Universidade de Évora
Departamento de Informática
Linguagens de Programação
Ano letivo 2019 - 2020

Execução de Programas TISC

Alunos:
Luís Ressonha - 35003
Rúben Teimas - 39868

Docente:
Teresa Gonçalves

1 de Junho de 2020

Indice

1 Introdução

O trabalho final da *UC*, Linguagens de Programação, tem como objetivo desenvolver uma máquina *TISC*.

Previamente, no 1º trabalho desta *UC*, implementámos a parte da máquina que é responsável por ler e carregar as instruções de um programa *TISC* para a memória de instruções.

Essa implementação foi utilizada neste trabalho juntamente com as restantes estruturas que, de forma conjunta, permitem a exececução de um programa *TISC*, formando assim a máquina. Para a implementação da máquina foi necessário desenhar também os *RA*(registo de ativação).

De forma a podermos reaproveitar o código do trabalho anterior, este foi também implementado utilizando a linguagem *Java* juntamente com o analisador lexical *JLex* e o analisador sintático *CUP*. A linguagem adoptada tem bastantes estruturas de dados *built-in* o que facilitou bastante o nosso trabalho.

De forma a compilar o trabalho é necessário, dentro da pasta **ficheiros**, executar o comando ***make***.

Depois de compilado basta, na mesma pasta, executar o comando ***make run*** $<$ ***../exemplos/programa.tisc*** em que **programa** é o nome do programa a ser executado.

2 Estrutura da máquina

A máquina TISC implementada por nós é composta pelos seguintes componentes:

- **Estruturas Principais**:
  - Memória de instruções;
  - Memória de execução;
  - Pilha de avaliação;
- **Estruturas Auxiliares**:
  - Armazenamento de labels;
  - Lista temporária de argumentos;
- **Registos**:
  - Program Counter;
  - Environment pointer;

2.1 Memória de Instruções

A memória de instruções, previamente definida no trabalho anterior, é a estrutura onde são guardadas as instruções lidas de um programa *TISC*.

É representada por um *ArrayList* de elementos *Instrucao*, a classe abstrata partilhada por todas as instruções.

A escolha desta estrutura de dados deve-se ao facto da mesma ter uma inserção e acesso de complexidade temporal constante ,*O(1)*, bem como um tamanho dinâmico, o que permite adicionar tantas instruções quanto necessárias.

2.2 Memória de Execução

A memória de execução é a estrutura na qual se encontram os registos de ativação dos blocos cujo o tempo de vida não expirou, ou seja, ainda são necessários.

Inicialmente, como optámos por utilizar uma linguagem de programação orientada a objetos, pensámos em criar uma classe *RegistosdeAtivacao* e representar esta estrutura como um *ArrayList* de *RegistosDeAtivacao*, contudo o enunciado diz que a memória de execução deve ser análoga a um vetor de inteiros.

Assim sendo representámos a memória de execução como um *ArrayList* de inteiros, que formam os registos de ativação. Optámos por esta estrutura pois a diferença entre a mesma e um *Vector* tem a ver com o acesso de *threads*, algo que não é usado neste trabalho.

2.3 Pilha de Avaliação

Na pilha de avaliação ficam os valores colocadas na mesma através de instruções especificas ou de resultados de operações.

Esta estrutura é representada como uma *Stack* de inteiros.

2.4 Armazenamento de Labels

Esta estrutura permite associar a cada label a posição em que se encontra na memória de instruções.

É representada através de uma *Hashtable* cujo a *key* é uma String, que representa a label, e o value é um *Integer*, que representa a posição na memória de instruções. O acesso constante e permitir associar uma chave a um valor tornam esta estrutura de dados numa escolha óbvia.

2.5 Lista Temporária de Argumentos

Permite guardar temporáriamente os argumentos passados à chamada de função pela instrução *set_ arg <inteiro>*.

É representada através de um *ArrayList* de inteiros. Esta estrutura de dados tornou-se bastante conviniente pois o seu método *add(int index, int value)* permitiu adicionar o argumento na posição correta sem qualquer esforço.

2.6 Program Counter

O *program counter* aponta para a posição, na memória de instruções, da próxima instrução a ser executada.

É representado como um *int* de modo a que seja um indice da Memória de Instruções (*ArrayList*).

2.7 Environment Pointer

Este registo aponta para o registo de ativação do bloco em execução.

À semelhança do *program counter* é representado por um *int* de modo a ser um indice da Memória de Execução (*ArrayList*).

3 Registos de Ativação

A máquina de arquitetura *TISC* está preparada para executar linguagens com âmbitos de identificadores estáticos, assim sendo os registos de ativação necessitam, para além dos elementos mais básicos, de um *Acess Link*.

Tal como se pode observar na *Figura 1*.

Figura 1: Desenho de Registo de Ativação

O registo de ativação por nós desenhado é constituído por uma parte de tamanho fixo (*Control Link*, *Acess Link*, Endereço de Retorno, Número de Args, Número de Vars) e por uma parte de tamanho variável, que depende do número de variáveis e de argumentos, (Args, Vars).

Assim sendo, um registo de ativação com 2 argumentos e 1 variável teria um tamanho de 8 inteiros, isto é, 5 inteiros da parte fixa + 2 inteiros de argumentos + 1 inteiro da variável.

4 Execução de programas *TISC*

A execução de programas *TISC* deve começar sempre na *label program* que funciona como uma função *Main*.

Para que isto se verifique implementámos uma função *comecaMain()* que procura, no Armazenamento de Labels, a posição dessa mesma label na memória e aponta o *program counter* para essa mesma posição, executando de seguida a instrução, criando assim o 1º bloco.

Antes de executar a função *comecaMain()* o *environment pointer* aponta para a posição -1, sendo este um valor arbitrário escolhido por nós.

A execução da máquina termina quando o tempo de vida do bloco mais exterior termina, ou seja, o *environment pointer* volta a apontar para a posição -1.

O fluxo de execução pode ser observado, de forma simplicada, na Figura 2.

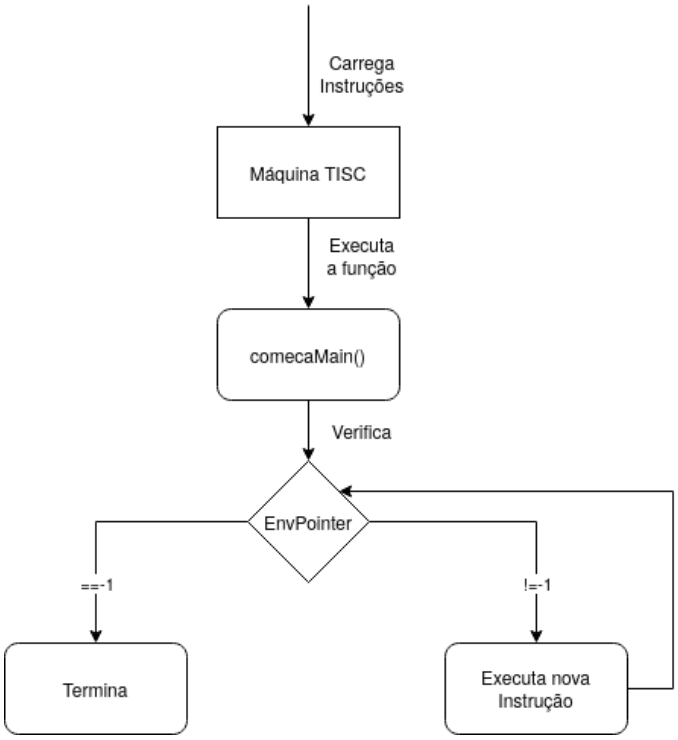

Figura 2: Fluxograma simplificado da execução da máquina

A máquina dispõe ainda de funções auxiliares relacionadas essencialmente com a obtenção do valor do *Acess Link*, de váriavéis e de argumentos.

5 Instruções

Abaixo encontram-se as instruções implementadas representadas em pseudo-código. Existem algumas diferenças entre o verdadeiro código e esta representação, não só pela sintaxe de *Java* como pela uso, no pseudo-código, de algumas funções que não existem.

5.1 Aritméticas

```
add:
    o2 = desempilha()
    o1 = desempilha()
    empilha(o1 + o2)
    PC = PC + 1

sub:
    o2 = desempilha()
    o1 = desempilha()
    empilha(o1 - o2)
    PC = PC + 1

mult:
    o2 = desempilha()
    o1 = desempilha()
    empilha(o1 * o2)
    PC = PC + 1

div:
    o2 = desempilha()
    o1 = desempilha()
    empilha(o1 / o2)
    PC = PC + 1

mod:
    o2 = desempilha()
    o1 = desempilha()
    empilha(o1 % o2)
    PC = PC + 1

exp:
    o2 = desempilha()
    o1 = desempilha()
    empilha(o1 ^ o2)
    PC = PC + 1
```

5.2 Manipulação de inteiros

```
push_int (arg):
    empilha(arg)
    PC = PC + 1
```

5.3 Acesso a variáveis

```
push_var (arg1, arg2):
    envPtAmbiente = segueAL(arg1)
    var = getVar(envPtAmbiente, arg2)
    empilha(var)
    PC = PC + 1

store_var (arg1, arg2):
    envPtAmbiente = segueAL(arg1)
    varAGuardar = desempilha()
    guardaVar(envPtAmbiente+5+numArgs()+arg2-1,
        varAGuardar)
    PC = PC + 1
```

5.4 Acesso a Argumentos

```
push_arg (arg1, arg2):
    envPtAmbiente = segueAL(arg1)
    var = getArg(envPtAmbiente, arg2)
    empilha(var)
    PC = PC + 1

store_arg (arg1, arg2):
    envPtAmbiente = segueAL(arg1)
    argAGuardar = desempilha()
    guardaArg(envPtAmbiente+5+arg2-1, argAGuardar)
    PC = PC + 1
```

5.5 Manipulação de Variáveis

```
set_arg (arg1):
    valorArg = desempilha()
    tmpArgs.adiciona(arg1-1, valorArg)
    PC = PC + 1

call (arg1, label):
    novoCL = EnvPt
    EnvPt = getSize(memExecut)-1
    novoAL = atribuiAcessLink(arg1)
    endereoRetorno = ++PC
    PC = obtemPos(label)

locals (arg1, arg2):
    numeroArgumentos = arg1
    numeroVariaveis = arg2
    memExecut.copiaArgumentos(tmpArgs)
    memExecut.alocaEspaoVar(arg2)
    PC = PC + 1

return:
    enderecoRetorno = EnvPt+2
    tmpEnvPt = EnvPt
    EnvPt = CL
    popRA()
    PC = enderecoRetorno
```

5.6 Salto

```
jump (label):
    PC = obtemPos(label)

jeq (label):
    arg1 = depempilha()
    arg2 = desempilha()

    if arg1 == arg2:
        PC = obtemPos(label)
    else:
        PC = PC + 1

jlt (label):
    argA = depempilha()
```

```
        argB = desempilha()

        if argA > argB:
            PC = obtemPos(label)
        else:
            PC = PC + 1
```

6 Output

```
print (label):
    valor = desempilha()
    imprime(valor)
    PC = PC + 1

print_str (frase):
    imprime(frase)
    PC = PC + 1

print_nl:
    imprime("\n")
    PC = PC + 1
```

7 Resultados

Depois de termos implementado todas as funcionalidades da máquina testámos o exemplos fornecidos tendo sido estes os resultados:

1. **dia.tisc:** "O dia 25 de Maio de 2013 e' o 145 -esimo dia do ano".

   Observação: Procurámos num calendário e 25 de Maio de 2013 é realmente o dia número 145 desse ano.

2. **factorial_rec.tisc:** "10 3628800".

   Observação: O fatorial de 10 é 3628800 e é feito um print do valor passado na chamada inicial(10), pelo que o resultado se encontra correto.

3. **factorial.tisc:** "3628800".

4. **fibonacci.tisc:** "fibonacci(25) = 121393 O que está certo."

   Observação: Experimentámos a correr o programa com valores diferentes de 25 sendo o output o fibonacci do valor passado juntamente com a mensagem: "O que nao está certo, devia ter dado 121393".

5. **funcfunc.tisc:** "400000
   abc = 100002 "

   Observação: Esta foi a uma das funções que não testámos, fora da execução, pelo que não sabemos se o resultado está certo, mas confiamos que sim!

6. **mdc.tisc:** "O maior divisor comum entre 3544 e 4232 é 8."

7. **mixtura.tisc:** "x = 254"

   Observação: Tal como o programa *funcfunc.tisc*, também este não foi confirmado.

8. **sethi.tisc:** "x = 100 , y = 10 , z = 1

   1000"

   Observação: Este programa foi desenhado por nós manualmente, juntamente com o desenho dos registos de ativação, pelo que os resultados coincidem.

8 Conclusão

Após uma análise do nosso código e do resultado da execução dos exemplos parece-nos que o trabalho foi concluído com sucesso.

As chaves para a concepção deste trabalho foram o desenho coeso dos registos de ativação e a revisão da apresentação "Âmbito, funções e gestão de memória".

Depois destes conceitos estarem bem cimentados poucas dificuldades surgiram dado que a implementação das instruções eram relativamente simples.

Este trabalho permitiu-nos assim explorar conceitos de baixo nivel das linguagens de programação de âmbito estático, no qual acabámos por utilizar também conceitos de outras *UC's*, principalmente *ASCI* e *Compiladores*.