Desenvolvimento de um jogo de *Pokemon* em *Assembly* RISC-V

Felipe Nascimento Rocha

***Abstract*—Este artigo descreve a criação de um jogo de *Pokemon* programado em *Assembly* RISC-V, usando o RARS (RISC-V Assembler and Runtime Simulator) como ferramenta de desenvolvimento.**

***Index Terms*—RISC-V, RARS, Assembly, Pokémon, desenvolvimento de jogos, modularidade.**

I. Introdução

O Jogo desenvolvido é uma versão simplificada de um jogo de *Pokemon* tradicional, com um sistema de batalhas, lutas em ginásios e escolhas de *Pokemons*. O uso do *RARS* permitiu a utilização de recursos de depuração e verificação de códigos e a utilização de elementos de entrada e saída como o *Bitmap Display* e o *Keyboard and Display MMIO Simulator* para visualização e interação do jogo.

O artigo detalha algumas das etapas do desenvolvimento como a criação de cenários, animações, mecânicas de jogo, e os desafios enfrentados e soluções encontradas durante o processo de programação usando a arquitetura RISC-V [5]. O objetivo do artigo é apresentar uma abordagem para programar em *Assembly* RISC-V, além de demonstrar a possibilidade de criar um jogo usando essa tecnologia.

II. Modularidade

O desenvolvimento foi feito de forma modular [1], com cada *Pokemon*, ataque, item, inimigo, inventário e outros dados sendo armazenado em um arquivo separado. Esse tipo de abordagem permitiu uma maior flexibilidade no desenvolvimento do jogo em partes separadas, uma vez que novos elementos e funcionalidades poderiam ser adicionados facilmente sem a necessidade de modificar o código em outros lugares.

Para implementar essa estratégia, utilizou-se uma lista de *half-words* adequada para armazenar as informações de cada elemento do jogo. Por exemplo, para *Pokemons*, cada um é armazenado em um *array* de dados onde cada *Pokemon* possui um índice: 0, 1, 2, 3. Esse arquivo tem o nome *pokemon_index.s*, e é uma lista de *half-words* e possui:

- *Pokemon* index - Posição 0
- Pontos de Vida atual - Posição 2
- Pontos de Vida total - Posição 4
- Tipo do *Pokemon* - Posição 6
- Ataques 1, 2, 3 e 4 - Posição 8, 10, 12, 14

Ao iniciar o jogo, o programa lê esses arquivos de dados e carrega as informações de cada elemento para a memória quando necessários enquanto a manipulação desses dados é feita no decorrer de eventos dentro do jogo.

Além disso, essa abordagem também tornou a manutenção do jogo mais fácil, uma vez que os dados poderiam ser atualizados ou corrigidos sem afetar outros elementos. Isso significa que, se houvesse um erro em um *Pokemon* específico, por exemplo, era possível corrigi-lo simplesmente atualizando o arquivo de dados e testando-o, sem precisar alterar o código do jogo em outras partes.

III. Sistema de Batalhas

O sistema de batalha foi implementado com grande atenção aos detalhes por ser uma parte fundamental do jogo de *Pokemon*. A implementação é dividida em quatro opções: atacar, abrir inventário, trocar de *pokemon* e fugir. Para permitir que o jogador escolha entre essas opções, foi criado um menu principal, que permite alternar entre elas.

Fig. 1. Menu de Ataque no sistema de Batalhas.

Cada uma dessas opções funciona com um *loop* próprio que gera uma interface específica. Por exemplo, quando o jogador escolhe a opção de ataque, é gerado um novo menu com os ataques disponíveis do *pokemon* (que são buscados da base de dados correspondente àquele *pokemon*) em questão.

Outro elemento chave que foi implementado no sistema de ataque é a mecânica de vantagem entre tipos. Cada ataque é atribuído a um tipo específico, como fogo, água, grama, etc. Esses tipos têm vantagens e desvantagens quando usados em *pokemons* de determinados tipos, o que significa que certos tipos são mais eficazes contra alguns *Pokemons*. O cálculo de dano é feito da seguinte maneira:

Seja D dano total, T o tipo de vantagem do ataque sobre o *pokemon* e P o valor de ataque do ataque utilizado:

$$D = T \times P$$

Onde T varia entre 0.5 quando o ataque possui desvantagem sobre o tipo do *Pokemon* inimigo, 1 quando não há vantagem

nem desvantagem, e 1.5 quando o ataque possui vantagem sobre o tipo do *Pokemon* inimigo.

Para economizar performance, não foram utilizados *floats* na implementação do jogo, logo, esse cálculo foi realizado utilizando valores como 50, 100 e 150 e depois divididos como inteiros, desprezando o resto.

A inteligência artificial do inimigo seleciona seus ataques aleatoriamente, mas *pokemons* inimigos tendem a ter mais vida total o que deixam as batalhas equilibradas.

O sistema de inventário permite ao jogador utilizar itens. Durante a implementação desse sistema, foi criada uma lista de *half-words* que determinam quais itens estão disponíveis no inventário do jogador. Cada item possui um índice e a quantidade de pontos de vida que o item irá curar no *pokemon* atual em combate. Essa informação é suficiente para atualizar a vida do *pokemon* em batalha.

O sistema de troca permite ao jogador selecionar e alternar entre diferentes *pokemons* durante as batalhas. Foi criada uma lista que mostra todos os *pokemons* disponíveis para o jogador naquele momento. Essa lista é atualizada durante a fase de seleção.

A partir dessa lista, é possível gerenciar um dado que indica qual pokemon está sendo usado atualmente em batalha (*current_pokemon*) e alterá-lo para o outro pokemon selecionado. Esse dado de *pokemon atual* é usado para renderizar os *sprites* e ataques dos *pokemons* durante as batalhas, garantindo que o pokemon correto seja exibido na tela.

IV. Animação e *Sprites*

Para animar o personagem principal, são utilizados dados que contêm informações sobre a sua direção e o seu *frame*. Cada vez que o personagem se move, um novo *frame* é acrescentado, e a direção é alterada quando o personagem muda de direção.

Para que o personagem se mova pelo mapa, foi criado um procedimento que calcula o quadrado 16x16 do mapa atual que corresponde à última posição do personagem. Em seguida, esse quadrado é renderizado, sendo impresso em cima de onde o jogador estava no movimento anterior.

Os *sprites* de *background* são renderizados abaixo de todas as outras camadas de *sprites* no momento em que o jogador é teleportado, permitindo que os elementos do jogo sejam exibidos corretamente na tela e economizando performance.

Para mudança de mapas, foi criada uma lista de dados que descreve todos os teleportes disponíveis no mapa atual. Essa lista contém informações sobre quantos teleportes o mapa possui, onde o teleporte fica, qual mapa o personagem deve ser teleportado e em que posição ele deve ser colocado.

Quando o jogador se posiciona em um local de teleportação, o jogo utiliza essas informações para atualizar a posição do personagem no mapa e para exibir uma animação de transição para o novo mapa.

Também foi implementado um sistema que imprime uma caixa de texto semelhante a do jogo de *Pokemon*, essa caixa de texto foi utilizada para contar a história e interagir com o jogador caso necessário alguma informação de jogabilidade.

Fig. 2. Caixa de diálogos no início do jogo.

V. Trabalhos Futuros

Uma possível melhoria para o projeto apresentado seria a implementação de um algoritmo que automatizasse a geração de *tiles* em um mapa, reduzindo o tamanho de cada *background* e melhorando a performance do jogo. Atualmente, a impressão dos sprites é feita em todos os pixeis (*320x240*), o que demanda um grande número de *bytes* na memória. Com a utilização de *tiles* menores, seria possível imprimir mais informações na tela em menos espaço e consequentemente, possibilitar sua execução na placa.

Outra melhoria ainda necessária é o sistema de colisões que por enquanto, só checa os limites horizontais e verticais do mapa, elementos do jogo podem ser atravessados o que tira a imersão do jogador.

VI. Dificuldades e Conclusão

O desenvolvimento de um jogo em *assembly* é um projeto desafiador e envolve muitas complexidades e dificuldades. Uma das principais dificuldades encontradas foi a quantidade de trabalho necessário para ser feito por apenas uma pessoa. O desenvolvimento de um jogo requer muitas habilidades e conhecimentos diferentes.

Outra dificuldade enfrentada durante o projeto foi a necessidade de chamar procedimentos dentro de procedimentos, o que pode levar a uma série de problemas de gerenciamento de memória e controle de fluxo que foi solucionada com o aprendizado e uso de *stack pointers* corretamente.

Porém, a maior dificuldade encontrada durante o desenvolvimento foi o gerenciamento de memória. Como a quantidade de memória disponível na placa é limitada, seria necessário encontrar maneiras eficientes de economizar a memória utilizada pelo jogo. A implementação de certos elementos do jogo como a animação dos *sprites* e *backgrounds* exige o uso de grandes quantidades de memória, logo, o uso de memória excedeu (e muito) o limite da placa.

Por fim, a leitura do código em *assembly* é uma tarefa difícil, uma vez que o código é muito diferente do código de linguagens de programação de alto nível.

O projeto foi um desafio que envolveu vários fatores. Durante o processo, foram encontradas várias dificuldades. No entanto, com o uso de técnicas e estratégias adequadas e esforço, foi possível superar alguns desafios e criar um

jogo semelhante ao original. Acredito que com mais tempo seria possível implementar formas mais eficientes de gerenciar memória para sua execução na placa.

REFERENCES

[1] *Pokemon RISC-V*. GitHub. [Online]. Available: https://github.com/felipenrocha/pokemon-riscv

[2] *Tradutor MIDI RISC-V*. GitHub. [Online]. Available: https://github.com/Zen-o/Tradutor_MIDI-RISC-V

[3] *Gerenciador de Conversão*. GitHub. [Online]. Available: https://github.com/gss214/Gerenciador-de-Conversao

[4] *Slides da Sala*.

[5] *The RISC-V Instruction Set Manual*. Volume I: User-Level ISA . [Online]. Available: https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf