Relatório sobre o Trabalho de Programação Assembly: Jogo 2048

Adriano Gomes da Silva[1], **Ricardo Turella**[1]

[1]Universidade de Caxias do Sul (UCS)
Caxias do Sul, Rio Grande do Sul, Brasil

{agsilva11,rturella3}@ucs.br

Resumo. *Este presente trabalho é um relatório sobre a implementação do jogo 2048 feita em linguagem de montador 8086 para a disciplina de Arquitetura de Computadores II.*

1. Problema

O jogo 2048 ocorre em uma matriz 4x4 (também chamada de tabuleiro, nesse trabalho). Inicialmente, todas as casas estão vazias, exceto duas aleatórias, que podem conter o valor 2 ou 4. O jogador realiza movimentos que movem todas as peças em uma das quatro direções: para cima, para baixo, para esquerda e para direita. Todas as peças são deslocadas para a direção escolhida e, quando duas peças de mesmo valor ficam adjacentes, elas são mescladas em uma única peça com o dobro do valor das duas. Se peças adjacentes não possuírem o mesmo valor, elas não serão mescladas e permanecerão no tabuleiro. Após o movimento, uma peça aleatória é colocada em uma casa vazia do tabuleiro. Essa peça pode ter valor tanto de 2 como de 4.

O jogador só pode selecionar uma direção em que pelo menos uma casa sofrerá alguma modificação. O objetivo do jogo é conseguir uma peça com valor de 2048. O jogo termina quando uma peça com 2048 é obtida, ou quando não há mais movimentos disponíveis.

A pontuação final é obtida pela soma de todas as peças mescladas do jogador. As 5 maiores pontuações são armazenadas nos recordes, juntamente com a quantidade de movimentos necessários para atingi-las e o nome do jogador que atingiu. O nome do jogador é solicitado quando ele atinge uma das cinco melhores pontuações. Os recordes podem ser acessados pelo menu principal do jogo.

Há a opção de jogar o jogo automaticamente, através do simulador. O número de vezes para se simular é solicitado. Ao final de todas as simulações, será exibido a menor quantidade de jogadas necessárias para atingir cada valor de peça igual ou maior de 32, juntamente com a pontuação no momento.

2. Solução

2.1. O Algoritmo

O algoritmo fundamental para o funcionamento do programa é o de movimentação, localizado na proc movimento. A base de sua implementação se deu após diversos diálogos entre os autores deste trabalho. O modo optado para seu funcionamento é o seguinte: quando um momento é realizado, a matriz a ter suas peças movimentadas em uma direção,

tem seu offset passado através do registrador BP para a proc movimento, vindo da proc qualDirMover. Além disso, no registrador BH é passado quem está realizando a jogada. Os possíveis estados de BH são: 0 == player; 1 == simulador; 2 == simulador com matriz clone. em BL é passada a direção na qual está sendo movida. Isto é importante, pois a proc andaUmaPos depende do valor de BL para incrementar corretamente o valor no registrador passado em AX. O registrador DI possui o índice da casa onde os movimentos irão começar, sendo: 0, movimento para cima ou esquerda; 12, movimento para baixo; 3, movimento para a direita.

Logo ao início, os registradores de controle são zerados e a parte alta do registrador CX (CH) passa a ser 4, indicando a quantidade de movimentos a serem feitos. Já a sua parte baixa (CL) fará o controle do avanço de linhas e/ou colunas. Como registrador temporário de controle de verificação, foi escolhido a parte alta do registrador AX. Os registradores SI e DI são utilizados, com DI sendo o índice da casa com uma peça e SI sendo o índice a ser utilizado para buscar uma peça. Ambos iniciam com o mesmo valor e este é também armazenado em um byte na memória, o qual auxiliará na mudança de linha ou coluna, ao fim da varredura das 4 peças. O valor igual, porém, no registrador SI, logo é incrementado na direção correta para efetuar a busca de uma casa que possua algum valor. Caso não sejam encontrados valores, ocorre um jump para a verificação se há ou não uma próxima linha ou coluna a ser varrida pelo algoritmo. Caso um valor seja encontrado, é feita uma comparação entre o valor em DS:[BP][DI] e DS:[BP][SI], para verificar se ambos são da mesma potência. Se não forem, DI é incrementado na direção correta até uma casa vazia ser encontrada e/ou DI e SI possuírem o mesmo valor. Caso uma casa vazia seja encontrada através do índice em DI, o valor em DS:[BP][SI] é passado para a casa em DS:[BP][DI], caracterizando o movimento lógico realizado pelo algoritmo quando uma peça é encontrada e as potências não podem ser somadas. Caso os valores em DS:[BP][DI] e DS:[BP][SI] possam ser somados, o valor em DS:[BP][DI] é incrementado (o que indica que, a exemplo, se na casa 3 está o valor 2 na matriz lógica, este valor passará a ser 3, assim, sua representação na tela, graficamente, será 8.). O valor em DS:[BP][SI] é zerado e as posições de memória que fazem o controle das peças que já foram movidas e das casas vazias são ambas incrementadas. Neste ponto, caso BH seja 2, o algoritmo efetua um jump para o ponto onde o valor em DS:[BP][SI] é zerado e o índice em DI é movimentado na direção correta através da proc andaUmaPos, pois a matriz clone possui um funcionamento especial. Caso BH seja 1, será verificado se o valor formado é maior do que quatro. Se for, a proc armazenaValsIA é chamada. Isso corresponde a funcionalidade disposta ao final da simulação, para exibir quais peças na potência de dois, acima de 32, foram alcançadas, com o menor número de movimentos e o escore no instante em que foram alcançadas. Se BH for 0, tal proc não será executada e o então o escore é somado ao valor formado na junção das duas peças. O valor da junção é passado a AX e comparado com 2048 para marcar uma variável de controle caso o seja. A variável a qual faz o controle lógico do escore da tela, é somada a AX e então a proc irá para as verificações finais. O valor em DS:BP[SI] é, então, zerado e ocorre um jump para o trecho em que DI é movimentado na direção correta, através da proc andaUmaPos. Se o valor em DS:[BP][DI] for zero, o valor em DS:[BP][SI] é passado para ele. Caso contrário, ocorre a verificação se chegou ao final a varredura daquela linha ou coluna. Se chegou ao fim, vai para a próxima linha ou coluna, incrementando corretamente com base no valor da variável valColAnterior. Após o incremento, é verificado se existem

mais linhas ou colunas a serem varridas, comparando a parte baixa do registrador CX com 4. Se não existirem, ocorrem os respectivos desempilhamentos relacionados aos empilhamentos iniciais e a proc de movimentação retorna para quem chamou-a. Caso existam movimentos, a execução descrita nesta subseção ocorre novamente.

2.2. O Simulador

O simulador foi implementado de modo que ele possui 4 fatores que influenciam na decisão de movimento a escolher. Dentre os quatro movimentos possíveis para a escolha: para cima, para baixo, para a direita, para a esquerda, são somados valores dispostos em uma matriz auxiliar (denominada MatrizClone), a qual serve para distribuir pesos para as posições da matriz principal, o que faz com que os maiores valores fiquem em um lugar especifico, definido por esta matriz de pesos. Outro fator que o simulador leva em conta e a quantidade de pecas livres apos um movimento teste, valor este que e contabilizado junto com o peso mais o valor das pecas dispostas no tabuleiro. Ao retornar, caso o numero de posições vazias no tabuleiro forem menores que dez, o valor de adaptação do movimento recebe uma punição diminuir suas chances de ser escolhido. Não obstante, além de tais fatores, um terceiro fora implementado, com vistas a simular uma estrategia mais próxima da qual e utilizada por um jogador humano: a possibilidade do simulador testar as quatro direções para as quais ele pode movimentar-se e, apos isso, testar mais quatro direções com base em cada uma delas, sendo a quantidade de testes definida em código. Assim, ele pode fazer uma projeção de qual a melhor direção a se escolher. Este método, porém, resultou em um desempenho não satisfatório, com o simulador alcançando apenas o valor máximo 512 em um teste que iterou sobre 5 mil simulações.

A tabela 1 mostra o resultado de uma simulação de 100 jogos. A menor quantidade de jogadas que foi necessária para atingir determinados valores de peças é mostrada na coluna do meio, e a quantidade de pontos no momento em que a peça foi obtida é mostrada na terceira coluna.

Tabela 1. Resultado de simulação de 100 jogos, seed=1000d

Valor da peça	Quantidade de jogadas	Pontos
32	12	68
64	28	256
128	48	572
256	98	1428
512	180	3276
1024	não atingido	não atingido
2048	não atingido	não atingido

2.3. Descrições das PROCs

2.3.1. modoVGA

Esta proc efetua a troca de modo através da interrupção 10H para o modo 320 de largura por 240 de altura, com uma página e 256 cores

2.3.2. modoText

Esta proc efetua a troca para o padrão modo utilizado pelo DOS. Este, é o modo texto, o qual divide a tela em 40 colunas por 25 linhas.

2.3.3. escChar

Escreve o caractere em AX na posição atual do cursor. Em BX, terá a cor do caractere.

2.3.4. posicionaCursor

Posiciona o cursor em uma linha (DH) e coluna (DL) na tela.

2.3.5. converteNumStr

Esta proc possui duas funcionalidades: se SI=0, converterá o número em AX para string e colocará no endereço em DI. Se SI=1, o número será convertido de int para caractere e exibido na tela, na posição DH(linha) x DL(coluna).

2.3.6. statusTela

Responsável por exibir o status do jogo em andamento, que inclui o número atual de pontos, a quantidade de jogadas e a maior pontuação obtida no jogo. Caso o jogador ultrapasse a maior pontuação obtida, a mesma será substituída pela pontuação atual do jogador.

2.3.7. limpaMsgsStatus

Limpa o conteúdo de uma string (transforma todos seus caracteres em 0). Utilizada para limpar as strings de números que que são usadas na tela de status do jogo.

2.3.8. aiSimulacoesFim

Exibe a tela de estatísticas quando uma simulação é terminada. Consiste em uma tabela em que cada linha contém uma potência de 2, e a menor quantidade de jogadas que o simulador precisou para obter uma peça com esse valor, e a respectiva quantidade de pontos no momento que o simulador obteve essa peça.

2.3.9. statusPrintNums

Proc que imprime números da parte de status do jogo. Recebe em BP o endereço do número.

2.3.10. printTelaNumeros

Monta o tabuleiro na tela. Desenha as 16 peças, que consistem em um número e uma borda ao redor do números. Cada valor diferente de peça tem uma cor correspondente.

2.3.11. limpaFimTela

Limpa a parte inferior da tela, chamando a proc executaScroll, que se encontra abaixo do tabuleiro do jogo, que fica com pixels coloridos em decorrência dos deslocamentos das peças acima.

2.3.12. executaScroll

Esta proc recebe em BH o atributo a ser colocado na tela, apos o scroll, em cx linha/coluna inicial e dx linha/coluna final. Em al recebera o numero de linhas movidas no scroll.

2.3.13. telaPreta

Apaga tudo o que está na tela, chamando a proc executaScroll, deixando-a completamente preta.

2.3.14. printSqrTela

Coloca dois quadrados na tela, um de cor em BX e um preto menor, para formar cada uma das 16 peças do tabuleiro.

2.3.15. printTela

Imprime uma string na tela, apontada por BP e de tamanho em CX. DX conterá a posição na tela e em BX a cor.

2.3.16. printSetaIaTela

Coloca as setas de direção quando um movimento é feito no jogo ou no simulador. A seta aponta para a mesma direção na qual o movimento foi realizado.

2.3.17. setAtributoNum

Define a cor de uma peça baseado no valor da mesma, retornando-a em BX. Se DX=0, deve conter em DI o índice da peça na matriz principal do jogo. Se DX!=0, então deve receber em SI a potência do valor a se obter a cor.

2.3.18. setColuna

Proc que define em que linha e coluna será escrito o número de cada peça do jogo. Em DH estará a linha e em DL estará a coluna.

2.3.19. verificaMultiplos

Verifica se existem peças adjacentes de valor igual. Uma peça é adjacente de outra se estiver imediatamente acima, abaixo, à esquerda ou à direita dela. Retorna 1 em AX se sim, e 0 se não. Usada para verificar se ainda há movimentos válidos para se realizar, pois se duas peças adjacentes têm o mesmo valor, elas podem ser mescladas. Essa proc somente é chamada quando não há mais peças vazias no tabuleiro, isto é, quando todas peças têm um valor; por isso é necessário verificar apenas as adjacentes.

2.3.20. lerQuantidadeJogadas

Solicita ao jogador a quantidade de simulações para a inteligência artificial realizar e lê a entrada do jogador, retornando-a em CX. Lê somente números e a tecla enter. Números que precisem de mais de 16 bits são lidos, mas geram um retorno indeterminado.

2.3.21. ai2048

Chama outra proc para determinar a quantidade de simulações para executar, e então realiza o loop principal do simulador de jogos, o qual itera duas vezes sobre uma matriz clone de cada movimento testado na matriz principal, para determinar qual matriz trara o melhor valor de adaptação e sera sera o movimento escolhido. Ao final, chama a proc para exibir os resultados.

2.3.22. aiFitness

Esta proc efetua o produto escalar da matriz em passada em BP, pela matriz de pesos. O valor eh reunido em AX, que, por sua vez, sera multiplicado pelo numero de casas vazias (se estiverem vazias + de 10) caso contrario, prossegue e retorna em AX o o melhor valor de adaptação levando em conta os critérios acima. Ela recebe em recebe em BP a a ser testada.

2.3.23. maiorPecaCanto

Proc que busca na matriz de pesos a posição com o maior peso, ou seja, em qual canto esta a peca com maior valor. Em seguida, busca na matriz oficial do jogo a peca com maior valor e verifica se as duas posições são iguais. Se não forem, retorna 0. Se forem, retorna o valor da posição.

2.3.24. ai2048mover

Esta proc recebe em BP o offset da matriz que sera movida para os testes do simulador. Esta matriz sera a matriz clone, ou seja, de um movimento inicial realizado na proc principal do simulador, esta proc testara todos os movimentos possíveis e retornara no registrador DX o maior valor de adaptação de uma direção executada sobre a matriz clone. Este resultado sera levado em conta na escolha de qual direção ira ser escolhida para ser executada sobre a matriz oficial pelo simulador.

2.3.25. player2048

Proc executada durante o jogo. Irá ler o movimento inserido pelo jogador e executá-lo, caso seja válido. Chamará os procedimentos para desenhar o tabuleiro e o status do jogo. Irá verificar se o jogo terminou e chamar a proc de tratamento correspondente.

2.3.26. fimDeJogo

Exibe a tela de game-over e juntamente estatísticas do jogo (pontos e número de jogadas). Verifica se o jogador obteve um novo recorde e, se sim, chama as procs que solicitam o nome e colocam o recorde na tabela armazenada da memória. Depois, exibe a tela de recordes. Dessa tela, é possível retornar à tela inicial do jogo.

2.3.27. leNomeRecorde

Chamada quando um novo recorde é obtido, solicita para o jogador inserir o nome e lê. Recebe em AX a posição de um novo recorde, para colocar o nome do jogador, o número de jogadas e o número de pontos nas tabelas necessárias.

2.3.28. deslocaRecordes

Recebe a posição de um novo recorde em CX, e desloca (tanto da tabela de recordes quanto do vetor de recordes) todos os recordes que estão abaixo do valor em CX, para dar espaço para o novo recorde.

2.3.29. qualDirMover

Recebe em AL o código da tecla de movimento e ajusta posições de memória internas (”variáveis”) e registradores para tratar do movimento. Mostra a tecla que foi pressionada na tela e chama a proc para realizar efetivamente o movimento.

2.3.30. delay

Realiza um delay de 300 milissegundos através da interrupção 15h.

2.3.31. movimento

Proc responsável pelo movimento das peças, o movimento é recebido em BL. Desloca as peças de acordo com o movimento recebido, e mescla quando encontrar duas peças adjacentes de mesmo valor e soma o valor na pontuação do jogo.

2.3.32. armazenaValsIA

Proc que busca por todas as peças do tabuleiro e verifica se alguma tem valor igual ou superior a 2^5. Caso houver, verifica se deve atualizar as estatísticas da simulação, que armazenam a menor quantidade de jogadas para atingir determinados valores de peças.

2.3.33. andaUmaPos

Recebe em AX o índice (posição entre 0 e 15) de uma casa e retorna a próxima casa, baseado na direção de um movimento (cima, baixo, esquerda, direita). A direção é recebida em BX. Por exemplo, se o movimento for para cima, irá retornar a posição da casa imediatamente acima da que foi recebida.

2.3.34. numeroAleatorio

Gera um número aleatório e coloca em AX.

2.3.35. colocarPeca

Coloca uma peça, de valor 2 ou 4, em uma posição vazia aleatória do tabuleiro.

2.3.36. resetarJogo

Responsável por deixar o jogo em seu estado inicial para jogar. Isso inclui zerar todas as casas e colocar a primeira peça do jogo, zerar o score e a quantidade de jogadas, e outras variáveis de controle.

2.3.37. printSprite

Esta proc possui a funcionalidade de colocar, pixel a pixel na tela, um valor passado em DI. No caso desta implementação, ela possui algumas restrições que não a deixam genérica, pois é utilizada apenas para colocar o título ”2048”no menu principal. Ela coloca, em posições passadas a DH e DL, um byte na memória de vídeo (no modo VGA, a memória de vídeo começa em 0A000H), identificada em ES. A proc se encarrega de copiar, linha a linha, todo o vetor de bytes passado a ela. Sua utilização se dá pois fora um pré-requisito do trabalho a exibição do título em no mínimo duas linhas. Em virtude disso, os autores deste trabalho não acharam necessário torná-la genérica, devido a sua utilização única.

2.3.38. lerTecla

Lê uma tecla pressionada pelo jogador e retorna em AL.

2.3.39. mostraRecordes

Exibe a tela com a tabela de recordes.

2.3.40. menu

Exibe a tela inicial do jogo, com o menu. Aguarda o jogador pressionar uma tecla e, se for uma tecla de uma ação válida (jogar, recordes, etc.), chama a proc responsável.

2.4. Descrição das principais variáveis

2.4.1. *MatrizJogo*

Matriz principal do jogo. É um vetor com 16 elementos. Cada elemento é um byte. Isso significa que os valores não são armazenados por sua quantidade, mas sim por sua potência de 2. Por exemplo, ao invés de armazenar 512 (9 bits, uma word é necessária), se armazena apenas o 9. Este armazenamento é feito desta maneira para facilitar a exibição dos valores das peças na tela. O valor dentro das posições de memória da MatrizJogo servirá como deslocamento para a exibição dos números em formato string na tela, presentes no vetor numsTela. A exemplo: a string ”1024”fica na posição 10*4 do vetor de strings. Caso o número 10 esteja na MatrizJogo, este é utilizado para ser multiplicado por 4 e então a exibição do valor correto é feita na tela. Este controle lógico ocorre pois a manipulação dos números diretamente na parte gráfica tornaria o projeto demasiado complexo, o que fugiria do escopo deste trabalho.

2.4.2. *numsTela*, *atribsTela* e *numsTelaScore*

numsTela é um vetor com 12 strings de 4 caracteres cada, cada elemento é a representação em string de uma possível peça do jogo; ou seja: contém a representação em string do valor de cada uma das 12 possíveis peças. *atribsTela* é um vetor que contém a cor de cada uma das 12 possíveis peças. *numsTelaScore* é o vetor que contém o valor de cada uma das 12 possíveis peças.

2.4.3. *MatrizClone*, *MatrizPesos*, *melhorScoreClone*, *melhorMovIA* e *melhorScoreIA*

Respectivamente, uma cópia da matriz lógica principal do jogo, a qual será utilizada na proc *ai2048* para a verificação em *profundidade* dos movimentos. Por ser uma cópia idêntica da Matriz principal, possui o mesmo tamanho que ela. A *MatrizPesos* é um vetor de 16 posições as quais possuem um valor atrelado a cada uma. A ordem dos valores se dá de que o maior deles fique em algum canto da MatrizPesos e as posições adjacentes, possuem menores valores, até chegarem a zero, no outro extremo. Esta matriz servirá

para calcular o produto escalar entre a matriz passada à proc *aiFitness*, para definir o grau de aptidão do movimento feito. Por fim, respectivamente, as três variáveis restantes nesta subseção mantém o registro de qual foi o melhor escore entre os movimentos efetuados por um clone, qual o movimento que possui o melhor grau de aptidão, ou seja, o maior escore dentre o calculado para os outros movimentos e seu escore, para comparação com o melhor escore da iteração sobre a matriz clone, para definir se será ou não o movimento na direção escolhida antes dos testes da matriz clone atual o escolhido.

2.4.4. *JogadasJogador* e *scoreTela*

Contém, respectivamente, a quantidade de jogadas e a quantidade de pontos que o jogador possui no jogo em andamento.

2.4.5. *RecordesTextos*, *RecordesPontos* e *RecordesContador*

A tabela de recordes encontra-se em *RecordesTextos*. Cada linha da tabela contém um número, o nome inserido pelo jogador, a quantidade de jogadas e o número de pontos. *RecordesPontos* é um vetor com os recordes em número, utilizado para determinar quando um novo recorde é atingido e a posição de novos recordes. *RecordesContador* contém a quantidade de recordes que já estão presentes na tabela. É necessário pois, de outra forma, não haveria como identificar se a tabela de recordes está vazia, ou quando ela termina.

3. Conclusão

O desenvolvimento deste trabalho transcorreu de modo contínuo, em fases separadas. Primeiramente os autores planejaram o que deveria ser implementado, estudando o melhor modo a fazê-lo, levando em consideração as funcionalidades básicas do projeto. A subdivisão do projeto em módulos facilitou o acoplamento final do mesmo, garantindo uma implementação sem demasiados empecilhos. Nota-se que, em virtude desta abordagem, um fator influenciou uma característica específica em um momento já avançado do projeto: o deslize das peças. Pelo funcionamento único da proc de movimento, o qual fora explicada na seção 2.3.24, a constante atualização da tela ao movimentar de casa em casa não se fez possível sem que a proc fundamental para o jogo fosse re-implementada desde seu início. Em virtude disso, os autores deste trabalho preferiram complementar as outras áreas do projeto, deixando esta para futuras implementações. A fase final de desenvolvimento envolveu a refatoração de alguns trechos de código, comentários melhorados e a elaboração deste relatório. O desenvolvimento deste trabalho mostrou-se de grande aprendizado para a prática da programação na linguagem Assembly x8086 vista em aula e, além disso, da implementação de boas práticas de programação sempre que possível. Isto, em conjunto com a fluidez do ciclo de desenvolvimento e o desafio da implementação do jogo 2048 permitiram aos autores deste trabalho terem uma experiência única, sem demasiadas complicações.