Laboratório 01 - Grupo 09

Antônio Vinicius de Moura Rodrigues 19/0084502,
Gabriel Pinheiro da Conceição 19/0133724,
Leandro de Sousa Monteiro 17/0060454

[1] Universidade de Brasília - Instituto de Ciências Exatas
Dep. Ciência da Computação - CIC0099 - Organização e Arquitetura de Computadores
2020.2 - Turma A - Professor Marcus Vinicius Lamar
Prédio CIC/EST - Campus Universitário Darcy Ribeiro
Asa Norte 70919-970 Brasília, DF

Abstract. *This report aims to put into practice the use of tools that make it possible to program codes in machine language in a more human-friendly way.*

Keywords: *Machine language, Assembly, RISC-V,*

Resumo. *Esse relatório visa por em prática a utilização de ferramentas que possibilitam a programação de códigos em linguagem de máquina de forma mais amigável para humanos.*

Palavra-chave: *Linguagem de máquina, Assembly, RISC-V,*

1. Introdução

”Assembly ou linguagem de montagem é uma notação legível por humanos para o código de máquina que uma arquitetura de computador específica, utilizada para programar códigos entendidos por dispositivos computacionais, como microprocessadores e microcontroladores. O código de máquina torna-se legível pela substituição dos valores em bruto por símbolos chamados mnemônicos.”[Wikipédia 2020]

Adicionalmente, para o uso da linguagem Assembly é necessário eleger uma CPU a se trabalhar como corrobora phixies, quando diz que: “Cada tipo de CPU tem sua própria linguagem de máquina e linguagem de montagem, portanto, um programa de linguagem de montagem escrito para um tipo de CPU não será executado em outro.”

Diante disso, a ISA/CPU escolhida para implementação nesse relatório será o RISC-V. Segundo wiki: O ”RISC-V é livre para ser usado para qualquer finalidade, permitindo a qualquer pessoa ou empresa projetar e vender chips e software RISC-V sem precisar pagar royalties.

Embora não seja o primeiro conjunto de instruções livre, ele é importante porque foi projetado com foco para dispositivos computadorizados modernos, como computação em nuvem, aparelhos móveis, sistemas embarcados e internet das coisas.”

Por fim, todos os testes serão feitos utilizando o RARS (*RISC-V Assembler and Runtime Simulator*), software que torna possível simulação e montagem de códigos em Assembly RISC-V a partir de dispositivos que interpretam aplicativos *.jar*, ou seja, possuem a linguagem Java instalada.

2. Procedimentos

2.1. Simulador/Montador Rars

2.1.1. Compilação para Assembly

Para testar a compilação de códigos em linguagem C para Assembly RISC-V, foi utilizado o site Compiler Explorer. Os códigos gerados pelo compilador, a partir dos exemplos em C disponibilizados, foram feitos com duas diretivas de otimização: -O0 e -O3.

2.1.2. Adaptação para o Rars

O código do programa sortc.c foi compilado no site Compiler Explorer para Assembly RISC-V, com a diretiva de otimização -O0. Para que o código pudesse ser executado no Rars, foram feitas mudanças em (colocar as mudanças)

2.1.3. Diretivas de Otimização

O programa sort_mod.c foi compilado para Assembly RISC-V com as diretivas de otimização -O0, -O1, -O2, -O3 e -Os. Os códigos obtidos foram executados no Rars, para que os números de instruções e o tamanho (em bytes) de cada código fosse analisado.

2.2. Compilador cruzado GCC

2.2.1. Compilação de programas triviais

Foram compilados programas triviais escritos em C no Compiler Explorer utilizando-se as diretivas de otimização -o3 e -o0 de modo a entender como o Assembly aloca os registradores e utiliza a memória.

Foi notado que por padrão o Assembly utiliza os registradores do tipo a, para armazenar imediatos, ou seja, argumentos. Adicionalmente, nota-se que ao terminar a operação feita com os registradores de argumento o montador utiliza os registradores do tipo s para enviar os resultados das operações ao locais desejados.

2.2.2. Compilação do programa “sort.c” com diretiva -O0

Foi gerado o arquivo sort.s baseado no arquivo sort.c utilizando as diretivas de otimização -o0. Para que o código funcionasse no Rars foi necessário alterar diversas linhas do código de modo a corrigir os saltos condicionais e incondicionais.

Em linhas como “j 10214 <sort+0x36>” foi necessário procurar a origem da função “sort” e somar ao endereço encontrado a quantidade de instruções em hexadecimal fornecida. Nesse exemplo a linha que continha sort + 9, ou seja, 36/4 = número de instruções, foi substituída por um termo chave qualquer como “funcao1” e esse mesmo termo utilizado para substituir “10214 <sort+0x36>” na linha original

O código bruto, sem as devidas alterações para que funcione no RARS possui mais de 150 linhas. Já o código feito inteiramente em Assembly para a execução das mesmas funcionalidades conta com 82 linhas.

2.2.3. Compilação do programa "sortc_mod.c" com diretivas diversas

Tabela 1. Tabela comparativa com os diferentes valores das diretivas aplicadas no arquivo "sort_mod.c"

Diretiva	Instruções	Tamanho dos Códigos (bytes)
-O1	4.120	2.749 bytes
-O2	3.720	2.237 bytes
-O3	3.154	2.220 bytes
-Os	3.920	2.605 bytes

2.3. Solução de Labirintos

Os testes e demais explicações quanto à parte de implementação dos experimentos que envolvem os labirintos podem ser vistos em vídeo neste link.

2.3.1. Desenhe o labirinto no centro da tela

Para desenhar o labirinto no meio da tela, foi preciso pegar o eixo X e Y, tanto do labirinto quanto do Bitmap e dividi-los por 2, tendo esses resultados em mãos basta pegar o eixo X do Bitmap e subtrair o eixo X do labirinto, assim temos o endereço da posição que devemos começar a desenhar o labirinto no eixo X do Bitmap, o mesmo foi feito com o eixo Y, no final basta somar os dois valores ao endereço inicial do Bitmap que teremos o endereço de onde o labirinto deve começar para ser desenhado no centro da tela.

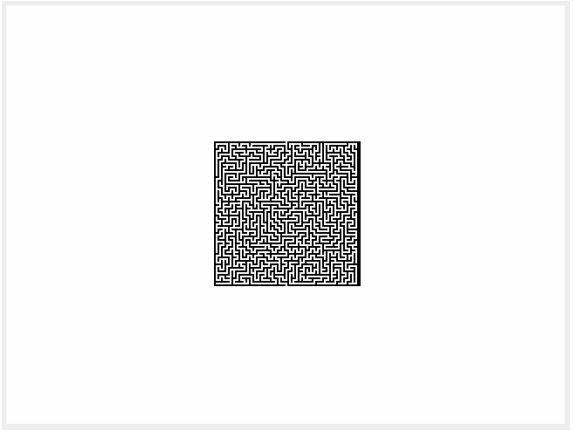

Figura 1. Labirinto desenhado no centro da tela.

2.3.2. Caminho-resolução do labirinto

Para implementar a resolução do caminho do labirinto, foi feito uma copia do labirinto original, onde nela foi possível alterar os valores dos pixels com o intuito de utiliza-los como flags, ou seja, o algoritmo busca nos 4 endereços que o circulam o valor 255, que na codificação de cor de 8 bits por pixel representa o Branco, caso ele encontre ele pula pra esse endereço e altera o endereço anterior para a cor Verde, que é representado pelo 60, o endereço que contém a cor verde nos informa que aquele endereço já foi visitado uma vez. Quando o algoritmo não encontra nenhuma cor branca nos endereços ao seu redor, ele busca as cores verdes, e pula pra elas alterando o valor do endereço que estava para a cor preta, que é representada pelo 0, com a cor preta o algorítimo sabe que não deve mais voltar naquela posição. Depois de executar esses passos em uma quantidade indeterminada de vezes, o algoritmo consegue encontrar a saída e determinar a resolução do labirinto pelas cores Verdes que não foram pintadas de Pretas.

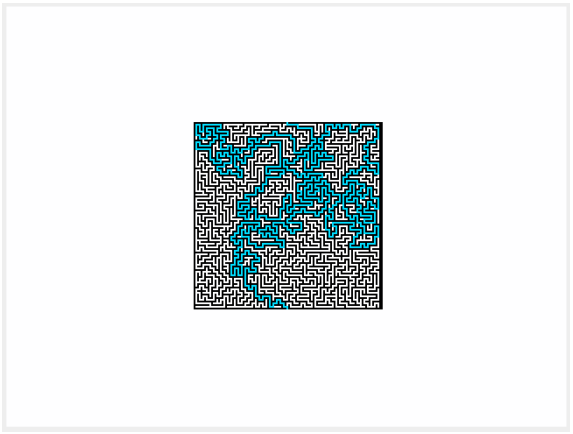

Figura 2. Labirinto com a resolução completa do caminho.

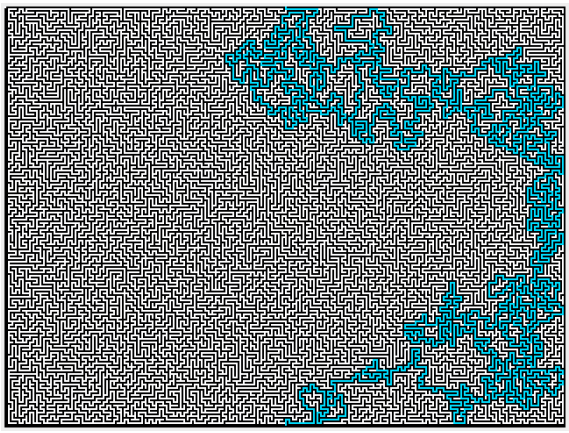

Figura 3. Maior Labirinto com a resolução completa do caminho.

Para animar o labirinto, basta executar os mesmos processos listados acima buscando as cores Verdes, dessa vez temos certeza que nos endereços ao redor sempre haverá o valor 60 que mostra o próximo endereço do caminho, com isso, basta alterar o valor

da memória do Bitmap e pular para esse endereço. Fazendo isso e utilizando o Time conseguimos animar e ver o processo da resolução do labirinto sendo executada.

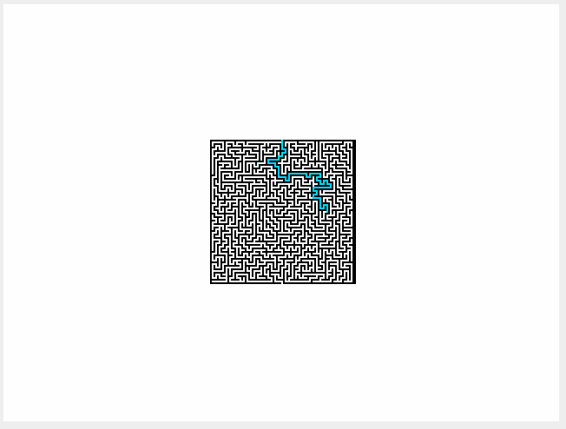

Figura 4. Labirinto com a resolução do caminho sendo executada.

2.3.3. Histograma do número de instruções: Solve_maze em 20 labirintos aleatórios de tamanho 51x51

Ao executar o código para 20 labirintos aleatórios de 25 linhas por 25 colunas, foram obtidos os dados da Tabela 2, onde os números de instruções por labirinto foram divididos em 6 intervalos. Um histograma com a frequência de ocorrência de cada intervalo está na Figura 5

Tabela 2. Dados do histograma do número de instruções para execução dos labririntos

Classe	Intervalo do Número de Instruções (N)	Frequência
1	99966<N<106477	3
2	106477<N<112988	3
3	112988<N<119500	3
4	119500<N<126011	3
5	126011<N<132522	2
6	132522<N<139034	6

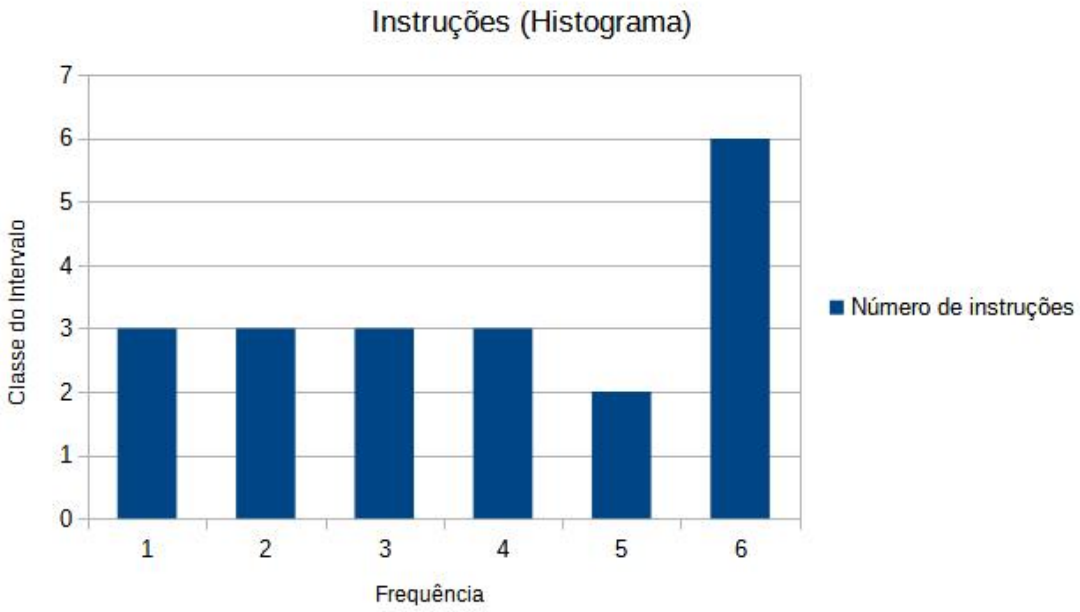

Figura 5. Histograma do número de instruções

Tabela 3. Desvio Padrão e Média do número de instruções

	Número de Instruções
Média	121782.7
Desvio Padrão	12666.9

2.3.4. Gráfico NxI (I=instruções) para 20 labirintos aleatórios de tamanho variado NxN

Tabela 4. Número de Instruções por Labirinto NxN

N	Instruções
5	7865
11	83514
17	89484
23	113346
29	174492
35	133764
41	169920
47	324808
53	262692
59	251950
65	487840
71	326622
77	563508
83	791032
89	934034
95	954906
101	772534
107	603102
113	1080652
119	1041706

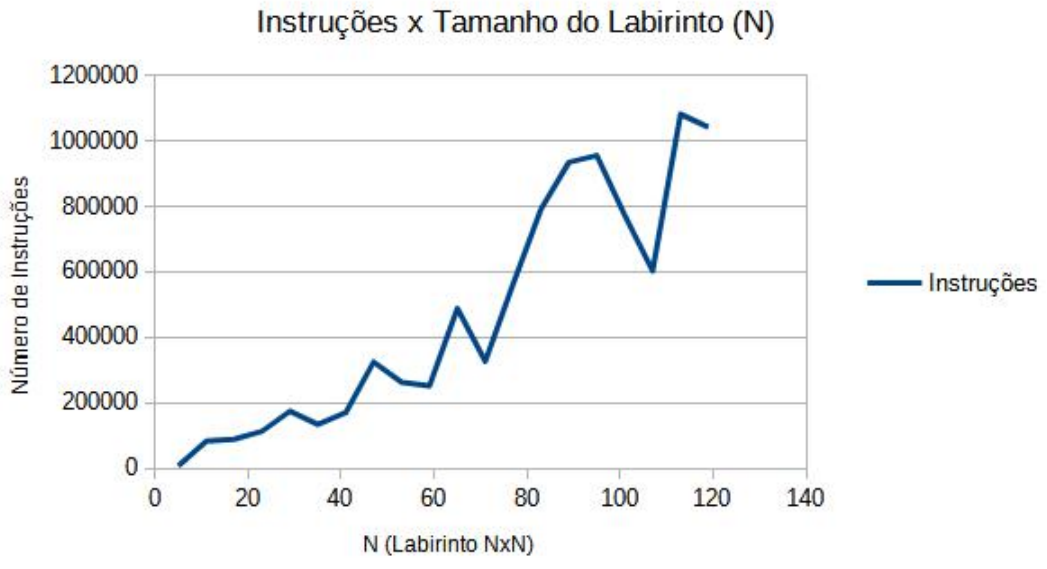

Figura 6. Gráfico N x I

No gráfico da Figura 6, é possível que perceber que há um tendência de que quanto maior for o labirinto, mais instruções serão executadas pelo programa para resolvê-lo.

2.4. Análise de Workflow (função *main*) para labirinto 319x239

2.4.1. Contagem de Instruções

Na execução do procedimento *main* foi possível calcular um total de 1.747.630 instruções executadas a fim de imprimir e animar instantaneamente a resolução do maior labirinto possível (319x239)

2.4.2. Tempo de Execução

Para executar todo o procedimento foram necessários 2775 ms ou $\approx$ 2,775 segundos

2.4.3. Frequência do processador RISC-V equivalente

Para o cálculo da frequência do processador utilizando os dados acima e CPI = 1, basta utilizar a fórmula:

$$t_{exec} = I \times CPI \times T \tag{1}$$

Fazendo as manipulações e substituições necessárias obtêm-se que a frequência em Hz é igual a:

$$f = (I \times CPI)/t_{exec} \tag{2}$$

$$f = (1747630 \times 1)/2,775 \tag{3}$$

$$f = 629.663,124 \tag{4}$$

2.4.4. Tamanho da Pilha

Para determinação do tamanho da pilha foi necessário analisar o código e monitorar o valor do registrador sp (*stack pointer*). No entanto, na implementação utilizada não foi necessário o uso da pilha para armazenamento dos vetores que definem o caminho e portanto o valor da pilha permaneceu inalterado durante toda a execução dos programas. A saber: 0x7fffeffc

3. Conclusão

Diante dos experimentos realizados nesse relatório, fica claro que a linguagem Assembly torna a programação em linguagem de máquina mais vantajosa.

Isso se dá por vário motivos, mas principalmente porque é possível notar experimentalmente, na seção 2.3.4 que quanto maiores as proporções do labirinto submetido

aos algoritmos de resolução, maior é a quantidade de instruções montadas pelo assembler utilizado.

Nas seções 2.2.3 e 2.3.3 nota-se ainda que a utilização de Assembly torna a execução de programas mais eficiente, visto que na maioria das linguagens de alto nível o código passa por um compilador e posteriormente ao montador para então ser convertido em código de linguagem de máquina. Em assembly, como foi possível notar, o código é diretamente montado pelo RARS, pulando a etapa de compilação. Pode-se obter mais informações sobre os processadores no seguinte site: [Garrett 2014]

Referências

[Garrett 2014] Garrett, F. (2014). Saiba o que é processador e qual sua função. `https://www.techtudo.com.br/artigos/noticia/2012/02/o-que-e-processador.html`. [Online; accessed 13-December-2020].

[Wikipédia 2020] Wikipédia (2020). Linguagem assembly. `https://pt.wikipedia.org/wiki/Linguagem_assembly`. [Online; accessed 24-March-2021].