Laboratório 01 de Organização e Arquitetura de Computadores

Maria Claudia Campos Martins
Universidade de Brasília
170109968@aluno.unb.br

Vitor de Oliveira Araújo Araruna
Universidade de Brasília
202060980@aluno.unb.br

1. Objetivos

Este laboratório tem como objetivo desenvolver uma aplicação que gere um código objeto montado em Hexadecimal, em arquivo texto ASCII, no formato MIF [1], a partir de um outro arquivo texto ASCII que contém instruções Assembly MIPS disponíveis para consulta no requisito 2 do roteiro disponibilizado [3].

2. Introdução

Um computador é um sistema que contém um processador e ainda pode conter dispositivos de entrada, de saída, de memória ou, ainda, dispositivos que forneçam rede ou alguma conectividade em geral. Para compreender o funcionamento de um computador, é fundamental estudar seu processador. O processador nada mais é que a unidade central de processamento de um computador(CPU) e é responsável pela interação e conexão entre todos os programas instalados em um computador, sendo que dentro desse processo,ele interpreta informações, realiza várias outras operações e ainda pode gerar uma interface com seu usuário. Dado que o processador realiza diversas funções, muitas delas altamente complexas, a compreensão total de seu funcionamento pode ser muito trabalhosa. Assim, é usada uma abstração, que simplifica esses processos e tarefas de tal forma que seja possível compreender e explica-los [2].

A arquitetura do conjunto de instruções é uma abstração usada para padronizar instruções que serão passadas para a máquina/computador, para que os elementos da máquina consigam realizar a comunicação entre si. A arquitetura utilizada no presente trabalho é a MIPS, ou *Microprocessor without Interlocked Pipeline Stages*, uma arquitetura de microprocessadores RISC desenvolvida pela *MIPS Computer Systems*. O MIPS32, sendo tratado neste trabalho como MIPS, é uma arquitetura que consiste de uma unidade processadora de inteiros (CPU), outros dois co-processadores. Para montar uma arquitetura MIPS basta seguir os seguintes princípios de projeto [2]: 1) simplicidade oferece regularidade, que diz respeito ao fato de que as instruções em MIPS realizam operações mais simples; 2) menor significa mais rápido, o que pode ser explicado por conta do MIPS ser baseado em apenas 32 registradores, sendo que cada word também possui 32 bits no máximo; 3) agilize os casos comuns, em que o MIPS sempre busca economizar espaço e tempo e 4) um bom projeto exige bons compromissos, onde o tamanho de cada instrução MIPS é mantida em 32 bits.

O Assembly ou linguagem de montagem, é uma forma mais legível da linguagem da máquina, que é feita em binário, seguindo os princípios de uma determinada arquitetura para implementar códigos em processadores. Dessa forma, o Assembly consegue implementar, por meio de mnemônicos, instruções de variados tipos como, por exemplo, do tipo R, J e I, em que os 32 bits de cada uma são divididos de formas diferentes para implementar variadas operações. Em uma execução de um programa no computador, o código Assembly é obtido após o compilador ler o código em linguagem de alto nível. Esse código Assembly passa então por um montador, ou assembler, para que as instruções sejam montadas em binário. Antes de ser enviado para a máquina, esse código montado, ainda é processado pelo linker, que realiza a conexão entre referências de diversos arquivos. Depois do linker, o código é enviado para a máquina e ela o executa.

3. Materiais e métodos

Dado que a aplicação deve ser feita na forma de um script Python, foram utilizados um editor de texto e interpretador Python, além de poderosas ferramentas fornecidas pela própria linguagem. A base teórica deste trabalho é o livro-texto *"Computer Organization and Design, The Hardware/Sofware interface"*[2]. Com isso foi possível implementar o procedimento que se segue.

Primeiro, o arquivo texto de entrada .asm foi lido linha a linha. Cada linha foi examinada seguindo a ideia: se a linha começa com um label, com o seguinte formato "NomeDoLabel:", seu nome e endereço são armazenados em uma tabela feita através de um dicionário, cuja chave é o endereço e o valor é o nome do label. A cada linha lida, para saber quando ocorrerá a próxima instrução, foi acrescentado 1 byte ao endereço. As instruções foram armazenadas em outra tabela indexada no valor do endereço da instrução sendo

lida ali. Os saltos de 1 byte entre cada linha foi usado pois a memória no MIPS pode ser acessada de byte a byte ou de 4 bytes a 4 bytes. Ao chegar no final do arquivo, todas as informações estão armazenadas nas tabelas.

Para a leitura e montagem da seção .data, o arquivo é novamente aberto e lido linha a linha até '.text'. Cada linha é separada entre seus componentes, que são .word ou .byte e seus valores, e esses valores são armazenados numa lista cujo índice é o endereço da instrução. Os valores armazenados são processados para o formato inteiro, no caso do .byte, e, então, são convertidos juntamente com os índices para hexadecimal. Depois da conversão, o arquivo MIF, saida_data.mif, é gerado com esses endereços e valores. Vale notar que antes de escrever o conteúdo desse arquivo, foi configurado um cabeçalho seguindo o requisitado pelo próprio arquivo MIF. Esse cabeçalho pode ser visto na imagem 1, sendo que DEPTH é o número máximo de linhas de endereços e valores a serem colocadas no arquivo, o número escolhido é o suficiente para um arquivo de entrada teste com as instruções selecionadas pelo requisito 2 [3]. WIDTH diz respeito ao tamanho do número de bits, tando do endereço quanto dos valores a serem armazenados nesses endereços, no caso foi usado 32 pois é padrão do MIPS. ADDRESS_RADIX e DATA_RADIX são a base dos valores do endereço e das instruções, HEX é hexadecimal. CONTENT, BEGIN e END marcam o local que as informações montadas ficam no arquivo.

Da tabela de instruções, cada instrução armazenada ali foi divida em seus componentes, que são mnemônico, registradores, vírgulas, constantes ou endereços. Cada um desses componentes, com exceção da vírgula, oferece informações preciosas sobre como a instrução deve ser montada. Do mnemônico é extraído o opcode e, no caso de instruções do tipo R, o function. Os registradores são lidos e convertidos para seus respectivos números, independente do formato de sua máscara. De acordo com sua posição na instrução e do tipo da instrução, o registrador é colocado como rs, rd, rt ou shamt. Para valores de constantes ou de endereços, eles são colocados nos seus respectivos campos assim como os registradores. Todas essas informações são armazenadas em uma lista de dicionários. Cada linha da lista representa uma instrução, cada dicionário de instruções possui o valor de seu endereço, a instrução em assembly, o tipo da instrução e os valores de seus campos.

Dessa lista de dicionários é possível converter os valores do campo para binário e concatenar esses campos de forma que os 32 bits obtidos represente a instrução como um todo. Assim, basta converter os valores de endereço e da instrução para hexadecimal e escrever no arquivo de saída, saida_text.mif, seguindo a mesma estrutura de saida_data.mif.

Outro ponto que vale notar é que para cada arquivo de saída MIF o endereço é inicializado em $0_decimal$. O que

```
1   DEPTH = 16384;
2   WIDTH = 32;
3   ADDRESS_RADIX = HEX;
4   DATA_RADIX = HEX;
5   CONTENT
6   BEGIN
7
8   00000000 : 00000001;
9   00000001 : 00000002;
10  00000002 : 00000003;
11
12  END;
```

Figura 1. Exemplo de um arquivo MIF. Os valores usados no cabeçalho desse exemplo são os mesmos utilizados neste trabalho. Os endereços e valores entre BEGIN e END são apenas um código hipotético.

não é um problema pois para usos futuros, as aplicações que irão ler esses arquivos realizam a separação de cada um na memória da forma devida.

Seguindo esses procedimentos, foi possível implementar uma aplicação que monte as instruções dadas pelo requisito 2 do roteiro [3], para qualquer arquivo de texto como entrada.

4. Resultados

Dada a lista de instruções do requisito 2 do roteiro [3], sua montagem foi feita com sucesso por meio do script nomeado como 'requisito1.py', que também pode ser encontrado neste repositório do Github. Esse script gera dois arquivos de saída, 'saida_data.mif' e 'saida_text.mif'. Nota-se, em ambos arquivos, que o padrão de 32 bits para cada instrução foi mantido e o endereço foi mapeado de 1 a 1 byte. A princípio, qualquer máscara de registrador permitida pelo Assembly MIPS pode ser colocada dentro de uma instrução, assim como valores inteiros sinalizados. Para algumas instruções, como li, é permitida a entrada nos campos com valores hexadecimais, seja por conta da grandeza do número ou por representar um endereço de memória. Não foram realizados casos testes com arquivos de entradas diferentes.

5. Discussão e Conclusões

Durante o processo de desenvolvimento do script, as primeiras discussões foram de como seria, de forma linear, a recuperação de informações de cada instrução e de como essas informações seriam armazenadas. Primeiramente foi

salvo em uma lista de dicionários (chamada instructionDict) o endereço e instrução de cada linha respectivamente. Após essa etapa, verificamos o tipo de cada instrução de acordo com seu mnemônico e adicionamos a informação de seu tipo nos dicionários presentes na lista.Para essa tarefa ser realizada, foram declaradas 3 listas que diferenciavam o tipo de cada mnemônico.

Com isso, já tínhamos armazenado nos dicionários, as informações do tipo da instrução, a instrução em si e seu endereço. Faltando apenas formatar a instrução, a fim de recuperar os valores de cada campo dela e depois concatenar, com o objetivo de gerar um valor de 32 bits. O problema que enfrentamos foi que nem todas as instruções de um determinado tipo vão possuir a mesma quantidade de informações, assim, na função de formatar as instruções do tipo R por exemplo, foi necessário acrescentar algumas linhas de código para verificar o formato diferente dessas instruções. O mnemônico JR por exemplo: a instrução em que ele estava inserido só acompanhava outra informação (rs), enquanto outros mnemônicos do mesmo type como ADD, possuíam mais informações como o rt e rd por exemplo. No entanto, conseguimos reavaliar o código e computar as informações de maneira correta, independente da quantidade delas em cada tipo.

Apesar do trabalho ter cumprido de forma satisfatória com o objetivo de gerar um código objeto em Hexadecimal, em arquivo texto MIF, a partir de um arquivo texto ASCII de entrada com algumas instruções Assembly MIPS selecionadas, há alguns pontos que tornam o resultado obtido imperfeito. Como não foram testados outros arquivos de entrada, não se sabe como irá ocorrer o fluxo de gerar o código objeto final, tanto para a seção de .data quanto para .text. Ainda é observado que o script responsável por implementar a aplicação não possui estrutura bem definida, com comentários isolados que não explicam o trecho de código que se segue de forma plena. Nesse caso, se o código fonte for reestruturado de forma que sua leitura e compreensão sejam feitas de forma natural, com o uso melhor dos comentários, por exemplo, seria possível implementar funções e processos de forma mais eficiente, tornando o desempenho do script como um todo bem melhor que o atual.

Referências

[1] I. F. de Santa Catarina. Inicialização de memória com arquivos .mif e .hex. `https://wiki.sj.ifsc.edu.br/wiki/index.php/Inicializa%C3%A7%C3%A3o_de_mem%C3%B3ria_com_arquivos_.MIF_e_.HEX`, Acessado em 20/03/20201.

[2] D. A. Patterson and J. L. Henessy. *Computer Organization and Design, The Hardware/Software interface*. Morgan Kaupmann, 2012.

[3] F. Vidal. Laboratório 01, 2021. `https://aprender3.unb.br/pluginfile.php/635327/mod_resource/content/12/Lab01-2-2020-TB.pdf`.