Compiladores I

Nome: Ivanir Paulo Cardoso Ignacchitti 2017093437
Nome: Gustavo Ribeiro Alves Rodrigues 2018130972

● Introdução

Nesse trabalho prático foi feita a implementação de um montador (assembler) de uma linguagem assembly simples.

Um montador(assembler) é um programa que traduz a linguagem assembly para a linguagem de máquina (binários). O programa recebe como entrada um arquivo de texto contendo o código fonte da linguagem a ser traduzida e tem como saída um arquivo contendo código em linguagem de maquina.

● O Problema

Como neste trabalho estamos trabalhando com uma versão simplificada de um montador, o arquivo de entrada consiste em uma linguagem assembly simples.

Cada instrução pode conter 4 campos sendo eles:

<label:> <operador> <operando> <;comentário>

Sendo o campo **<operador>** o único obrigatório.

Além disso, o campo operador pode ser uma instrução ou uma pseudo-instrução que foram definidas no escopo do trabalho.

● Implementação

Foi implementado um montador em 2 passos.

No primeiro passo, o arquivo de entrada é lido e a partir das instruções gerada uma tabela de símbolos que é preenchida com todas as instruções que possuem uma label.

No segundo passo, o arquivo de entrada é lido novamente, porém agora os operandos são alocados. Além disso, também é feita a contagem do número de instruções e é calculada a posição da memória das instruções que possuem uma label como operando. Por último é gerado um arquivo de saída no formato necessário para o teste.

Os campos das instruções devem ser separados por espaço como sugerido na seção anterior. Além disso, o campo comentário deve ser infixado com o caractere **;** e o campo label deve ser pósfixado com o caractere **: .**

O montador foi implementado na linguagem **c++** com o auxílio da biblioteca **stl.**

O código fonte do montador é formado por uma classe principal chamada **Assembler.** Nesta classe foram implementados os métodos ***firstpass*** e ***secondpass***, eles representam os passos de um montador two pass.

Outro método implementado por essa classe é o método ***run***, que executa todo o montador gerando assim o arquivo de saída.

Falando em arquivo de saída, a classe **Assembler** possui o método ***saveFile***, responsável por fazer a criação e a escrita da tradução no arquivo de saída.

Classes **MOT** e **POT** foram implementadas para representarem a tabela de instruções e de pseudo-instruções.

Uma outra decisão do projeto foi implementar uma classe chamada Instruction.

Essa classe é responsável por settar os campos das instruções antes de serem enviadas para os métodos das classes **MOT** e **POT.**

Duas classes foram implementadas para a tabela de símbolos. A primeira classe é a classe símbolo que possui 2 atributos, label que representa a label da tabela e a value, que representa a memória que é usada pela instrução + operando caso possua. Já a segunda classe é a própria tabela de símbolos que guarda cada símbolo quando necessário durante a primeira passada do algoritmo.

Por último, a classe **FontCode** é responsável pela leitura do código fonte e as manipulações necessárias dos seus caracteres. Nela possuimos os metodos para ler uma linha e para separar uma linha pelos seus espaços.

- ## Decisões de Implementação

  A pilha se move de acordo com as instruções que alteram a variável sp, que começa setada no topo da pilha. Se nenhuma operação alterar o apontador sp, ele ficará setado no topo da pilha. Essa decisão foi tomada pois eu não sabia se esse valor seria alterado ou não por mim devido a alguma instrução qeu alteraria a posição to ponteiro.

  Os valores dos operandos podem ser uma Label porém não podem ser negativos. Isso acontece pois todo operando é uma posição de memoria e no nosso caso usar valores negativos pode gerar “resultados inesperados”.

  O location counter (**LC**) faz a contagem de qual o próximo valor de memória livre na execução dos dois passos da montagem. Em cada passagem o valor de LC começa zerado e é incrementado pelo tamanho do endereçamento da operação e do operando. Uma variável (**PN**) faz a mesma coisa. Essa decisão foi tomada seguindo o algoritmo encontrado na internet.

Todo arquivo de entrada deve ser finalizado com END, um loop eterno irá acontecer. Estou levando em conta que a sintaxe da linguagem possui o END como uma pseudo-instrução obrigatória para a leitura do arquivo. Caso não haja um END é impossível que o algoritmo passe para o segundo passo.

Não pode haver espaços em um label. Como citado acima, além disso, só pode haver um espaço entre os campos da instrução. Como não foi especificado no enunciado, foi implementado assim.

O Montador não lê linhas só com comentários. Essa decisão também foi tomada pensando na praticidade e na não especificação no enunciado.

A pseudo-instrução WORD adiciona um 0 na tradução gerada. Essa decisão aconteceu pois não ficou claro pra mim o porquê ao usar o comando WORD 0, o valor gerado no código fonte foi 0. Além disso, o enunciado deixou claro que a pseudo-instrução WORD aloca uma unidade de tamanho, não parecendo ter ligação com o seu operando.

● Como Compilar e Testar

No diretório principal do código, existe um Makefile pronto para ser usado.

- Abra o terminal no diretório principal do projeto.
- Execute o comando make.
- Um arquivo binário foi gerado com o nome Assembler
- Basta usar o comando ***./assembler input output***
- ***input:*** variável que representa o arquivo de entrada, ou seja, o código fonte.
- ***output:*** variável que representa o arquivo de saída.