Instituição: Universidade Federal de Minas Gerais
Disciplina: Organização de Computadores I
Professor: Daniel Fernandes Macedo
Monitor: Leandro Noman Ferreira
Aluna: Lorena Mendes Peixoto
Matrícula: 2017015002

Trabalho Prático 1 - Simulador MIPS

1. Introdução

Quando um programa é feito por um ser humano, em linguagem de alto nível, e solicita-se à máquina que o execute, ocorre o processo de compilação. Neste, tal linguagem é traduzida para outra equivalente, porém de baixo nível e compatível com o entendimento do computador.
Geralmente, esta segunda codificação é binária, já que a máquina “entende” apenas 0’s e 1’s (eletronicamente falando, níveis lógicos alto e baixo, dados por faixa de tensão elétrica específica).

2. Objetivos

O presente trabalho tem por objetivo simular um processador MIPS. Um código em Assembly é compilado e gera o respectivo binário, que deve ser lido e interpretado como o próprio processador faria.

3. Desenvolvimento

Primeiramente, foi feito um planejamento acerca do tipo de estrutura a ser utilizado. A escolha, inicialmente, era de ler, do arquivo, cada conjunto de 32 bits (4 bytes) em forma de *string*, de modo a facilitar as comparações e definições dos parâmetros das funções. O trabalho foi desenvolvido com base nesse modelo, mas tive dificuldade em converter as *strings* em inteiros para realizar as operações. Além disso, percebi que havia uma forma mais rápida e eficiente de implementar, mas não usaria *string*. Por esse motivo, houve um recomeço de todo o processo, desta vez utilizando todos os valores como inteiros - e não mais como vetores de *char*.

Foi criada uma estrutura de dados chamada *Instrução*, que possui os seguintes atributos: *opcode, type, operator1, operator2, destiny, shift, funct e address*. Todas as instruções do programa possuem essas características, mas seu uso depende do *type*, do *opcode* e da *funct*. Um vetor de tamanho *0x1000* é alocado de modo que comporte tal quantidade máxima de instruções. Além dele, foi criado outro vetor de inteiros que representa a memória de registradores.

A leitura do arquivo binário é feita por meio da abertura *fopen()* no modo *"rb"* (*read binary*). Na implementação, são lidos 16 bits (2 bytes) por vez, de modo a definir os atributos de uma instrução (que é composta por 32 bits). Essa escolha de quantidade foi feita pelo seguinte motivo: nos comandos do tipo R e do tipo I, é possível definir uma quantidade inteira de atributos a partir de uma única leitura. Com os primeiros 16 bits, no caso do tipo R, pode-se definir *opcode* (6 bits)*, destiny* (5 bits) e *operator1* (5 bits). Com os segundos 16 bits, definem-se *operator2* (5 bits), *shift* (5 bits) e *funct* (6 bits). No caso do tipo I, da mesma maneira, definem-se *opcode* (6 bits)*, operator1* (5 bits) e *destiny* (5 bits) na primeira leitura e *immediate* (16 bits) na segunda.

No caso das instruções do tipo J, que não podem ser definidas de forma inteira a partir de duas leituras de 16 bits, foi utilizada a seguinte estratégia: lidos os primeiros 10 bits do *address*, eles são deslocados de 16 posições para a esquerda e somados aos próximos 16 bits da segunda leitura. Dessa forma, o valor original é mantido e não há perda de informação.

Feita a leitura e a definição dos atributos de cada função, inicia-se o processo de funcionamento do programa. Uma variável chamada *PC* controla o fluxo da execução. A cada iteração realizada, ele é incrementado de 1. Quando ocorre alguma instrução de *branch* ou *jump*, essa variável recebe o valor do endereço para o qual a próxima iteração deve ser direcionada, mudando o endereço atual do vetor principal e retomando o fluxo de execução a partir do novo ponto. Ao chegar na instrução *halt*, a execução termina e o processo é finalizado.

4. Considerações

O trabalho prático foi, particularmente, um dos mais interessantes já propostos até o presente momento no curso (em comparação aos de AEDS). Fugindo da dissertação em terceira pessoa, essa é uma opinião pessoal, pois tenho grande interesse na área de *hardware* e suas proximidades. Desenvolver um programa que me coloca na posição da máquina foi **muito** legal, já que tive a oportunidade de compreender, na prática, como é feita a leitura e interpretação de um código-fonte em alto/médio nível (seja C, Assembly ou qualquer outra linguagem). A maior dificuldade que tive foi devido à má escolha de projeto que fiz inicialmente. Perdi cerca de 4 dias insistindo em uma implementação que não dava certo. Quando resolvi recomeçar, me planejei melhor e consegui fazer o trabalho em uma madrugada e mais algumas horas do dia. Para elaborar a versão entregue, devo ter gasto cerca de 10 horas (o que é muito, mas diminui com a experiência).

Em relação ao prazo de entrega do trabalho, acredito que não teria conseguido fazê-lo se fosse para a primeira data estipulada (devido à quantidade absurda de provas, trabalhos e listas de exercícios que temos no curso). Com o primeiro adiamento, de uma semana, foi possível desenvolvê-lo com mais calma, mas ainda assim com correria. Eu, pessoalmente, o teria entregue sem que funcionasse completamente, pois estava presa à primeira implementação e nas várias outras atividades de outras matérias. Com o segundo adiamento, pude me dedicar verdadeiramente ao trabalho, fazê-lo com calma e absorver

conhecimento (o que acredito que seja o principal objetivo de um trabalho). Por isso, a consideração que tenho a fazer sobre este assunto é de que dar um prazo maior para que o aluno faça o trabalho é muito importante para que o principal objetivo seja cumprido. No meu entendimento, a falta de tempo hábil para conclusão do trabalho, aliada à necessidade da pontuação para aprovação no curso, colocam em risco a absorção do conteúdo. O aluno acaba se desesperando em fazer o código funcionar e não absorve o conteúdo.

Por fim, reitero que o presente trabalho prático foi muito interessante e com certeza agregou consideravelmente aos conhecimentos esperados para a matéria de Organização de Computadores I.