# Trabalho Prático 1 – Compiladores 1

Departamento de Ciência da Computação - UFMG - 2021/1

Daniel Souza de Campos - 2018054664

Letícia da Silva Macedo Alves – 2018054443

## Sumário



## 1 Introdução

O objetivo desse trabalho é desenvolver um montador de dois passos específico para uma máquina virtual/emulador de acordo com o seu conjunto de instruções Assembly.

A função do montador é receber um programa em alguma variante de linguagem Assembly e gerar a sua tradução em sequências de bits que possam ser executadas pelo computador. Além disso, deve indicar em qual posição de memória o programa deve ser carregado e qual será o seu ponto de partida na memória.

Por ser um montador de dois passos, o processo de tradução do programa Assembly para a linguagem de máquina, acontece, como o próprio nome já diz, em duas fases.

Primeiramente, ele deve reconhecer, nas instruções, todas as referências a símbolos que não fazem parte naturalmente do conjunto de instruções e pseudo-instruções da sua linguagem Assembly. Para tal, o montador faz uso de uma tabela de símbolos que armazena pares de chave-valor na qual a chave é o símbolo em questão e o valor é o endereço no qual ele foi definido dentro do programa Assembly.

Na segunda fase, o montador deve fazer uma nova passada pelas instruções Assembly e, efetivamente, traduzir cada instrução em uma sequência de bits que o computador possa entender. Dessa vez, em posse da tabela de símbolos, será possível usá-la para traduzir os símbolos não presentes no conjunto de instruções e pseudo-instruções do montador.

## 2 Ambiente de Desenvolvimento

O trabalho foi desenvolvido em máquinas com sistema operacional Windows, na linguagem C++11 com o compilador g++. Usamos também como ferramenta de desenvolvimento o Visual Studio Code.

Além disso, foi utilizado o GitHub como sistema de versionamento de código e o trabalho está disponível em https://github.com/Pendulun/Assembler .

## 3 Sobre a Máquina Virtual

A máquina virtual alvo já estava pronta e foi projetada especificamente para esse trabalho da disciplina.

O tipo de dado que ela consegue tratar é somente inteiros e, portanto, a saída do montador deve ser apenas sequências de números inteiros e não sequências de números binários.

O conjunto de instruções pode ser visto na seguinte tabela:

| Cód. | Símb. | Oper. | Definição | Ação |
|---|---|---|---|---|
| 0 | HALT | | Parada | |
| 1 | LOAD | R M | Carrega Registrador | Reg[R] ← Mem[M + PC] |
| 2 | STORE | R M | Armazena Registrador | Mem[M + PC] ← Reg[R] |
| 3 | READ | R | Lê valor para registrador | Reg[R] ← "valor lido" |
| 4 | WRITE | R | Escreve conteúdo do registrador | "Imprime" Reg[R] |
| 5 | COPY | R1 R2 | Copia registrador | Reg[R1] ← Reg[R2] * |
| 6 | PUSH | R | Empilha valor do registrador | AP ← AP – 1; Mem[AP] ← Reg[R] |
| 7 | POP | R | Desempilha valor no registrador | Reg[R] ← Mem[AP]; AP ← AP + 1 |
| 8 | ADD | R1 R2 | Soma dois registradores | Reg[R1] ← Reg[R1] + Reg[R2] * |
| 9 | SUB | R1 R2 | Subtrai dois registradores | Reg[R1] ← Reg[R1] – Reg[R2] * |
| 10 | MUL | R1 R2 | Multiplica dois registradores | Reg[R1] ← Reg[R1] × Reg[R2] * |
| 11 | DIV | R1 R2 | Dividendo entre dois registradores | Reg[R1] ← dividendo(Reg[R1] ÷ Reg[R2]) * |
| 12 | MOD | R1 R2 | Resto entre dois registradores | Reg[R1] ← resto(Reg[R1] ÷ Reg[R2]) * |
| 13 | AND | R1 R2 | AND (bit a bit) de dois registradores | Reg[R1] ← Reg[R1] AND Reg[R2] * |
| 14 | OR | R1 R2 | OR (bit a bit) de dois registradores | Reg[R1] ← Reg[R1] OR Reg[R2] * |
| 15 | NOT | R | NOT (bit a bit) de um registrador | Reg[R] ← NOT Reg[R] * |
| 16 | JUMP | M | Desvio incondicional | PC ← PC + M |
| 17 | JZ | M | Desvia se zero | Se PEP[zero], PC ← PC + M |
| 18 | JN | M | Desvia se negativo | Se PEP[negativo], PC ← PC + M |
| 19 | CALL | M | Chamada de subrotina | AP ← AP – 1; Mem[AP] ← PC; PC ← PC + M |
| 20 | RET | | Retorno de subrotina | PC ← Mem[AP]; AP ← AP + 1 |

*Tabela 1 - Instruções aceitas pela Máquina Virtual*

A máquina virtual também possui 4 registradores de propósito geral: R0, R1, R2 e R3. Além disso, outros 3 registradores de propósito específico:

- PC: Contém o endereço da próxima instrução a ser executada
- AP: Aponta para o elemento no topo da pilha
- PEP: Consiste em dois bits que armazenam o estado da última operação lógico/aritmética realizada, sendo que um dos bits armazena se a última operação resultou em 0 e o outro se resultou em um valor negativo.

Seguindo a tabela, como exemplo, se tivéssemos a instrução MUL R1 R2, ela deveria ser traduzida como a sequência de números: 10 1 2.

# 4 Sobre o Montador

Como foi dito antes, o montador deve receber como entrada um arquivo com extensão ".amv" contendo sequências de instruções Assembly e traduzi-las, em dois passos, em uma sequência de números inteiros que sejam executáveis pela máquina virtual.

Acontece que cada linha de instrução Assembly tem um formato bem definido com partes opcionais. Segue o formato das instruções

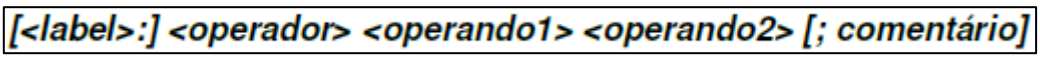


*Figura 1 - Formato das instruções Assembly aceitas*

De acordo com esse formato, toda instrução deve ter, obrigatoriamente, um operador e, dependendo da operação, 0, 1 ou 2 operadores. Além disso, ela pode ter um comentário que é identificado pelo símbolo "**;**". Comentários devem ser ignorados na tradução, já que eles servem apenas para quem está lendo o código. Além disso, é possível ter um Label seguido de "**:**" no início da instrução.

## 4.1 Labels

O Label é uma nomenclatura para a instrução que a segue e serve para indicar pontos específicos no código que podem ser referenciados ao longo do programa Assembly. Dessa forma, um Label de nome "Funcao1" pode ser usado como operando de memória em algumas instruções.

Esses Labels são exatamente os símbolos que serão incluídos na tabela de símbolos. No desenvolvimento do trabalho, **assumimos que todo Label utilizado foi definido em algum momento** no arquivo de instruções, não fazendo o tratamento de erro caso ele seja utilizado e não tenha um valor associado na tabela de símbolos.

### 4.2 Pseudo-instruções

Além das instruções indicadas na Tabela 1, existem duas pseudo-instruções para o montador:

- “WORD I”: Reserva a posição de memória e aloca com o inteiro representado por “I”
- END: Indica o final do programa para o montador

Durante o desenvolvimento, **assumimos que a pseudo-instrução WORD sempre será precedida por um Label**. Isso se deve ao fato de que não faria sentido alocar um espaço para um inteiro e não o referenciar ao longo do programa.

### 4.3 Instruções aceitas

O processo de tradução se dá linha a linha, considerando apenas linhas válidas seguindo o formato das instruções. Entretanto, **algumas coisas que tratamos para não atrapalhar o processo de tradução** são:

- Linhas em branco
- Linhas contendo apenas comentários
- Tamanho do espaçamento entre termos da instrução
- Tabs entre termos da instrução

**Assumimos que números inteiros não podem ser utilizados diretamente em operandos de memória**. Eles devem ser, primeiramente, declarados via pseudo-instrução “WORD I” junto com um Label e referenciados por esses Labels.

**Assumimos, também, que sempre é passado um registrador válido em operandos de registrador**. Da forma que o montador foi implementado, se for passado algo que tenha um número inteiro como segundo caractere, esse número será impresso na tradução. Logo, se, por exemplo, for passado “R96” no operando de registrador, será impresso o número “9” na tradução e isso deve acarretar em um **erro de execução no emulador**. Da mesma forma, se for passado “R268”, o número “2” será impresso na tradução e não dará erro de execução tratando-se do acesso a registradores no emulador, entretanto, erro de lógica pode acontecer. Se for passado “ABC”, um **erro no processo de tradução** ocorrerá.

### 4.4 Saída do Montador

O montador deve produzir um arquivo de extensão “.mv” que tenha o seguinte formato:

- A primeira linha do arquivo deve conter a String “MV-EXE”
- Linha em branco

- Linha com quatro inteiros em sequência separados por espaço, sendo eles:
    - **Tamanho do programa**: Quantas posições de memória o programa vai ocupar.
    - **Endereço de carregamento**: Posição que definimos **0 por padrão**
    - **Valor inicial da pilha**: **Mil posições de memória depois** relativa à última posição de memória ocupada pelo programa: **Endereço de carregamento + Tamanho do Programa + 1000**
    - **Entry Point do Programa**: Posição de memória onde está o primeiro operador da primeira instrução do programa
- Linha em branco
- Sequência de inteiros representando a tradução do programa

Para o Entry Point do Programa, **foi considerado sempre a primeira instrução no arquivo que esteja presente na Tabela 1**. Dessa forma, pode-se começar o programa com instruções compostas de Labels seguidos da pseudo-instrução Word sem que elas sejam consideradas pontos de entrada.

Efetivamente, **o nome do Label não influencia no ponto de entrada do programa**.

## 5 Testes Realizados

Para testar se o montador estava traduzindo corretamente e se sua saída podia ser executada normalmente pelo emulador, alguns casos de teste foram criados. Todos eles são bem simples, mas abrangem os requisitos do trabalho.

- Teste1.amv: Um programa que lê um inteiro e o imprime somado de 100;
- Teste2.amv: O mesmo que o Teste1, porém, com a ordem de instruções diferente;
- Teste3.amv: Um programa que lê um inteiro e o imprime somado de 300;
- Teste4.amv: Um programa que faz a seguinte conta ((10*3)/2)%4 e imprime seu resultado
- Teste5.amv: Programa que lê um número e faz a seguinte conta: (NUM+1)*2
- Teste6.amv: Programa que lê dois números e imprime o AND, OR e NOT de ambos.
- TestePiramide.amv: Um programa que lê um **inteiro positivo** e imprime uma sequência de números inteiros crescente de 0 até esse número e depois decrescente desse número até 0.

No Teste5.amv, percebemos uma coisa interessante. Quando temos a definição de um Label, é adicionado o seu endereço absoluto na tabela de símbolos. Se esse Label for seguido da pseudo-instrução “WORD I”, então, naquele endereço de memória estará o inteiro “I”. Se, ao longo do programa, usarmos a instrução STORE <Registrador> <Label>, o número que estava gravado nessa posição de memória poderá mudar.

O problema, e não foi testado se isso realmente acontece, é que se tivermos um Label, digamos “FUNC”, no qual é seguido por uma instrução presente na Tabela 1 e, eventualmente, realizarmos a operação STORE <Registrador> FUNC, estaríamos, efetivamente, mudando o inteiro salvo em Mem[FUNC] e, teoricamente, alterando a instrução ali salva. Dessa forma, caso fosse pedido para reexecutar a instrução em FUNC, digamos, com um CALL FUNC, estaríamos executando uma instrução diferente da original.

Isso poderia ser uma ***feature*** deveras interessante se esse comportamento for considerado correto, trazendo um dinamismo e flexibilidade para o programador.

## 6 Conclusão

O objetivo do trabalho era desenvolver um montador de dois passos que traduza uma sequência de instruções Assembly para uma máquina-virtual que aceita um conjunto específico de instruções e pseudo-instruções.

Esse objetivo foi atingido visto que todos os testes, que abrangem todas as instruções possíveis, foram traduzidos e executados com sucesso pelo emulador.

O cálculo do endereço relativo ao PC da definição de um Label apontado por um operando de memória e o tratamento de espaçamento e Tabs ao longo das instruções foram as nossas maiores dificuldades.

No final do desenvolvimento, pudemos colocar em prática o material estudado na matéria de Compiladores 1 e entender melhor o processo de tradução por um montador já que precisamos buscar outros materiais e explicações sobre o seu funcionamento.