Relatório do Trabalho Final de OAC

Andreza Rodrigues Costa* Pedro Trajano Ferreira† Rodrigo Silva Lopes Zedes‡
Sabrina Carvalho Neves§ Thiago Tomás de Paula¶

Universidade de Brasília

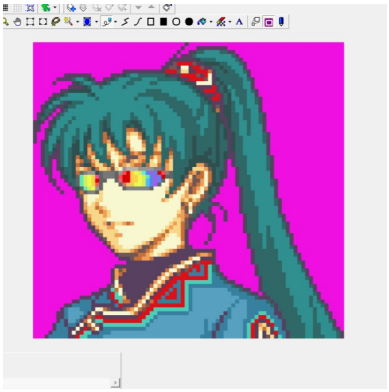


Figura 1: Lyn de juliete. Julynete.

RESUMO

O trabalho final do curso busca conciliar de maneira prática as duas principais áreas do conhecimento abordadas no curso: processadores e linguagem Assembly RISC-V. Neste semestre, fizemos um jogo baseado no Fire Emblem 7 para o GBA, que deve ser executado pela FPGA; é imposto ao jogo uma sequência de requisitos, e é o objetivo deste documento mostrar quais são esses requisitos e como foram atendidos. Ao final, discutimos a performance do jogo dentro e fora da FPGA, onde concluímos que o projeto foi bem sucedido. Finalmente, levantamos alguns pontos a serem melhorados ou que não puderam ser implementados devido a limitações de tempo e memória.

Palavras-chave: FPGA · OAC · RISC-V · Fire Emblem 7

1 INTRODUÇÃO

O curso de Organização e Arquitetura de Computadores foca de maneira geral em dois assuntos, que são tratados em duas metades do curso: como processadores são arquitetados para lidar com instruções, e como as instruções são organizadas.

As arquiteturas de processador são exploradas na segunda parte da matéria, principalmente através de aulas em laboratório, enquanto a organização das instruções é tratado num primeiro momento, via aulas teóricas.

O trabalho final da disciplina busca cobrar o entendimento do aluno acerca de ambas área do conhecimento citadas por meio de um jogo feito em Assembly RISC-V (32 bits) que deve ser executável na FPGA. Por Assembly RISC-V, entende-se a linguagem de baixo nível criada por Andrew Waterman et. al em 2011. A FPGA é um conjunto de elementos lógicos programável pelo usuário que no caso deste trabalho foi configurada como um processador Pipeline com módulo de operações em ponto flutuante.

O objetivo deste documento é relatar, de maneira breve e geral, a estratégia adotada para criar o jogo dadas suas restrições e requerimentos.

Contextualizado o trabalho final, na próxima seção apresentamos os requerimentos do jogo, para eventualmente evidenciar como elas foram atendidas.

2 FUNDAMENTAÇÃO TEÓRICA E TRABALHOS RELACIONADOS

O projeto deve-se basear no jogo Fire Emblem 7: the Blazing Blade, e atender a diversos requerimentos. Estes são divididos em 11 tópicos no roteiro, a saber,

1. história do jogo;
2. música e efeitos sonoros;
3. mínimo de 5 telas jogáveis distintas, que podem ou não ser estáticas, com espaços abertos e paredes;
4. cursor, áreas de movimentação e ataque dos personagens;
5. fases com número crescente de inimigos e aliados (inclusa a animação deles).
6. IA que controla os inimigos e sistema de turnos;
7. pelo menos três tipos de armas: machado, lança e espada, levando em consideração o sistema de "pedra, papel e tesoura" relacionado a eles;
8. cenas de batalhas, podendo ser apenas uma imagem mostrando os personagens e as vidas deles;
9. menu de ações do jogador;

*18/0074474
†19/0055251
‡18/0139380
§17/0113973
¶19/0038641 – thiago.tomas@aluno.unb.br

10. movimentação dos personagens;

11. dois tipos de terrenos especiais.

Para o requerimento 3, usamos um entendimento um pouco solto de parede, mas que fora aprovado pelos monitores. Os outros requerimentos foram plenamente atingidos em seu entendimento natural, como veremos na próxima seção. Para finalizar esta seção, gostaríamos de citar (agradecer) alguns programas/tutoriais feitos por ex-alunos que foram muito úteis para a criação do jogo:

- Leonardo Riether – FPGRARS, eficiente simulador de código Assembly RISC-V;
- Gabriel B. G. e Davi Patury – Tradutor de midi para RARS;
- Davi Patury – Tutorial de Renderização dinâmica no Bitmap Display
- Victor Lisboa – Tutorial de convsersão de imagens para o RARS

3 METODOLOGIA

O objetivo desta seção é mostrar como cada requisito listado na seção anterior foi atingido, de forma que dedicamos uma subseção por item. Frequentemente apontaremos funções e diretórios presente no git do trabalho do grupo, mantido privado até a apresentação no dia 05/10/2022.

3.1 História do jogo (com caixas de dialogo)

No nosso jogo, os intrépidos personagens se deparam com um problema: estão sem sal grosso para o churrasco na praia de fim de semestre.

Este problema, logicamente, só pode ser resolvido obtendo sal do próprio Himalaia, trajeto longo que apresentará vários inimigos (veganos).

A história é desenvolvida (e novos personagens são apresentados) via diálogo entre a Carolina (garota de cabelo azul) e a Lyn (garota de cabelo verde).

Definimos um diálogo por fase, resultando em cinco diálogos, presentes nos módulos `DialogoX.s`, onde `X` indica a fase daquele dialogo. As linhas de diálogo em si são impressas pela função `printString`, presente no `FuncoesDoSystem.s`. Os módulos se encontram na pasta `fases`, e as falas em si (assim como toda string usada no jogo), em `falas`.

Figura 2: Primeira linha do diálogo da fase 2.

3.2 Música e efeitos sonoros

A maioria dos efeitos sonoros se encontram no ambiente de luta, e ocorrem quando: uma barra de vida (1 HP) é impressa, `midiHP.s`; uma barra de vida é tomada, `midiDano.s`; um ataque ocorre, `midiAtaque.s`. Em particular, um ataque pode ter três sons diferentes dependendo se (i) conectou; (ii) conectou e não foi crítico; (iii) conectou e foi crítico. Fora da luta, os feitos sonoros ficam no diálogo: `midiTexto.s` toca um som por caractere de fala impresso, tendo o símbolo > um som particular. A música toca apenas nas telas jogáveis, através do módulo `Musica.s`. Quando chamada, esta função verifica se uma nova nota deve ser tocada a partir do tempo de duração da última nota tocada; quando a música deva continuar, o instrumento é decidido de acordo com a fase atual.

Sejam efeitos sonoros ou música, os sons na FPGA são executados pela função `midiOut.s` explicitada no `FuncoesDoSystem.s`.

3.3 Telas jogáveis distintas

O percurso praia-Himalaia foi dividido em 5 fases: `Fase1.s`, na praia; `Fase2.s`, na floresta/mata; `Fase3.s`, no meio da estrada na floresta; `Fase4.s`, no fim da estrada na floresta, e; `Fase5.s`, no Himalaia.

Uma fase é concluída quando seu objetivo é atingido, isto é, todos os inimigos são eliminados. Por um lado, se todos os personagens no time do jogador são eliminados, ele perde o jogo e é dirigido, pela função `VerficiaGameOver.s`, para a tela de derrota, definida em `Derrota.s`

Figura 3: Tela de derrota. Da esquerda para a direita, temos os personagens Brigand, Soldier e Assassin.

Por outro lado, quando o objetivo da fase é concluído, a função `VerificaWin.s` automaticamente passa o jogador para a próxima fase, ou, caso a fase 5 tenha sido bem sucedida, para a tela de vitória, presente na função `Vitoria.s`.

Figura 4: Tela de vitoria. Da esquerda para a direita, temos os personagens Lyn, Dorcas, Sain, Dart e Yogi.

Tanto na tela de vitória quanto de derrota o usuário terá a opção de encerrar o jogo ou jogar novamente.

As paredes do nosso jogo são simplesmente as sprites de mar/-tubarão na primeira fase. Antes de ir para o próximo item, vale comentar que teria sido impossível implementar a sprite de 5 telas jogáveis (mapas) distintos de maneira séria sem usar a técnica

de tiling, onde alguns pequenos elementos visuais de mapa são repetidos para formar o mapa completo. Aqui, usamos tiles 16x16 para formar mapas de 320x240, o que é feito pela função `PrintTiling`. As tiles usadas, assim como os seus mapeamentos, se encontram na pasta `sprites`, em `tiles` e `tilemaps`, respectivamente.

Figura 5: Sprites de mar: as paredes do nosso jogo.

3.4 Cursor, áreas de movimentação e ataque dos personagens

A área de movimentação é o quadriculado azul e vermelho impresso ao redor do personagem jogável quando o usuário dá enter nele. As tiles azuis são alcançáveis e as vermelhas não. O `ImprimeAreaMovimentacao.s` cuida dessa funcionalidade, enquanto o cursor funciona graças ao módulo `MovimentaCursor.s`. Quanto a área de ataque, decidimos imprimir simplesmente um + vermelho quando na decisão do jogador de atacar um inimigo. Os inimigos em si não têm áreas de movimentação explícitas, mas possuem o mesmo alcance dos aliados.

Figura 6: Área de movimentação. Perceba o vermelho nos quadrados dos inimigos.

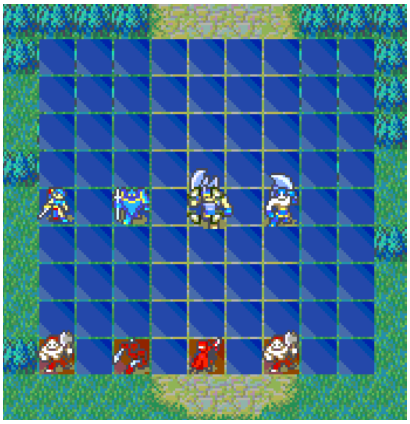
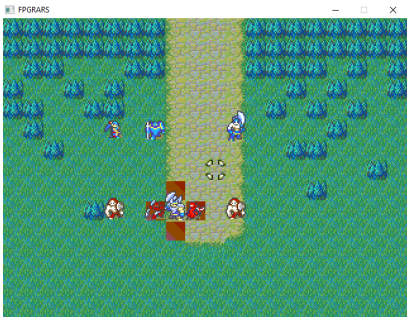


Figura 7: Área de ataque



3.5 Inimigos e aliados

A fase x possui $2x$ personagens, um jogável (aliado) e outro não (inimigo). Os aliados chamam-se Lyn, Yogi, Dorcas, Dart e Sain, e os inimigos, Brigand, Soldier e Assassin. Um novo aliado é introduzido por fase, acompanhado de um inimigo. Confira a seguir os dados de cada personagem, definidos no `.data` do `FireEmblem.s`.

Tabela 1: Designação dos *structs* do personagens.

Nome	HP	Arma	ID
Lyn	18	0	0
Brigand	20	1	1
Yogi	25	2	2
Soldier	20	2	3
Dorcas	20	1	4
Assassin	18	0	5
Dart	20	1	6
Brigand2	20	1	7
Sain	25	2	8
Soldier2	20	2	9

A animação dos personagens, majoritariamente, ocorre nas telas jogáveis, via `SetupMapa.s`. Este módulo será discutido a seguir; aqui, é interessante restringir o comentário aos grupos de funções `EscolheIdleX.s` e `ImprimeX.s`. O primeiro grupo é usado nas telas jogáveis, e decide qual sprite do personagem `X` deve ser impressa de acordo com seu estado (não mostrado na tabela 1); já o segundo grupo é particular às telas de vitória e derrota, e além de decidir a próxima pose do personagem `X`, já a imprime na tela. Também indo de encontro à animação nas telas jogáveis, a animação nas telas de vitória e derrota não ocorrem após 150 ms por sprite, mas após terem passados uma certa quantidade de tempo num certo estado (pose). Esta última técnica de animação fica mais próxima da adotada no Fire Emblem 7 e foi implementada na `AtualizaEstados.s`.

3.6 IA que controla os inimigos e sistema de turno

A IA desenvolvida pelo grupo é bastante simples, primeiro verificamos o inimigo mais próximo do primeiro personagem aliado e fazemos com que ele tente chegar perto dele e se a quantidade de movimentos necessária for menor que a área de movimento o inimigo vai o mais perto possível, também incluímos na inteligência do inimigo que se ele passar perto de um personagem aliado ele irá atacá-lo ao invés de continuar o movimento até o primeiro personagem aliado.

Essa lógica continua até todos os cinco personagens aliados forem atacados ou até todos os inimigos atacarem. Isso pode levar a alguns bugs e incoerências que não conseguimos resolver, como dois inimigos ficarem na mesma posição, mas isso não acarretou em uma falha catastrófica do jogo.

O esquema de turno funciona basicamente igual ao jogo original: os personagens ficam cinzas e quando todos os personagens aliados ou inimigos daquela fase ficarem cinzas o turno do adversário começa.

Para captar a mudança de turno, criamos a função `VerificaVez.s`, que passará o turno para a máquina caso a flag seja ativada. Em geral, o controle de todos os personagens ocorre no `SetupMapa.s` e suas funções satélite (`RecuperaSprite.s`, `SalvaSprite.s`, etc.).

3.7 Armas

Decidimos por utilizar 3 armas no jogo: a espada, o machado e a lança. Mantendo-nos fiéis ao Fire Emblem 7, a espada é forte contra

o machado e fraca contra a lança enquanto o machado é forte contra a lança e a lança é fraca contra o machado. Personagens com código de arma 0 possuem espadas, 1 possuem machados e 2 possuem lanças. Quando uma arma é forte contra outra, o dano base será 10; caso seja fraca, será 3; caso seja neutra (armas iguais), será 6. Se ainda houve a sorte de obter crítico, soma-se 5 ao ataque base; por outro lado, se houve o azar de obter erro, o ataque é anulado (mensagem "Miss!"). O valor do ataque é decidido em `Ataque.s`, e será influenciado pelo terreno especial, como veremos.

3.8 Cenas de batalhas

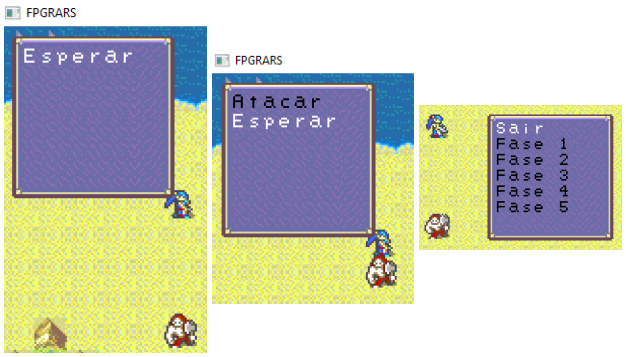
A batalha ocorre entre um personagem aliado e um personagem inimigo, de maneira estática (i. e., sem animação), conduzido turno a turno por mensagens, como definido em `Luta.s` e suas funções satélite. Cada personagem realiza 1 ataque (caso não morram antes), sendo o personagem que inicia a luta o primeiro a atacar. Para decidir o símbolo da arma do personagem assim como sua sprite de corpo inteiro e nome, foi necessário o ID na Tabela 1.

Figura 8: Cena/ambiente de luta. Perceba os símbolos das armas e as setas que indicam vantagem/desvantagem. O aliado fica sempre à esquerda.

3.9 Menu de ações do jogador

Excetuando-se os já citados menus de fim de jogo, existem 3 menus nas telas jogáveis: um padrão que oferece ao personagem apenas a opção de esperar (terminar sua vez); uma variação do padrão que pergunta ao personagem se ele deseja atacar um inimigo ao seu alcance, e; um menu de cheats que permite ao usuário pular para a fase desejada.

Figura 9: Da esquerda para a direita: menu padrão, menu de luta, e menu de cheats, acessado clicando-se na tecla P.



3.10 Movimentação dos personagens

A movimentação dos personagens é a mais simples possível: do lado aliado, dado que o cursor selecione uma tile viável, o personagem irá primeiro igualar sua posição em x àquela do cursor para então igualar a em y; do lado inimigo, cada personagem encontra primeiro o aliado mais próximo (vivo), para depois executar a mesma movimentação citada, trocando o cursor pelo aliado alvo. As funções `MontaMenuX.s` e `MovimentaMenuX.s` são responsáveis por imprimir a sprite dos menus e atualizar essa sprites de acordo com o comando de movimentação do usuário, nessa ordem.

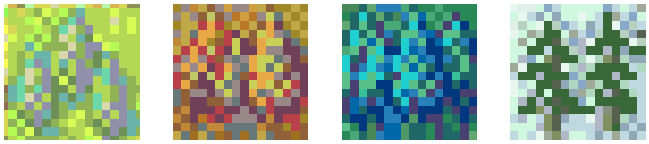
Percebe que esta movimentação simples só é possível se a distribuição de paredes for trivial, que é o caso no nosso jogo. Quando distribuição mais complicada implicaria diretamente num algoritmo de movimentação não trivial e potencialmente deselegante em resultados, como o DFS, BFS, etc.

3.11 Terrenos especiais

Existem dois tipos de terrenos especiais: árvores e montanhas. Quando o personagem se encontra sobre a tile de uma árvore, seu ataque (caso tenha conectado), diminui em 1 ponto; se dessa vez o personagem se encontra sobre a tile de uma montanha, seu ataque (caso tenha conectado) aumenta em 2 pontos. A ideia por trás desses efeitos é que as montanhas fornecem high-ground enquanto as árvores tiram visibilidade e bloqueiam o caminho. Por fim, embora isso não tenha sido considerado um terreno especial, vale citar que as animações de mapa da fase 5 são mais lentas que as das outras fases. Discutidos os requerimentos, gostaríamos de apresentar as técnicas usadas pelo grupo para otimizar a memória de dados.

Figura 10: Tiles de highground.

Figura 11: Tiles de floresta.



3.12 Economia da memória de dados

A FPGA tem uma memória de dados extremamente limitada: apenas 128 kiB. Dessa forma, precisou-se tomar algumas medidas de otimização de memória, na forma de duas funções comentadas abaixo.

3.12.1 Impressão dos mapas

Dadas as dimensões do bitmap, o natural seria criar mapas (telas jogáveis) de 320 bytes de largura por 240 bytes de altura, totalizando 76,8 kB ou 75 kiB. Este caminho é obviamente insustentável, e por causa disso o armazenamento de dados do mapa deve ser mudado. Seguindo a sugestão dos monitores, mudamos o estilo para o de tiling, onde um punhado de sprites pequenas são juntadas para formar o mapa maior. Por esse método, o mapa anterior pode ser feito usando, por exemplo, 5 tiles de 16 bytes de largura por 16 bytes de altura, totalizando 1,280 kB ou 1,25 kiB, isto é, uma redução de 98% no uso de memória. Com esse espaço liberado, não só o trabalho como um todo se torna viável como também podemos realocar dados para várias sprites mais ou menos decorativas, como os mugshots, sprites de corpo todo e os símbolos de armas nas lutas.

3.12.2 Impressão cinza dos personagens

Tentando ser fiéis ao jogo, nos deparamos com o problema de tornar uma sprite cinza. Como faríamos isso sem (i) refazer toda a sprite e (ii) tornar o resultado consistente entre personagens sem usar dons artísticos?

A solução estava em aprender como uma cor é codificada em byte na FPGA/RARS. Dado um valor RGB, o byte de cor correspondente β é

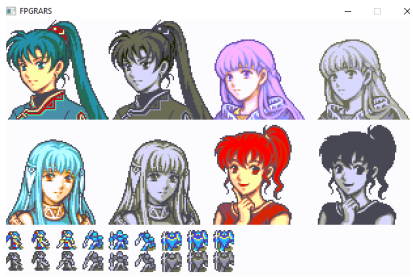
$$\beta = R//32 + [(G//32) \ll 3] + [(B//64) \ll 6].$$

Por exemplo, tomando o caminho contrário, a cor invisível $0xC7 = 11000111$ tem $R = 224$, $G = 0$ e $B = 192$. Note entretanto que o valor do RGB não é único; $R = 255$, $G = 0$ e $B = 255$ é outra tripla que gera o byte invisível.

Em todo o caso, eis como tornar um byte cinza: basta converter cada valor RGB para a média aritmética entre eles, seguindo a sequência "conversão byte-RGB $\rightarrow$ RGB normal fica cinza $\rightarrow$ conversão RGB-byte". É claro, o pixel invisível não deve ser acinzentado.

No código, as funções `ByteCinza`, `byte2RGB` e `RGB2byte`, cujos objetivos são claros pelas labels, é que realizam o serviço. Elas se encontram dentro do `PrintByte.s`. Confira a imagem a seguir que mostra algumas sprites antes e depois de serem acinzentadas.

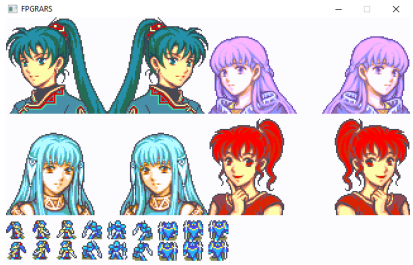
Figura 12: Efeitos do grayscale em algumas sprites.



3.12.3 Espelhamento horizontal de tiles

O jogador deve ter percebido que várias sprites são simétricas, como o retângulo decorativo na luta, ou a movimentação lateral dos personagens. Podemos aproveitar esse fato para evitar criar sprites espelhadas, o que foi feito na função `PrintByteInverso.s`, onde o label da imagem é impresso de maneira espelhada.

Figura 13: Efeitos da impressão inversa em algumas sprites.



4 Resultados Obtidos

Executando o `FireEmblem.s` criado pelo grupo tanto na FPGA quanto no FPGRARS, percebe-se que o jogo roda bem e de maneira idêntica (salva a menor frequência de clock da FPGA), exceto talvez na música. A FPGA toca a música de maneira mais "picotada" do que no FPGRARS, algo que o RARS também parece fazer. Não entendemos por que isso acontece, mas parece advir da forma como o sintetizador da placa foi feito. Talvez as notas tenham uma rápida queda de volume após sua duração programada. De maneira geral, consideramos o projeto final bem sucedido em implementação.

5 Conclusões e Trabalhos futuros

Como vimos na seção de metodologia, todos os requisitos apresentados na seção 2 foram atendidos plenamente, com direito a brindes como telas de vitória e derrota e personagens cinzas. Como comentado na seção anterior, a execução do programa na placa também foi bem-sucedida, apesar dela distorcer a música e ter pouca memória.

Ainda assim, deve-se deixar claro que o trabalho carece de algumas melhorias, principalmente o conserto do bug onde dois personagens inimigos podem ocupar a mesma tile.

Teríamos gostado também de criar pequenas cutscenes entre o fim de uma fase e o início da próxima, ao fim das quais o diálogo iniciaria. Nelas, os personagens aliados apareceriam no topo do mapa e desceriam até suas posições padrão.

Finalmente, também teria sido interessante obter uma variedade maior de músicas, e inicialmente tínhamos pensado em uma por fase, mas deixamos a ideia de lado para nos focar em outros problemas.

Referências

Além das citadas na segunda seção: aulas e slides do prof. Lamar, monitorias online e presenciais do curso.

Agradecimentos

Não fizemos as várias sprites que usamos no trabalho, todas foram obtidas a partir das PNGs neste site especializado. Também não fizemos a música, esta foi obtida neste site, e chama-se "Maiden of the dark - Idoun's theme".