprofundou o respeito pela engenharia de software clássica, unindo o aprendizado técnico à valorização da cultura dos videogames. Conclui-se que o Assembly, embora desafiador, oferece um controle inigualável sobre o hardware simulado.

Referências

Patterson, D. A. and Hennessy, J. L. (2017). *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Morgan Kaufmann.

Waterman, A. and Asanović, K. (2019). *The RISC-V Instruction Set Manual. Volume I: User-Level ISA*. RISC-V Foundation.

Lamar, M. V. (2025). Especificação do Projeto The Legend of Zelda - Introdução aos Sistemas Computacionais. Universidade de Brasília.

RARS (2024). RISC-V Assembler and Runtime Simulator. Disponível em: https://github.com/TheThirdOne/rars.