

Metroid em Assembly RISC-V

Atila 190103035 *

João 211042757 †

Lucas 180022563 ‡

Nathalia 170153169 §

Universidade de Brasília, 15 de setembro de 2024

RESUMO

Nesse artigo é descrito o jogo realizado na disciplina de OAC com tema metroid, implementado em Assembly RISC-V e executado em um FPGA. O código foi feito de forma que certos trechos da memória iriam ditar o comportamento da fase semelhante a uma engine de jogo, dessa forma aumentando o grau de abstração durante o desenvolvimento do projeto.

Palavras-chave: Assembly RISC-V · FPGA · Metroid · Jogo ·

1 INTRODUÇÃO

O projeto final da disciplina de Organização e Arquitetura de Computadores (OAC) envolveu o desenvolvimento de um jogo inspirado no clássico Metroid, utilizando a linguagem Assembly RISC-V. A implementação foi realizada em uma FPGA, proporcionando uma interação direta com o hardware e uma experiência prática. O objetivo principal foi aplicar conceitos de arquitetura de processadores e programação em Assembly, resultando na criação de um jogo funcional que integra elementos gráficos e de jogabilidade. O código foi estruturado de forma modular, permitindo a fácil manipulação dos objetos do jogo e a adaptação dos comportamentos por meio de ajustes em variáveis de memória.

Segue os requisitos que foram atendidos e as seções do texto que falam sobre tais requisitos caso hajam

1. Ataques do jogador - Seção 3.15
2. Movimentação e animação do(s) personagen(s) jogáveis - Seção 3.1 - 3.4
3. Mínimo de 2 itens, com pelo menos 1 permitindo o acesso a lugares novos e pelo menos 1 impactando o combate - Seção 3.6
4. Informações sobre a vida e equipamento da Samus - Seção 3.16
5. Pelo menos 3 salas distintas, isto é, três ambientes separados por uma porta.
6. Mínimo de 3 tipos de inimigos diferentes com IA (número de inimigos em aberto), sendo um deles um chefão - Seção 3.7
7. Background móvel que acompanhe o movimento da Samus (horizontal ou vertical) - Seção 3.3

O requisito 5 está presente no jogo, porém não há o que pode ser escrito a respeito.

2 FUNDAMENTAÇÃO TEÓRICA

Este projeto está fundamentado em conceitos-chave da arquitetura de computadores, programação em baixo nível e a integração de software com hardware. A escolha de um processador RISC-V como base para o desenvolvimento do jogo proporciona um contexto prático para a aplicação desses conceitos, permitindo explorar desde a codificação em assembly até a simulação em um FPGA.

2.1 ARQUITETURA RISC-V

O processador RISC-V (Reduced Instruction Set Computer) é uma arquitetura aberta e extensível, amplamente utilizada em pesquisas acadêmicas e desenvolvimentos industriais. Ele se destaca por sua simplicidade, modularidade e eficiência, o que o torna uma excelente escolha para projetos que exigem baixo consumo de recursos e alta performance.

2.2 PROGRAMAÇÃO EM ASSEMBLY

A linguagem assembly utilizada no desenvolvimento do jogo é um exemplo clássico de programação em baixo nível. Diferente das linguagens de alto nível, onde o compilador gerencia a abstração do hardware, o assembly permite o controle direto sobre o processador, registradores e memória. Isso possibilita a criação de instruções altamente otimizadas, adequadas para a execução em sistemas embarcados, como o FPGA.

No contexto do jogo, a programação em assembly foi utilizada para controlar a movimentação do personagem, as interações com o ambiente e a inteligência artificial dos inimigos. Cada uma dessas funcionalidades foi implementada com foco na eficiência, aproveitando os recursos limitados de hardware e a necessidade de otimizar o tempo de execução.

2.3 SIMULAÇÃO EM FPGA

Os FPGAs (Field Programmable Gate Arrays) são dispositivos programáveis que permitem a implementação de arquiteturas personalizadas diretamente no hardware. A utilização de um FPGA no projeto permitiu a execução do jogo em um ambiente realista, simulando o comportamento do processador RISC-V em um sistema físico.

2.4 QUARTUS E RARS

O Quartus Prime é uma IDE criada pela Intel para a programação de dispositivos lógicos programáveis como FPGAs. Ele oferece ferramentas para a síntese de HDL (Hardware Description Language), simulação, análise e implementação de circuitos digitais, permitindo aos usuários projetar e testar circuitos personalizados em hardware. O Quartus Prime é utilizado em projetos que envolvem sistemas embarcados e circuitos digitais de alta complexidade, oferecendo um ambiente completo para o desenvolvimento e depuração de designs de hardware.

O RARS (RISC-V Assembler and Runtime Simulator) desempenhou um papel crucial na criação dos arquivos .mif, que contém o código do jogo e seus dados.

*190103035@aluno.unb.br

†211042757@aluno.unb.br

‡180022563@aluno.unb.br

§170153169@aluno.unb.br

2.5 CONCEITOS DE COLISÃO, IA E MOVIMENTAÇÃO

Os mecanismos de colisão e inteligência artificial (IA) foram implementados utilizando conceitos simples, como a verificação de intersecção de bounding boxes e a distância entre personagens. A movimentação do personagem principal, Samus, e dos inimigos é controlada diretamente pelos valores armazenados nos registradores e arrays de memória, permitindo a detecção de barreiras, ações de ataque e deslocamentos automáticos baseados em IA.

3 METODOLOGIA

3.1 MECÂNICAS DE MOVIMENTAÇÃO DO JOGO

No jogo em Assembly RISC-V inspirado no clássico Metroid, a movimentação do personagem principal (Samus), dos personagens adversários e da tela é uma parte essencial da dinâmica do jogo. As mecânicas são implementadas utilizando uma série de arrays que armazenam variáveis como posição, profundidade do mapa e direção dos personagens.

3.2 MOVIMENTO DO PERSONAGEM PRINCIPAL

O movimento do Samus é gerenciado pelas teclas de controle. As teclas "a" e "d" controlam o movimento horizontal, movendo o personagem para a esquerda ou direita, respectivamente, em incrementos de 4 unidades. Já a tecla "w" é responsável pelo salto, que ajusta o registrador $a1$ para um valor positivo, permitindo que o personagem suba no eixo vertical. Quando o valor de $a1$ é positivo, a subrotina JUMP é acionada, permitindo que o personagem se move para cima. Caso $a1$ for 0, a gravidade é simulada pela subrotina GRAVITY, que faz o Samus "cair", incrementando o valor de $a0$ quando não há barreiras abaixo dele. O movimento horizontal é determinado diretamente pela posição do Samus na tela, enquanto o movimento vertical é percebido pelo deslocamento do mapa.

3.3 MOVIMENTO DO MAPA

O mapa se desloca verticalmente para dar a impressão de que o Samus está se movendo para cima ou para baixo, o que é realizado alterando o valor do registrador $a0$ que corresponde à profundidade do mapa. Cada pixel do mapa é desenhado em relação a esse registrador, fazendo com que o cenário se move para acompanhar a posição do Samus. O registrador $a0$ é incrementado ou decrementado dependendo da ação realizada, como a queda do Samus (incremento) ou um salto (decremento).

3.4 MOVIMENTO DOS PERSONAGENS INIMIGOS

A movimentação dos inimigos é governada por rotinas de inteligência artificial (IA). Existem três tipos de IA implementadas:

1. Movimento Horizontal: O personagem se desloca em um eixo horizontal com base em um ponto central, utilizando as variáveis AUX_AI0, AUX_AI1, e AUX_AI2 para determinar o alcance do movimento, a posição atual em relação ao ponto central e a direção de deslocamento.
2. Movimento Vertical: De maneira similar ao movimento horizontal, a IA do tipo vertical controla o deslocamento dos personagens em relação ao eixo y, com as mesmas variáveis ajustando o alcance e a direção do movimento.

3. Perseguição do Samus: A terceira IA faz com que o personagem persiga o Samus, ajustando sua posição vertical e horizontal de acordo com a localização atual do Samus na tela. Este comportamento simula uma perseguição contínua, onde o personagem ajusta sua posição para se alinhar com o protagonista, movendo-se em passos de 4 unidades.

3.5 POWER-UPS E INIMIGOS

No jogo, os power-ups e inimigos desempenham papéis fundamentais na progressão e desafio das fases. A mecânica de ambos é vinculada ao sistema de colisão e ao controle do personagem principal, sendo influenciados por interações diretas e pela inteligência artificial dos inimigos.

3.6 POWER-UPS

Os power-ups disponíveis no jogo são gerenciados pelo registrador $s2$, que indica o tipo de item que o Samus possui. Existem três estados possíveis para esse registrador:

1. Sem item ($s2 = 0$): Nesse estado, o Samus não possui nenhum power-up ativo.
2. Item de ataque ($s2 = 1$): Ao adquirir o item de ataque, Samus aumenta seu alcance de tiro de 60 para 92 unidades, permitindo derrotar inimigos a uma distância maior.
3. Item de fase secreta ($s2 = 2$): Este power-up permite que o Samus desbloqueie áreas escondidas ao colidir com blocos específicos no mapa. Esses blocos, que de outra forma seriam intransponíveis, dão acesso a fases secretas quando o jogador possui esse item.

3.7 INIMIGOS

Os inimigos são uma parte central da mecânica de jogo, desafiando o jogador com diferentes padrões de comportamento, definidos por três tipos de inteligência artificial (IA):

1. Inimigos de Movimento Horizontal: Esses inimigos patrulham uma área específica movendo-se da esquerda para a direita e vice-versa, conforme definido pelas variáveis AUX_AI0 (alcance), AUX_AI1 (distância percorrida) e AUX_AI2 (direção). A posição desses inimigos no mapa é ajustada constantemente, mantendo-os em movimento contínuo.
2. Inimigos de Movimento Vertical: Similar aos inimigos horizontais, mas com deslocamento ao longo do eixo vertical. Eles sobem e descem em uma área determinada, também utilizando as variáveis auxiliares para controlar alcance e direção.
3. Inimigos Perseguidores: Esses inimigos são programados para seguir o Samus, ajustando sua posição com base nas coordenadas atuais do jogador. Eles se movem tanto horizontal quanto verticalmente para manter a perseguição, utilizando uma abordagem simples, onde o deslocamento é feito em incrementos de 4 unidades por vez, tornando-os ameaças persistentes ao longo do jogo.

Adicionalmente, o jogo implementa uma mecânica específica para o boss, que está presente na última fase. Ao derrotar um personagem da fase do boss, um novo personagem é ativado (é colocado como 1 no array ALIVE na posição correspondente), simulando uma mudança de comportamento do inimigo, sem alterar sua aparência (usa-se sempre o mesmo ".data"), mas modificando seus padrões de IA. Isso cria a sensação de que o mesmo inimigo executa novas estratégias à medida que o jogador progide no confronto.

3.8 MECANISMO DE COLISÃO

O mecanismo de colisão no jogo é essencial para a interação entre o personagem principal, Samus, e os elementos do cenário, incluindo inimigos e power-ups. A colisão é tratada de forma eficiente e direta, com base na posição dos personagens e itens na tela, bem como nas variáveis de controle que determinam a ativação dessas interações.

3.9 COLISÃO COM INIMIGOS

A colisão com os inimigos é avaliada apenas se a variável ALIVE do inimigo estiver configurada como 1, indicando que o personagem está ativo no jogo. Caso o valor de ALIVE seja 0, o inimigo é considerado inativo e, portanto, não participa das rotinas de detecção de colisão nem é exibido na tela. Essa verificação inicial permite otimizar o processamento, evitando cálculos desnecessários para personagens já derrotados.

A colisão propriamente dita ocorre com base na intersecção dos "bounding boxes" de 16x16 pixels que delimitam tanto o Samus quanto os inimigos. O jogo compara as posições salvas nos arrays POSITIONX e POSITIONY para verificar se há sobreposição entre as caixas que cercam o Samus e o inimigo. Se houver intersecção, o jogo registra a colisão então o Samus perde uma vida e é repositionado no início da fase.

3.10 COLISÃO COM POWER-UPS

Quando o Samus entra em contato com um item no cenário, o jogo detecta a colisão da mesma forma que com os inimigos e blocos, verificando a intersecção das caixas delimitadoras de 16x16 pixels. Se a colisão for detectada, o valor do registrador s2 é atualizado para refletir o novo power-up adquirido, alterando as habilidades ou acesso do Samus conforme necessário. O valor de ALIVE correspondente ao Power-Up é colocado como 0.

3.11 CORES E BLOCOS DO JOGO

No jogo, os blocos e cores desempenham um papel importante na definição das barreiras, trocas de fase e áreas secretas. No entanto, o mecanismo de detecção desses elementos difere do utilizado para colisões com inimigos e power-ups. Em vez de uma verificação de intersecção de bounding boxes, a lógica é baseada na análise da cor dos pixels na vizinhança de posição do Samus, sendo que essa cor determina a ação a ser tomada pelo jogo.

3.12 BLOCOS

Os blocos são responsáveis por criar barreiras no mapa e controlar a transição entre diferentes fases. Eles são representados pelos arrays POSITIONX e POSITIONY, que armazenam suas posições no cenário, e pelo array TYPE, que define o tipo de cada bloco. Existem três tipos principais de blocos no jogo:

1. Bloco de Barreira (TYPE = 0x100): Esses blocos criam obstáculos físicos no cenário, impedindo o movimento do Samus. Ao tentar se mover para uma posição onde a cor do bloco corresponde a uma barreira, o jogo interpreta isso como uma colisão com o bloco e impede o deslocamento do Samus.
2. Bloco de Troca de Fase (TYPE = 0x101): Esses blocos permitem a transição para uma nova fase. Quando o Samus entra em contato com uma cor correspondente à troca de fase, o jogo inicia a transição para o próximo estágio.
3. Bloco de Fase Secreta (TYPE = 0x102): Esses blocos são semelhantes aos de troca de fase normal, mas só são ativados

se o Samus possuir o item de fase secreta (s2 = 2). Quando o Samus colide com a cor correspondente e tem o item, ele é transportado para uma fase secreta do jogo.

Cada fase contém 10 blocos, e suas posições são definidas pelos arrays POSITIONX e POSITIONY. O jogo compara a posição do Samus com a dos blocos para determinar se há uma interação. No entanto, essa verificação é feita com base na cor do pixel na tela, somada a 16 unidades no eixo y (no caso de movimento vertical) ou à esquerda/direita (no caso de movimento horizontal), para determinar se o Samus colidiu com um bloco ou transicionou para outra fase.

3.13 CORES

As cores no jogo servem para identificar zonas específicas no mapa que causam diferentes efeitos quando o Samus entra em contato com elas. As cores definem:

1. Barreira: Determina se o Samus pode ou não continuar seu movimento. Se a cor detectada na posição do Samus for a de uma barreira, o jogo impede seu avanço.
2. Troca de Fase Normal: Se o Samus tocar em uma cor correspondente a uma troca de fase, ele é movido para a próxima fase.
3. Troca de Fase Secreta: Caso o Samus toque em uma cor de fase secreta e possua o item correto, ele será levado a uma fase escondida.
4. Zona de Morte: No fim de cada fase, existe uma cor especial colocada para marcar uma área de morte. Se o Samus cair para além dos limites do mapa e tocar nessa cor, ele perde uma vida e retorna ao início da fase.

Essas cores são verificadas a cada movimento do Samus no eixo vertical ou horizontal, garantindo que o jogo responda de maneira adequada às interações com o cenário.

3.14 ARTE

Foi extraído as imagens de variadas fontes da internet (sites de sprite, inteligência artificial, mecanismo de busca). Para fazer os ".data" foi necessário um procedimento de três etapas:

1. Ajustar para o tamanho correto com a função resize do imagemagick (mapa de largura 320 e personagens de dimensão 16x16)
2. Converter para bmp com o gimp
3. Usar o programa disponibilizado no github chamado BMPto-ASM para converter de .bmp para .data

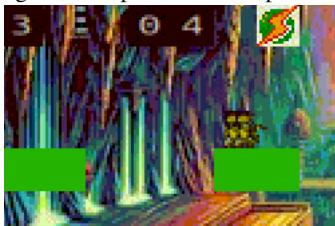


3.15 MECÂNICA DE ATAQUE

A mecânica de ataque é ativada quando o jogador pressiona a tecla "k". Nesse momento, o algoritmo calcula a distância L1 entre Samus e os inimigos presentes no mapa. Caso essa distância seja inferior a 60 unidades (ou 92 se o jogador possuir o item de ataque), o inimigo é eliminado, e seu respectivo valor no array ALIVE é alterado de 1 para 0, removendo-o do jogo.

3.16 INFORMAÇÕES DO SAMUS

Para evitar a criação de código desnecessário, a rotina PRINT_CHAR é reutilizada para exibir, no canto superior esquerdo do display, a quantidade de vidas (representada por um dígito), mísseis (dois dígitos) e o item que Samus possui (representado pela imagem do respectivo item capturado).

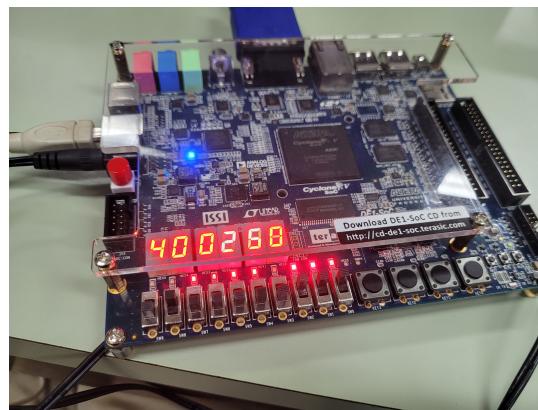


Para exibir essas informações, foram definidos 10 valores no segmento .data, correspondentes aos dígitos de 0 a 9, que são salvos no array TYPE como o valor do dígito somado a 64. Em hexadecimal, os dígitos de 0 a 9 correspondem aos valores de TYPE entre 0x40 e 0x49. A quantidade de vidas é salva na posição correspondente do array, somando-se o valor da registradora s4 a 64 (0x40). Da mesma forma, os dois dígitos que representam a quantidade de mísseis são extraídos da registradora a5, somados a 64, e salvos no array TYPE, para que sejam exibidos corretamente no display.

O item que Samus possui é representado pelos valores de TYPE que variam de 0x80 a 0x82, correspondendo a: nenhum item (0x80), item de ataque (0x81) e item de fase secreta (0x82). O valor do item, armazenado na registradora s2, é somado a 128 (0x80) e salvo em uma posição específica do array TYPE, permitindo sua exibição no display.

4 RESULTADOS OBTIDOS

Para executar o jogo em um FPGA, foi utilizado o arquivo "RV32IMF-Multiciclo.sof", que simula um processador RISC-V multiciclo e está disponível na plataforma Aprender3. Com o auxílio do software Quartus, o processador foi carregado no FPGA, juntamente com os arquivos "game_data.mif" e "game_text.mif", gerados previamente pelo programa RARS a partir do código assembly do jogo. Esses arquivos .mif contêm tanto os dados quanto o código do programa. Após a configuração, o jogo pôde ser visualizado e jogado através de um monitor conectado ao FPGA, demonstrando a funcionalidade correta da simulação e execução do jogo no hardware.



5 CONCLUSÕES E TRABALHOS FUTUROS

No decorrer do projeto houveram problemas na execução do jogo no FPGA devido a limitação de memória. Isso permitiu que fosse utilizado apenas um background o que tornou-se necessário o uso de tiles para gerar mapas diferentes. Uma memória um pouco melhor levaria a não ser necessário o uso de tiles, já que somente as cores no mapa são utilizadas para detectar troca de fase, barreira, etc. Talvez uma versão futura possa seguir da maneira idealizada

O fato de ser necessário apenas alterar trechos de memória para gerar novas fases tornou possível o desenvolvimento rápido das fases do jogo que poderão ser feitas caso haja uma versão futura do jogo.

A experiência demonstrou o sucesso da implementação de um jogo em RISC-V e a viabilidade do uso de processadores RISC-V para simular jogos em hardware real, com uma integração sólida entre software e hardware.

REFERÊNCIAS

- [1] Waterman, A., Lee, Y., Patterson, D., Asanović, K. (2011). The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA. EECS Department, University of California, Berkeley.
- [2] Patterson, D. A., Hennessy, J. L. (2017). Computer Organization and Design RISC-V Edition: The Hardware Software Interface. Morgan Kaufmann.