Tetris, RISC-V e Java: uma combinação inesperada

Leonardo Alves Riether Felipe de Araújo Flores

University of Brasília, Dept. of Computer Science, Brazil

Abstract

Nesse artigo serão descritos detalhes da implementação de um jogo estilo Tetris programado em assembly RISC-V e interpretado pelo programa RARS[TheThirdOne], com Bitmap Display, Keyboard MMIO e Interface MIDI. São apresentadas metodologias utilizadas para a construção do jogo e diferenças de uma implementação feita em alto nível para uma feita em assembly RISC-V.

Palavras Chave— Tetris, RISC-V, RISC, Assembly, Desenvolvimento de Jogos, Aprendizado, Baixo Nível

1 Introdução

Tetris[Wikipedia b] é um jogo de quebra-cabeça originalmente criado pelo programador e desenvolvedor de jogos soviético Alexey Leonidovich Pajitnov, em 1984. Esse jogo foi feito, inicialmente, para testar a capacidade de hardwares da época. Como o hardware evoluiu amplamente desde aquela época, hoje podemos rodar Tetris com uma JVM que interpreta assembly RISC-V, traduz para bytecode Java™, que é traduzido para instruções x86 e por fim é rodado no processador.

RISC-V[Foundation] é uma arquitetura do conjunto de instruções não-proprietária que vem sendo apoiada por cada vez mais empresas e indivíduos. Para investigar como podem ser construídos programas de computador com essa ISA, os autores construíram um jogo estilo Tetris em assembly RISC-V.

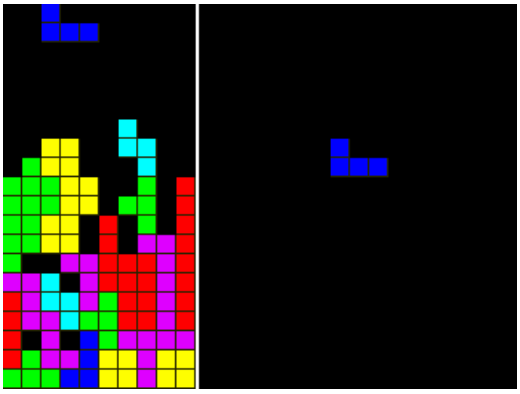

Figure 1: *Demonstração do jogo no RARS, com destruição de linhas desativada*

Este artigo tem como objetivo documentar como foi feita a implementação do Tetris em assembly RISC-V, descrevendo a metodologia utilizada, resultados obtidos e conclusões finais.

2 Metodologia

Por ser uma opção mais conveniente e barata do que adquirir um processador RISC-V, esse jogo foi programado no RARS: RISC-V Assembler and Runtime Simulator, uma ferramenta que permite o desenvolvimento de programas em assembly RISC-V dentro de um sistema operacional convencional, como Windows, Linux, ou qualquer sistema que rode Java. No RARS, é possível simular um teclado mapeado em memória (Keyboard MMIO) e uma tela para mostrar o jogo, também mapeado em memória (Bitmap Display). Uma vez que o RARS disponibiliza essas funções de certo alto nível, o desenvolvimento do jogo é facilitado fortemente. Mesmo assim, ainda é necessário o uso de estruturas básicas de programação, como laços de repetição e condicionais, o que torna a criação do jogo uma ótima oportunidade de aprendizado sobre linguagens de baixo nível, com um resultado visual e recreativo.

2.1 Representação das Peças

Há várias maneiras de se representar uma peça de Tetris, e cada uma tem suas vantagens e desvantagens. Um primeiro modelo considerado foi a representação por meio de uma matriz de bytes 4x4, onde se o byte da i-ésima linha e j-ésima coluna é diferente de zero, há um bloco naquela posição. Esse modelo é intuitivo e fácil de programar em linguagens de alto nível. Em assembly, porém, a manipulação dos índices da matriz torna esse modelo mais complexo de ser implementado, necessitando de dois laços de repetição aninhados e muitos registradores. Por esse motivo, o modelo descrito foi abandonado.

A segunda forma de representação pensada também consiste no armazenamento de uma matriz 4x4. Dessa vez, no entanto, cada bloco foi representado por apenas um bit: ligado se o bloco da posição existe, desligado caso contrário. Assim, é possível representar essa matriz com uma bitmask[Wikipedia a] de 16 bits, que foi utilizada como uma estrutura de pilha, em que o topo da pilha é o bit menos significativo. Essa estratégia facilita substancialmente as rotinas que utilizam as peças, visto que só são demandados dois registradores: um para contar quantos blocos foram retirados – isso determina a posição do bloco na matriz – e um para armazenar a bitmask. Além disso, as operações da pilha são extremamente simples de implementar: verificar qual é o topo da pilha é uma instrução `and` com a bitmask e o número 1 e retirar o topo da pilha é um bitshift para a direita. As operações para achar a posição do bloco na matriz a partir de um índice de quantos blocos foram retirados da pilha também são simples: sendo i esse índice, a linha é dada por $\lfloor \frac{i}{4} \rfloor$, o que pode ser feito pela instrução `srli $rd $i 2`, e a coluna é dada por $i \mod 4$, onde $\mod$ é a operação de resto da divisão, o que pode ser feito com `andi $rd $i 3`.

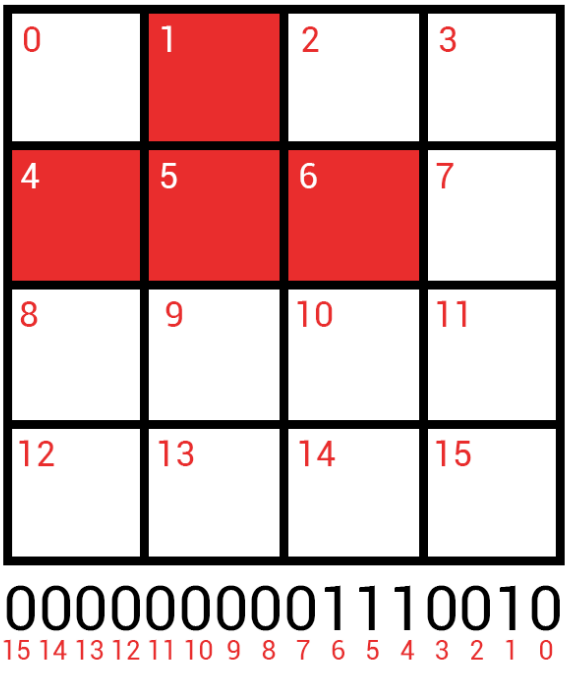

Figure 2: *Representação de uma peça por meio de uma pilha em bitmask*

Tendo esse modelo, a rotação das peças pode ser feita de maneira trivial.

2.2 Rotações

Poderia ter sido feita a rotação manual de cada bitmask: calculando-se as posições de um bit antes e depois da rotação e aplicando essa transformação cada vez que o jogador quer rotacionar a peça. Essa abordagem, no entanto, não é necessária, basta criar manualmente uma bitmask para cada rotação e para cada peça. Após essa etapa, foi criado uma word no segmento `.data` que identifica o índice de rotação das peças, que é utilizado para encontrar a bitmask certa. O resultado é um código (figura 3) sem legibilidade alguma, porém muito fácil de implementar e livre de bugs.

```
# Rotações manuais de 90d graus no sentido horário
piece:  .half 0x71, 0x113, 0x47,0x644
        .half 0xf, 0x1111, 0xf, 0x1111
        .half 0x74, 0x311, 0x17, 0x446
        .half 0x33, 0x33, 0x33, 0x33
        .half 0x36, 0x462, 0x36, 0x462
        .half 0x27, 0x232, 0x72, 0x131
        .half 0x63, 0x264, 0x63, 0x264
        .half 0xffff, 0xffff, 0xffff, 0xffff # bloco 4x4
```

Figure 3: *Bitmasks utilizadas para modelar as peças e suas rotações*

2.3 Matriz de Peças

Para fins de verificação de colisão entre peças, foi necessário criar uma matriz de bytes. Se uma posição da matriz é igual a 0, não há uma peça nessa posição, caso contrário, há uma peça. A matriz é declarada no segmento `.data` como um `.space 200`, para armazenar todas as 200 posições da matriz 20x10 do jogo. Dado isso, é possível acessar uma posição na i-ésima linha e j-ésima coluna (ambos começando no índice zero) com a fórmula $matriz[i \cdot 10 + j]$. Traduzir essa expressão para assembly RISC-V não é muito difícil, mas requer atenção e várias linhas de código, se comparado com C, sendo requisitados 4 registradores, caso o programador não queira alterar os valores de i e j durante o processo.

```
# a1 = i, a2 = j
li  t0 10
mul t0 t0 a1
add t0 t0 a2
la  t1 matriz
add t1 t1 t0
lb  t0 0(t1)
# t0 = byte em matriz[a1*10 + a2]
```

2.3.1 Geração de Peças

A geração de peças dentro do ambiente RARS é fácil, visto que há um `ecall` para gerar um número aleatório entre 0 e `a1`. Assim, podemos gerar um número para representar qual peça será utilizada e salvar em um registrador salvo (de `s0` a `s11`).

```
li a7 42 # syscall de geração de números
         # aleatórios no RARS
li a0 0
li a1 6
ecall
mv s8 a0
# s8 agora é o índice da nova peça
```

2.4 Desenho das Peças na Tela

O primeiro passo para desenhar uma peça na tela foi criar uma rotina de desenhar um bloco em uma determinada posição da matriz do jogo. Nessa rotina, são criados dois laços de repetição aninhados: o de fora para escolher em qual linha deve ser desenhado o pixel e o de dentro para escolher a coluna. Esses dois loops são mais complexos de serem implementados do que em C, pelo fato do loop de fora demandar duas labels, como mostrado abaixo.

```
    # t0 = i (loop de fora)
    # t1 = j (loop de dentro)
    mv t0 x0
    li t2 20 # max i
    li t3 10 # max j
loop_outer:
    bge t0 t2 loop_done # sai do loop de fora
    mv t1 x0
loop_inner:
    bge t1 t3 loop_outer_aft
    # ...
    addi t1 t1 1 # incrementa o j
    j loop_inner

loop_outer_aft: # segunda parte do
                # loop de fora!
    addi t0 t0 1
    j loop_outer

loop_done:      # ...
```

O que seria equivalente às seguintes 3 linhas de código em C:

```
for (int i = 0; i < 20; i++)
    for (int j = 0; j < 10; j++)
        // ...
```

Tendo a rotina de desenho de blocos, precisamos criar uma rotina de desenho das peças que carrega a bitmask de peça atual, fazendo uma cópia para um registrador, e desenha um bloco em cada posição que possui um bit ligado. Por causa do modelo de peça utilizado, apenas um laço de repetição é utilizado nessa etapa, o que facilita a escrita do código.

2.5 Loop Principal do Jogo

Sendo feitas as decisões de modelo de armazenamento das peças, maneira de fazer as rotações, acesso à matriz das peças, geração de peças e desenho de peças na tela, já é possível discutir como funciona o loop principal do jogo.

O primeiro passo executado é a leitura de entradas do usuário, feito com ecalls disponibilizados pelo RARS. A leitura das entradas é a primeira coisa a ser feita, porque, caso o jogo esteja pausado, o programa precisa detectar o momento em que deve sair da pausa, mas não deve atualizar mais nada. Com a leitura do teclado, são tomadas as respectivas ações para cada tecla apertada, como mover a peça, rotacionar ela, mover para baixo ou trocar estado de pausa do jogo.

Após isso, caso o jogo não esteja pausado, a peça é desenhada no lugar adequado e o processo é parado por um certo número de milissegundos, que muda conforme o jogo avança. Passado algum tempo, uma peça do mesmo modelo, porém de cor preta, é desenhada por cima da peça atual, efetivamente apagando-a.

Feito isso, são verificadas colisões com outras peças da matriz. Uma colisão nesse momento significa que a peça não consegue mais descer, portanto devemos colocá-la na matriz e gerar uma nova peça.

Por fim, ocorre a verificação de linhas completamente cheias e, se uma linha está cheia, ela é removida da matriz e da tela.

2.6 Interface MIDI

A interface MIDI disponibilizada pelo RARS, que propicia a oportunidade de tocar música e efeitos sonoros durante o jogo, é simples de entender o funcionamento, necessitando apenas da troca dos argumentos dos ecalls para mudar as notas e instrumentos. Com isso, foi possível criar uma música de menu e efeitos sonoros para quando as peças são empilhadas, deixando o jogo mais parecido

com outros Tetris atuais. Apesar disso, a interface MIDI do RARS está longe de ser perfeita, e começa a pular notas e desrespeitar delays após um tempo.

3 Resultados Obtidos

3.1 Programação

É certo que a facilidade de programar em linguagens de alto nível é muito maior que em linguagens de baixo nível. Esse fato torna a criação de um programa em assembly um processo mais complexo, que demanda mais tempo e atenção. Apesar de algumas estruturas serem bem similares a linguagens de alto nível; como loops *while* em C e a instrução `blt`, por exemplo; há distinções marcantes entre esses dois paradigmas. A falta de variáveis é uma das mais notáveis: ou o programador utiliza apenas registradores, que são escassos e podem ser mudados por outras funções facilmente, ou ele coloca explicitamente valores na pilha, que precisam ser carregados para dentro de registradores para serem manipulados.

Linguagens de alto nível, porém, são todas traduzidas, em última instância, para assembly. Desse modo, sempre é possível traduzir uma rotina em C, por exemplo, para uma em assembly RISC-V. Ao fazer esse processo de tradução, o programador adquire um conhecimento maior sobre como seu software roda em um computador, o que implica um maior entendimento tanto de linguagens de baixo nível quanto de alto nível.

Mesmo com os efeitos colaterais[Johnson] inerentes ao assembly, não foram detectados efeitos indesejados na execução do código. Um fato que ajudou nesse resultado foi o planejamento anterior à escrita do código e a divisão das rotinas em ”funções”, que evitam alterar registradores ”salvos”[WikiChip] e guardam valores que podem ser mudados por outras rotinas na pilha. Além disso, foram utilizados extensivamente comentários, principalmente para descrever o que cada registrador representa.

3.2 O Jogo

O resultado final do código é um Tetris bonito, funcional, divertido e *lagado*. Mesmo com algumas limitações, o jogo apresentou os resultados desejados: o jogador pode mover peças, rotacioná-las, há aumento na velocidade do jogo gradualmente, há colisão entre peças e entre peças e a borda da matriz, um sistema de pontuação e efeitos sonoros.

Para versões futuras do jogo, poderiam ser feitas várias alterações. São algumas delas:

- Em vez da matriz de peças guardar apenas se existe ou não uma peça na posição, ela poderia guardar a cor da peça ou zero se não existe peça ali. Isso facilitaria a implementação das peças caírem após uma linha ser destruída.
- Modificar a funcão de desenho dos blocos para criar blocos mais complexos, visto que os atuais são só uma cor sólida e uma borda preta de 1 pixel.
- Testar outras ferramentas além do RARS para rodar o jogo, possivelmente em um processador RISC-V ou um FPGA.
- Implementar um interpretador de assembly RISC-V em C+SDL, para conseguir rodar o jogo em várias plataformas, sem precisar do Java instalado.

3.3 Limitações

Foram obtidos alguns resultados indesejados ao desenvolver o jogo, alguns por causa de más decisões de design, outros por limitações do RARS.

Uma limitação nítida do jogo é a incapacidade de mover as peças para baixo quando uma linha é destruída, fazendo com que algumas peças fiquem ”flutuando” na tela, como é mostrado na figura 4. Essa limitação poderia ser corrigida, mas seria necessário mudar o modelo utilizado para armazenar a matriz de peças, além de criar uma rotina que redesenha a matriz inteira na tela, o que demandaria algum tempo. Fora isso, o RARS desenha os pixels na tela de forma muito lenta, e resedenhar a matriz poderia causar alguns bugs visuais.

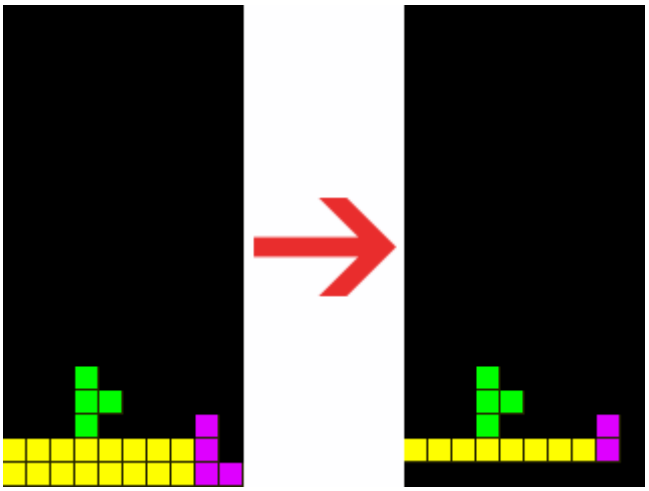

Figure 4: *Bug causado por uma má decisão de design do código*

Outra restrição encontrada é a forma de leitura das entradas do usuário, por meio do Keyboard MMIO. Como a leitura só é feita uma vez por frame, e cada frame só é desenhado de acordo com a velocidade do jogo – começando em 500ms –, as entradas do usuário possuem um delay perceptível, o que pode ser indesejado por alguns jogadores. Isso poderia ter sido corrigido utilizando um delay menor de leitura do teclado, porém só atualizando a tela e as peças em alguns dos frames.

Além desses problemas, um mais difícil de ser consertado foi encontrado: baixa velocidade de desenho no Bitmap Display do RARS. Por causa dessa lentidão, tornou-se inviável desenhar a tela inteira a cada frame, técnica que facilitaria o reparo do bug apresentado na figura 4. A solução encontrada para remendar essa limitação foi , em cada frame, desenhar apenas a peça que está se movendo. Toda vez que a peça se move, uma versão em preto dela é desenhada por cima da posição atual e depois a versão normal é desenhada na posição seguinte. Mesmo assim, quando a velocidade do jogo aumenta, ainda é possível ver alguns bugs visuais, como a passagem de frame antes da peça ter sido desenhada completamente na tela.

Por fim, o RARS provou não ser uma ferramenta muito confiável, tendo travado incríveis 27 vezes durante o desenvolvimento do projeto.

4 Conclusão

Este artigo apresentou o processo de criação de um Tetris em assembly RISC-V, com auxílio da ferramenta RARS, que disponibiliza um Bitmap Display e Keyboard MMIO para facilitar a interface do assembly com o usuário. Foram apresentados os modelos de armazenamento das informações utilizadas no jogo e técnicas empregadas para tornar a escrita do código mais simples.

É de se esperar que um programa escrito diretamente em assembly seja extremamente rápido. Esse não é o caso, mas o projeto ainda serviu de uma ótima experiência para aprender a pensar em baixo nível, o que ajuda a entender códigos de alto nível, principalmente na área de otimização de um programa para rodar mais rapidamente e utilizar o hardware de forma mais eficiente.

Finalmente, o projeto foi muito divertido de programar, apesar das travadas do RARS e assembly ser difícil de debugar. Sempre é muito satisfatório ver o código funcionando no final e dando o resultado esperado.

References

FOUNDATION, R.-V. Risc-v: The free and open risc instruction set architecture.

JOHNSON, K. What is a ”side effect”?

THETHIRDONE. Rars: Risc-v assembler and runtime simulator.

WIKICHIP. Registers - risc-v.

WIKIPEDIA. Mask (computing).

WIKIPEDIA. Tetris.