Documentação TP1 Compiladores

Gabriel Lima Canguçu
Giovanna Louzi Bellonia
Thiago Martin Poppe

30 de julho de 2021

1 Descrição do problema

Foi solicitado a criação de um montador de 2 passos para o assembly de uma máquina virtual que foi projetada exclusivamente para a disciplina. Além do conjunto de instruções fornecido para o trabalho, o montador recebe via linha de comando o nome de um arquivo com o código assembly que devemos realizar o processo de montagem.

A saída do montador será através da **saída padrão** e seguirá um formato previamente especificado, onde a primeira linha sempre será `MV-EXE`, indicando que é um arquivo executável pelo emulador. A segunda linha será composta por 4 números separados por espaço, sendo eles (i) o tamanho do programa, (ii) o endereço de carregamento do programa, (iii) o valor inicial da pilha e (iv) o entry point do programa, ou seja, a inicialização do registrador de propósito específico PC (contador de programa). Por fim, a terceira linha será a saída do montador propriamente dita, onde teremos o mesmo código em assembly sendo representado através de inteiros correspondentes aos códigos de operação e operandos da máquina.

1.1 Observações

- O desenvolvimento do montador de 2 passos foi feito na linguagem C++ utilizando Linux (versão Ubuntu 20.04) e WSL (por alguns integrantes do grupo).

2 Definições de projeto

2.1 O montador

O montador é um tipo de programa de computador que adota um mecanismo de tradução no formato de um dicionário, onde pegamos instrução por instrução, verificamos uma tabela, e "montamos" um programa equivalente ao programa original escrito dessa vez em linguagem de máquina. Por exemplo, o código referente à instrução `LOAD` do assembly da máquina virtual utilizada no trabalho será igual à 1; enquanto que a instrução `ADD`, por exemplo, será 8. Além disso, temos uma especificação do formato da instrução, sabendo por exemplo se uma certa instrução I terá 0, 1 ou 2 operandos, além dos tipos desses operandos, sendo registradores ou uma posição de memória.

No caso do trabalho prático, foi solicitado uma implementação de um montador de 2 passos. O primeiro passo consiste em inserirmos em uma tabela todos os símbolos desconhecidos que encontrarmos ao longo do programa, isto é as *labels* juntamente com a sua posição de memória dentro do programa. O segundo passo será para realizar o processo de montagem propriamente dito, onde iremos ler novamente o arquivo de entrada traduzindo as instruções e seus operandos para os seus respectivos códigos em linguagem de máquina.

2.2 Implementação do montador

Para modelarmos o nosso montador, optamos por criar uma classe chamada `Montador` que irá lidar com todos os passos e dependências associadas com o processo de montagem, fornecendo para o usuário uma interface com as funções `discoverLabels()`, para a execução do primeiro passo, e `translate()`, para a execução do segundo passo. Quando instanciamos essa classe, a função privada `initializeTable()` é chamada no construtor, inicializando assim a tabela de tradução com o conjunto de instruções fornecidos. Tanto a tabela de tradução quanto a tabela para armazenarmos a posição de memória das *labels* foram modeladas a partir de um `map`, nativo do C++, para simplificar e tornar o processo de tradução mais eficiente.

Para obtermos o código equivalente em linguagem de máquina, para ser executado na máquina virtual, basta chamarmos a função `mount` que imprime o cabeçalho adequado do programa bem como o seu corpo em linguagem de máquina.

2.3 Definição do cabeçalho da saída

Para a criação do arquivo executável no formato aceito pela máquina virtual, devemos fornecer 4 inteiros antes da saída do montador propriamente dita, sendo eles: (i) o tamanho do programa, (ii) o endereço de carregamento, (iii) valor inicial da pilha e (iv) entry point do programa.

Através do processo de montagem, podemos inferir o tamanho do programa e também o entry point do programa. Note que podemos ter em alguns casos definições de constantes antes da primeira instrução do código assembly, sendo assim nem sempre teremos que o entry point será igual ao endereço de carregamento. Para resolver esse problema, durante a descoberta das *labels*, o nosso montador salva a posição de memória da primeira instrução encontrada, assumindo que não iremos ter definições de subrotinas (*label* com seu corpo terminando em `RET`) antes da primeira instrução a ser executada.

Para o valor do endereço de carregamento foi definido uma constante arbitrária igual a 0 por simplificação. Já o valor inicial da pilha será definido através da seguinte fórmula $K+N+1000$, onde K é o tamanho do programa e N o valor do endereço de carregamento. Sendo assim, temos que a pilha terá um tamanho de no mínimo 1000 posições antes de gerar algum conflito com a região de memória definida para o código (foi dito que um valor de 1000 posições era o suficiente).

3 Testes desenvolvidos

Além de testarmos o código base fornecido pela especificação do trabalho prático, desenvolvemos três códigos de teste adicionais, sendo eles:

1. Um programa que lê um natural n como entrada, seguido de n inteiros e imprime o maior deles. Por exemplo, a leitura dos valores `5 1 77 25 59 20` terá resultado `77`.

```
; R0 -> Valor atual
; R1 -> Maior valor
; R2 -> R2 = 1 (auxilar para decremento do contador)
; R3 -> Valor de N (contador)

const1: WORD 1

LOAD R2 const1 ; R2 = 1
READ R3        ; lendo N
READ R0

bigger: COPY R1 R0 ; Salvando o maior em R1

loop: SUB R3 R2  ; contador = contador - 1
      JZ end     ; Pulamos para o final se o contador zerar

      READ R0

      SUB R1 R0
      JN  bigger ; verificamos se R0 > R1
      ADD R1 R0  ; restauramos o valor de R1 antes do SUB

      JUMP loop

end: WRITE R1
     HALT

END
```

2. Um programa que lê um número n como entrada e imprime o n-ésimo número de Fibonacci (ou -1 caso n seja inválido). Por exemplo, a leitura do inteiro `8` resultará no número `13`.

```
; R0 -> guardar o valor de N
; R1 -> guardar o valor de a, inicialmente Fib(0) = 0
; R2 -> guardar o valor de b, inicialmente Fib(1) = 1
; R3 -> registrador para operações auxiliares
; Obs.: o registrador R2 será a resposta do Fib(N)

READ  R0         ; lendo o valor de N
STORE R0 aux     ; guardando o valor de N
```

```
LOAD R3 const1
SUB  R0 R3
JZ   print_fib1  ; se N = 1 printamos Fib(1) = 0

LOAD R3 const1
SUB  R0 R3
JZ   print_fib2  ; se N = 2 printamos Fib(2) = 1

LOAD R3 const2
LOAD R0 aux
SUB  R0 R3       ; teremos que realizar o loop N-2 vezes
JN   print_error ; se N-2 < 0, terminamos em erro

LOAD R1 const0 ; carregando A = 0
LOAD R2 const1 ; carregando B = 1
LOAD R3 const1 ; fazendo com que R3 = 1

fib: STORE R2 aux ; guardando o valor de B
ADD   R2 R1  ; fazendo B = A + B (próximo Fib)
LOAD  R1 aux ; fazendo A = B

SUB  R0 R3 ; decrementando N e verificando se paramos
JZ   print_fibN
JUMP fib

print_error: LOAD  R3 error
     WRITE R3
     HALT

print_fib1: LOAD  R3 const0
    WRITE R3
    HALT

print_fib2: LOAD  R3 const1
    WRITE R3
    HALT

print_fibN: WRITE R2
    HALT

error:  WORD -1
const0: WORD  0
const1: WORD  1
const2: WORD  2
```

```
aux: WORD 0

END
```

3. Por fim, um programa que define constantes logo de início para testar o funcionamento correto do contador de programa.

```
const100: WORD 100
const200: WORD 200
const300: WORD 300

READ R0
LOAD R1 const100
CALL add
WRITE R0
HALT

add: ADD R0 R1
     RET

END
```