Documentação trabalho prático 1

Bruce Nunes Morrow - 2016111741
Gustavo Emanuel Faria Araujo - 2017002482

30 de julho de 2021

1 Objetivos

Documentar, de forma concisa, as decisões de projeto e testes realizados com o montador desenvolvido.

2 Decisões de projeto

O montador foi implementado na linguagem C++ usando o paradigma estruturado para simplificar seu desenvolvimento. Ele encontra-se dividido em três arquivos:

- **Montador.h:** fornece os cabeçalhos das funções do montador e uma enumeração que contém as possíveis instruções *assembly* da máquina virtual (MV);
- **Montador.cpp:** fornece as funções para leitura do arquivo de entrada, identificação das instruções *assembly* da MV, preenchimento da tabela de símbolos (primeiro passo do montador) e tradução dos comandos para o código de máquina (segundo passo do montador);
- **Main.cpp:** programa principal que recebe o arquivo de entrada via linha de comando, chama as funções do montador na ordem correta e imprime o código de máquina na saída padrão.

Decidiu-se criar a enumeração supracitada para os comandos a fim de facilitar sua interpretação e aumentar a eficiência do montador. Caso contrário, uma série de comparações envolvendo *strings* seria necessária ao longo do código, que são menos eficientes do que comparações entre elementos de enumerações.

O mapeamento entre os elementos de tal enumeração e seus respectivos códigos de máquina foi feito simplesmente através de um comando **switch** − **case**, também a fim de simplificação.

A tabela de símbolos foi implementada através de uma estrutura de dados do tipo std :: unordered_map<std::string, **int**>, na qual as chaves correspondem aos *labels* e os valores às posições de memória em que irão se encontrar, respectivamente. Dessa forma, a pesquisa na tabela apresenta complexidade temporal $O(1)$.

O primeiro passo do montador lê o arquivo de entrada linha a linha, elimina espaços extras, realiza um *parsing* para os elementos da enumeração supracitada e, sempre que encontra um *label*, o armazena na tabela de símbolos. O segundo passo relê o arquivo da mesma forma e constrói o código de máquina de cada linha. Fica claro, então, que ambas etapas têm complexidade temporal $O(n)$ e, por conseguinte, o montador como um todo também.

Quanto à organização na memória, considerou-se que a pilha tem tamanho máximo de 1000 posições (conforme informado no Teams) e que é a primeira coisa carregada na memória, ou seja, ocupa as posições de 0 a 999. O registrador AP (apontador do topo da pilha) começa com valor 999, uma vez que esta cresce "para baixo". O programa em si é carregado logo após a pilha, assim começa sempre da posição 1000.

3 Testes

Para os testes, implementou-se quatro programas a fim de exercitar todas as instruções da MV. A seguir apresenta-se a descrição de cada um deles e os resultados obtidos. Entre parêntesis no título de cada subseção está o nome do arquivo com o programa *assembly* sob o diretório tp1_BruceMorrow_GustavoAraujo/tst/.

3.1 Cálculo da mediana (Mediana.amv)

Programa sugerido na descrição do trabalho. Lê cinco números inteiros e imprime a mediana deles.

Utiliza as instruções: READ e WRITE para entrada e saída; WORD para reservar espaços de memória; LOAD e STORE para movimentar os valores, uma vez que os quatro registradores não são o suficiente para armazená-los juntamente com resultados de operações; SUB para comparações de magnitude; desvios condicionais JN e JZ; HALT e END para indicar o fim.

O programa gerado se encontra a seguir.

```
MV-EXE
341 1000 999 1000
3 0 2 0 330 3 0 2 0 326 3 0 2 0 322 3 0 2 0 318 3 0 2
   0 314 1 0 307 1 1 305 9 1 0 1 3 295 18 3 1 3 292 8 2
    3 1 1 289 9 1 0 1 3 278 18 3 1 3 275 8 2 3 1 1 273
   9 1 0 1 3 261 18 3 1 3 258 8 2 3 1 1 257 9 1 0 1 3
   244 18 3 1 3 241 8 2 3 17 231 1 0 235 1 2 229 1 1
   228 9 1 0 1 3 219 18 3 1 3 216 8 2 3 1 1 213 9 1 0 1
    3 202 18 3 1 3 199 8 2 3 1 1 197 9 1 0 1 3 185 18 3
    1 3 182 8 2 3 1 1 181 9 1 0 1 3 168 18 3 1 3 165 8
   2 3 17 155 1 0 160 1 2 153 1 1 152 9 1 0 1 3 143 18
   3 1 3 140 8 2 3 1 1 136 9 1 0 1 3 126 18 3 1 3 123 8
    2 3 1 1 121 9 1 0 1 3 109 18 3 1 3 106 8 2 3 1 1
   105 9 1 0 1 3 92 18 3 1 3 89 8 2 3 17 79 1 0 85 1 2
   77 1 1 76 9 1 0 1 3 67 18 3 1 3 64 8 2 3 1 1 60 9 1
   0 1 3 50 18 3 1 3 47 8 2 3 1 1 44 9 1 0 1 3 33 18 3
   1 3 30 8 2 3 1 1 29 9 1 0 1 3 16 18 3 1 3 13 8 2 3
   17 3 1 0 10 4 0 0 -1 0 1 0 0 0 0 0
```

Para a entrada $[4, 10, 9, -2, 4]$, a saída foi 4; e para a entrada $[1, 3, 0, -2, 10]$, foi 1, conforme esperado.

3.2 Sequência de Fibonacci (Fibonacci.amv)

Programa sugerido na descrição do trabalho. Lê um inteiro n e imprime o n-ésimo termo da sequência de Fibonacci de acordo com a tabela abaixo:

n	1	2	3	4	5	6	...
$Fibonacci(n)$	0	1	1	2	3	5	...

Utiliza as instruções: READ e WRITE para entrada e saída; WORD para armazenar constante e reservar espaço de memória; LOAD e STORE para movimentar os valores, uma vez que os quatro registradores não são o suficiente para armazená-los juntamente com resultados de operações; operações aritméticas SUB e ADD; desvio incondicional JUMP e condicional JZ; HALT e END para indicar o fim.

O programa gerado se encontra a seguir.

```
MV-EXE
44  1000  999  1000
3  0  1  1  37  1  2  34  9  2  1  1  3  28  9  0  1  17  20  9  0  1  17  11
     2  3  15  8  3  2  1  2  9  16  -16  4  3  16  2  4  2  0  1
```

Para a entrada 1, o resultado foi 0; para 3, foi 1; e para 8, foi 13, conforme esperado.

3.3 Pilha e funções (PushPopCall.amv)

Imprime as constantes 1 e 10 através do uso da pilha e de chamadas de função.

Utiliza as instruções: WORD para armazenar as constantes; LOAD para carregá-las nos registradores; PUSH e POP para usar a pilha; CALL e RET para chamada da função que imprime os valores; WRITE para a impressão; HALT e END para indicar o fim.

O programa gerado se encontra a seguir.

```
MV-EXE
24  1000  999  1000
1  0  18  1  1  16  6  0  19  5  6  1  19  1  0  7  2  4  2  20  0  1  10
```

O resultado foi 110, conforme esperado.

3.4 Operações aritméticas e lógicas (ArithmeticAndBitwise.amv)

Imprime os resultados de operações aritméticas e lógicas *bitwise* entre as constantes 5 e 10.

Utiliza as instruções: WORD para armazenar as constantes; COPY para restaurar um dos operandos; ADD, SUB, MUL, DIV, MOD, AND, OR e NOT para as operações (nessa ordem); WRITE para imprimir os resultados; HALT e END para indicar o fim.

O programa gerado se encontra a seguir.

```
MV-EXE
73  1000  999  1000
1  0  68  1  1  64  5  2  0  8  2  1  4  2  5  2  0  9  2  1  4  2  5  2  0  10
     2  1  4  2  5  2  0  11  2  1  4  2  5  2  0  12  2  1  4  2  5  2  0  13
   2  1  4  2  5  2  0  14  2  1  4  2  5  2  0  15  2  4  2  0  5  10
```

O resultado foi $[15, 5, 50, 2, 0, 0, 15]$, conforme esperado.