TAC

Interpretador de Código de Três Endereços

Manual de Referência

por Luciano Santos

Ref. Versão 0.11

Brasília, 23 de Novembro de 2014

CONTEÚDO

LISTA DE TABELAS

INTRODUÇÃO

1.1 OBJETIVO

Este manual de referência técnica documenta o programa *TAC* (*Three Address Code*), um interpretador de código de três endereços baseado no código de alto nível de Aho et al.. Este interpretador tem fins puramente didáticos, foi projetado para utilização no contexto de uma disciplina de tradutores/compiladores, no nível de graduação.

O código de três endereços reconhecido pelo TAC é Turing completo e pode naturalmente ser utilizado como código intermediário para a geração de código de máquina, no entanto, continua sendo um código de alto nível, com as seguintes características básicas:

- toda instrução recebe até três operandos, que podem ser endereços, referências à tabela de símbolos, símbolos temporários ou constantes;
- o código manipula *símbolos*, não memória ou variáveis fortemente tipadas, de tal maneira que as operações são bastante flexíveis, e assumem que o código interpretado foi gerado corretamente, emitindo *warnings* quando possível;
- o código manipula *endereços simbólicos*, não endereços reais, ou mesmo endereços relativos ou virtuais, isso significa que, em modo de interpretação, todos os endereços têm significado apenas para o interpretador e devem ser evitadas referências a endereços por meio de constantes ou operações aritméticas.

1.2 NOTAÇÃO

Ao longo deste manual, as seguinte convenções serão seguidas:

- termos novos introduzidos pela primeira vez, nomes de programas e palavras estrangeiras virão em *itálico*;
- comandos a serem executados no *prompt*, nomes de arquivos e diretórios, nomes de opções da linha de comando, exemplos de código fonte e regras de sintaxe virão em fonte `monoespaço`;

- valores a serem substituídos pelo usuário, tais como parâmetros e arquivos de entrada, serão indicados entre os sinais < e > e virão em *<itálico>*;
- valores opcionais (que podem ser fornecidos ou não) serão indicados entre os sinais `[` e `]`, `[desse jeito]`;
- caracteres que fizerem parte de regras sintáticas virão entre aspas simples, como em `'['`.

INVOCANDO O INTERPRETADOR

TAC é um projeto de *software* livre, distribuído sob a *Apache License Version 2.0*, de janeiro de 2004, com código fonte disponível em `https://github.com/lhsantos/tac`. Ele é compatível com ambientes Linux e foi testado em ambientes Mac OS e Windows+Cygwin[1].

As seções a seguir descrevem o processo de compilação e invocação do executável do TAC.

2.1 COMPILANDO O TAC

Para compilar o TAC, são necessários os programas *flex*, *bison*, *g++* (v. 4.7+) e *make*.

Após obter o código fonte por meio do *site* ou via `git`[2], abra uma linha de comando no sistema operacional e mova o diretório corrente para a raiz do código fonte (onde estão os arquivos `LICENSE` e `Makefile`). Para compilar e gerar o executável do TAC, faça:

```
make all
```

Este comando vai criar (se não existirem) os subdiretórios `obj` e `bin`, gerar arquivos objeto intermediários em `obj` e o executável final em `bin`. Caso deseje apagar todos os arquivos gerados no processo de compilação, faça:

```
make clean
```

2.2 LINHA DE COMANDO DO TAC

O executável do TAC é um programa de linha de comando, que deve ser invocado da seguinte maneira:

```
tac [opções] <entrada>
```

A entrada deve ser um arquivo de texto com um programa TAC válido, que, por convenção, deve ter a extensão *.tac*.

É possível definir as seguintes opções na linha de comando:

[1] http://www.cygwin.com/
[2] http://git-scm.com/

Tabela 1: Opções da linha de comando

Opção	Descrição
`-v, --verbose`	ativa o modo loquaz, que mostra na saída detalhes de cada etapa da execução do interpretador
`-b, --brief`	desativa o modo loquaz
`-d, --debug`	ativa o modo de depuração, que mostra a tabela de símbolos e uma lista das instruções com todas as referências simbólicas resolvidas, antes de iniciar a execução
`-s, --step`	ativa o modo passo a passo, que mostra a instrução atual sendo executada para cada iteração do programa

Se os modos loquaz, de depuração, ou passo a passo estiverem ativados, recomenda-se redirecionar a saída do interpretador para um arquivo de texto, da seguinte maneira:

```
tac -vds <entrada> > out.txt
```

ARQUIVO DE ENTRADA

Um arquivo *tac* é composto por duas seções: *table* e *code*. A seção *table* é opcional e, se presente, é utilizada para declarar uma tabela de símbolos.

Espaços e linhas em branco são ignoradas. Ao longo de todo o documento, são aceitos (e ignorados) comentários no estilo C, de uma linha, i.e., na forma:

```
// isto é um comentário\n.
```

Os comentários podem vir ao final de qualquer linha do arquivo, inclusive ocupando uma linha própria.

As seções a seguir descrevem em detalhes a sintaxe das duas seções de um arquivo *tac*.

3.1 TABELA DE SÍMBOLOS

A seção *table* é formada pelo marcador `.table` seguido de zero ou mais *definições de símbolo*, uma por linha. Uma definição de símbolo é da forma:

```
<tipo> <nome> ['['<tamanho do vetor>']'] ['=' <inicializador>]\n
```

O *tipo* de um símbolo pode ser `char`, `int` ou `float`, respectivamente, caracteres, números inteiros e números de ponto flutuante. Se definido, *tamanho do vetor* deve ser um número inteiro maior que zero ou vazio – `[]`, caso em que um *inicializador de vetor* é obrigatório.

O inicializador pode ser *simples*, i.e., uma constante inteira, um literal de caractere ou uma constante de ponto flutuante; ou pode ser um inicializador de vetor, que é uma lista de constantes do mesmo tipo, entre chaves – `{` e `}`, separadas por vírgula `','`. Inicializadores simples seguem a mesma sintaxe que a linguagem ANSI C.

Não são realizadas conversões implícitas para os inicializadores: um inicializador deve ser exatamente do mesmo tipo que seu símbolo, isto é, símbolos inteiros devem ser inicializados por uma constante inteira, vetores de caracteres devem ser inicializados com inicializadores de vetor com literais de caractere, e assim por diante. Inicializadores de vetor devem ter o mesmo número de elementos que o tamanho de vetor, se declarado.

Um caso especial de inicializador é um *literal de string* – caracteres entre aspas duplas, que é um *syntactic sugar* para inicializador de vetor de literal de caracteres.

Os nomes dos símbolos devem ser identificadores ANSI C únicos, não é permitido que dois símbolos tenham o mesmo nome.

As seguinte entrada é uma tabela de símbolos válida:

```
// exemplo de tabela de símbolos
.table
int n = 5
int m = 0xCAFEBABE
char c = 'A'
float f = 5.3f
int vetor[] = {1, 2, 3, 5}
float vf[4] = {1f, 2.0, .3, 5F}
char _char_array[] = {'H', 'e', 'l', 'l', 'o', '\0'}
char string1 [] = "Hello world!"
```

3.2 SEÇÃO DE CÓDIGO

A seção *code* é formada pelo marcador `.code` seguido de zero ou mais *instruções*, uma por linha. Uma instrução é da forma:

```
[rótulo] <nome> [<parâmetro>] [',' <parâmetro>] [',' <parâmetro>]\n
```

Um rótulo é um identificador ANSI C seguido de `':'`. Um rótulo deve ser um identificador único no programa, não podendo dois rótulos diferentes ou um rótulo e um símbolo qualquer da tabela utilizarem o mesmo identificador. Pode haver quebra de linhas e linhas em branco entre um rótulo e a instrução que vem logo em seguida. O rótulo especial `main:` é utilizado para indicar o ponto de partida do programa.

A sintaxe e a semântica de cada uma das instruções aceitas são descritas no Capítulo 5. Apenas para ilustração, o trecho abaixo é um arquivo de entrada válido:

```
// mostra um vetor na tela
.table
int v[5] = {5, 4, 3, 2, 1}
int size = 5
.code
main:
// $0 = 0, $1 = size - 1
```

```
mov $0, 0
sub $1, size, 1
// while $0 < size
L1:
slt $2, $0, size
brz L2, $2
// print v[$0]
mov $2, &v
mov $2, $2[$0]
print $2
// if $0 < size - 1, print ", "
slt $2, $0, $1
brz L3, $2
print ','
print ' '
L3:
add $0, $0, 1
jump L1 // loop
L2:
println
```

MODELO DE MEMÓRIA DO TAC

O TAC não é uma máquina virtual e não permite a manipulação direta de memória, mas apenas de símbolos. Por esse motivo, os endereços acessados em tempo de execução têm semântica definida apenas para os mecanismos internos interpretador e não devem ser manipulados da mesma maneira que um código *assembly* manipula endereços de máquina (real ou virtual).

Existem quatro grandes seções internas na memória global, distintas e não necessariamente contíguas:

- a seção *heap* armazena as variáveis estáticas definidas na tabela de símbolos;
- a seção *text* armazena as instruções;
- a *pilha* armazena variáveis locais a cada contexto (inclusive parâmetros de função) e pode ser expandida em tempo de execução, com as instruções *push* e *pop* (Seção 5.6);
- a seção de *memória dinâmica* armazena os símbolos alocados dinamicamente (Seção 5.8).

O TAC é uma máquina de 32 bits, podendo endereçar 2^{32} **símbolos**. As seções *heap* e *text* juntas, podem conter até $\lfloor \frac{2^{32}}{3} \rfloor$ símbolos, considerando-se cada instrução como um símbolo. A seção de pilha pode armazenar até $\lfloor \frac{2^{32}}{3} \rfloor$ símbolos e, por fim, a seção de memória dinâmica pode armazenar até $\lfloor \frac{2^{32}}{3} \rfloor + 1$ símbolos.

Cada chamada à instrução *call* (Seção 5.7) cria um novo *contexto*. O contexto ocupa um espaço na pilha global, chamado de *frame*, onde estão os parâmetros da chamada de função atual e possivelmente outros símbolos necessários ao contexto. Chamadas a *push* ampliam a pilha global e chamadas a *pop* diminuem esta pilha, de tal maneira que é possível corromper o *frame* atual e/ou invadir frames de chamadas anteriores, se a pilha não for manipulada corretamente.

Cada contexto possui símbolos adicionais e exclusivos que não ficam na pilha global, mas são gerenciados separadamente pelo interpretador. Dentre estes símbolos, estão até 1024 *temporários*, que são alocados e desalocados sob demanda e também o endereço de retorno da chamada de função atual.

A Seção 5.4 descreve a instrução *mov*, que permite, dentre outras possibilidades, obter o endereço simbólico de valores da tabela de símbolos. Além disso, conforme descrito no Capítulo 5, existem registradores especiais que guardam os endereços de *frames* e da pilha, do *program counter* e de retorno de função.

Pode-se assumir que os endereços dentro das seções de memória seguem a ordem crescente. Incrementar um endereço simbólico, desde que respeitados os limites das seções e dos dados, permite acessar o próximo símbolo na pilha, a próxima variável (ou elemento de vetor) ou a próxima instrução. Analogamente, decrementar um endereço simbólico acessa o símbolo anterior.

INSTRUÇÕES

Para todas as instruções a seguir, valem as seguintes definições:

1. Um **contexto** é uma abstração em tempo de execução para resolver *parâmetros de função* e *temporários*. O programa sempre inicia sua execução no contexto raiz, e cada chamada de subrotina empilha um novo contexto (desempilhado quando a subrotina retorna). A pilha de execução (que também contém os parâmetros) é global e cada contexto utiliza uma seção dela, porém os temporários são alocados separadamente e são exclusivos para cada contexto.

2. Um **parâmetro de função** é da forma `#`*n*, onde *n* é um inteiro não negativo. Esta expressão indica o enésimo parâmetro empilhado no contexto de uma subrotina, iniciando em zero. Por exemplo, se uma função é chamada com 3 parâmetros, então `#0` indica o primeiro parâmetro colocado na pilha, `#1`, o segundo, e assim por diante.

3. Um **temporário** é da forma `$`*n*, onde *n* é um inteiro não negativo. Esta expressão indica o enésimo temporário alocado para o contexto atual, iniciando em zero. Por exemplo, `$0` indica o primeiro temporário no contexto atual, `$1`, o segundo, e assim por diante. Além disso, os temporários especiais são definidos:

   - `$s` retorna o endereço simbólico do elemento no topo da pilha;
   - `$f` retorna o endereço simbólido do início da seção da pilha relativa ao contexto atual;
   - `$pc` retorna o endereço simbólido da instrução sendo executada;
   - `$ra` retorna o endereço simbólico de retorno do contexto atual;

4. Um **endereçável** é um elemento que possui endereço. Endereçáveis válidos podem ser parâmetros de função e símbolos da tabela. Observe que um rótulo não é um endereçável, pois indica o endereço de uma instrução, mas não tem um endereço próprio. Um rótulo é uma constante.

5. Um **destino** é um elemento que pode receber valores. Destinos válidos são parâmetros de função, símbolos e temporários.

6. Um **operando** é um elemento que contém valor. Operandos válidos são parâmetros de função, símbolos, temporários, constantes e rótulos.

Se uma instrução qualquer calcula valores e salva em um destino, a validade desta atribuição será verificada apenas em tempo de execução. Caso a expressão calculada seja de um tipo diferente do destino (e o destino não seja um temporário), um *warning* será emitido. Temporários sempre passam a ter o tipo de qualquer expressão que recebam. Se o destino não for um temporário, a expressão será calculada por meio de conversão implícita para o tipo do destino.

Os valores só são salvos no destino ao final da instrução, de tal modo que é possível referenciar o mesmo elemento tanto nos operandos como no destino.

Por fim, o nome de um símbolo sempre se refere ao valor deste símbolo, não ao seu endereço. Na Seção 5.4, é definida a sintaxe para obter-se o endereço de um símbolo. O nome de um vetor refere-se ao valor do primeiro elemento de um vetor. O endereço de um vetor refere-se ao endereço do primeiro elemento de um vetor.

5.1 LÓGICO-ARITMÉTICAS

Todas as instruções lógico-aritméticas são da forma:

```
<nome> <destino>, <operando> [, <operando>]
```

Devem ser respeitadas as considerações sobre tipo feitas no início deste capítulo. A seguir, são descritas todas as instruções lógico-aritméticas.

5.1.1 *add*

sintaxe:
`add <destino>, <operando>, <operando>`
semântica:
soma os dois operandos e atribui o resultado ao destino.

5.1.2 *sub*

sintaxe:
`sub <destino>, <operando`$_1$`>, <operando`$_2$`>`

semântica:
subtrai *operando*$_2$ de *operando*$_1$ e atribui o resultado ao destino.

5.1.3 *mul*

sintaxe:
`mul <destino>, <operando>, <operando>`
semântica:
multiplica os dois operandos e atribui o resultado ao destino.

5.1.4 *div*

sintaxe:
`div <destino>, <operando`$_1$`>, <operando`$_2$`>`
semântica:
divide *operando*$_1$ por *operando*$_2$ e atribui o resultado ao destino.

5.1.5 *and*

sintaxe:
`and <destino>, <operando`$_1$`>, <operando`$_2$`>`
semântica:
se *operando*$_1$ é diferente de zero e *operando*$_2$ é diferente de zero, atribui `1` ao destino, caso contrário, atribui `0`.

5.1.6 *or*

sintaxe:
`or <destino>, <operando`$_1$`>, <operando`$_2$`>`
semântica:
se *operando*$_1$ é igual a zero e *operando*$_2$ é igual a zero, atribui `0` ao destino, caso contrário, atribui `1`.

5.1.7 *minus*

sintaxe:
`minus <destino>, <operando>`
semântica:
atribui a destino o valor do operando multiplicado por -1.

5.1.8 *not*

sintaxe:
`not <destino>, <operando>`
semântica:
se *operando* é igual a zero, atribui `1` ao destino, caso contrário, atribui `0`.

5.1.9 *band*

sintaxe:
`band <destino>, <operando>, <operando>`
semântica:
os operandos serão lidos como inteiros, realiza a operação *and bit* a *bit* entre os operandos e atribui o resultado ao destino.

5.1.10 *bor*

sintaxe:
`bor <destino>, <operando>, <operando>`
semântica:
os operandos serão lidos como inteiros, realiza a operação *or bit* a *bit* entre os operandos e atribui o resultado ao destino.

5.1.11 *bxor*

sintaxe:
`bxor <destino>, <operando>, <operando>`
semântica:
os operandos serão lidos como inteiros, realiza a operação *xor bit* a *bit* entre os operandos e atribui o resultado ao destino.

5.1.12 *shl*

sintaxe:
`shl <destino>, <operando`$_1$`>, <operando`$_2$`>`
semântica:
os operandos serão lidos como inteiros, move *operando*$_1$ *operando*$_2$ *bits* à esquerda e atribui o resultado ao destino.

5.1.13 *shr*

sintaxe:
`shr <destino>, <operando1>, <operando2>`
semântica:
os operandos serão lidos como inteiros, move $operando_1$ $operando_2$ *bits* à direita e atribui o resultado ao destino.

5.1.14 *bnot*

sintaxe:
`bnot <destino>, <operando>`
semântica:
o operando será lidos como um inteiro, realiza a operação *not bit* a *bit* sobre o operando e atribui o resultado ao destino.

5.1.15 *mod*

sintaxe:
`mod <destino>, <operando1>, <operando2>`
semântica:
os operandos serão lidos como inteiros, faz a divisão inteira de $operando_1$ por $operando_2$ e atribui o resto ao destino.

5.2 RELACIONAIS

Todas as instruções relacionais são da forma:

`<nome> <destino>, <operando1>, <operando2>]`

Devem ser respeitadas as considerações sobre tipo feitas no início deste capítulo. A seguir, são descritas todas as instruções relacionais.

5.2.1 *seq*

sintaxe:
`seq <destino>, <operando1>, <operando2>`
semântica:
se $operando_1$ e $operando_2$ forem iguais, atribui `1` ao destino, caso contrário, atribui `0`.

5.2.2 *slt*

sintaxe:
`slt <destino>, <operando1>, <operando2>`
semântica:
se *operando*$_1$ for menor que *operando*$_2$, atribui `1` ao destino, caso contrário, atribui `0`.

5.2.3 *sleq*

sintaxe:
`sleq <destino>, <operando1>, <operando2>`
semântica:
se *operando*$_1$ for menor que ou igual a *operando*$_2$, atribui `1` ao destino, caso contrário, atribui `0`.

5.3 CONVERSÃO DE TIPO

Todas as instruções de conversão de tipo são da forma:

```
<nome> <destino>, <operando>
```

O tipo de origem e o de destino serão verificados. Se o destino (desde que não seja um temporário ou um símbolo alocado dinamicamente) ou a origem forem de tipos diferentes dos esperados pelas instruções, um *warning* será gerado. Os valores lidos que não estiverem no tipo de origem não são implicitamente convertidos, mas lidos no exato estado de seus *bytes*.

A seguir, são descritas todas as instruções de conversão de tipo.

5.3.1 *chtoint*

sintaxe:
`chtoint <destino>, <operando>`
semântica:
o operando será lido como um caractere, converte o valor do caractere para um inteiro e atribui ao destino.

5.3.2 *chtofl*

sintaxe:
`chtofl <destino>, <operando>`
semântica:
o operando será lido como um caractere, converte o valor do caractere para um número de ponto flutuante e atribui ao destino.

5.3.3 *inttoch*

sintaxe:
`inttoch <destino>, <operando>`
semântica:
o operando será lido como um inteiro, converte o valor do inteiro para um caractere (com possíveis perdas) e atribui ao destino.

5.3.4 *inttofl*

sintaxe:
`inttofl <destino>, <operando>`
semântica:
o operando será lido como um inteiro, converte o valor do inteiro para um número de ponto flutuante e atribui ao destino.

5.3.5 *fltoch*

sintaxe:
`fltoch <destino>, <operando>`
semântica:
o operando será lido como um número de ponto flutuante, converte a parte inteira do valor do número de ponto flutuante para um caractere (com possíveis perdas) e atribui ao destino.

5.3.6 *fltoint*

sintaxe:
`fltoint <destino>, <operando>`
semântica:
o operando será lido como um número de ponto flutuante, converte a parte

inteira do valor do número de ponto flutuante para um inteiro (com possíveis perdas) e atribui ao destino.

5.4 ATRIBUIÇÃO

A instrução de atribuição – `mov` – copia o valor de um operando em um destino. No entanto, existem várias versões desta instrução, que estão descritas na Tabela 2.

É importante lembrar que as considerações sobre tipos feitas no início deste capítulo devem ser observadas.

5.5 CONTROLE DE FLUXO

As instruções de controle de fluxo causam desvios na execução do programa, que podem ser condicionais ou incondicionais. Elas são descritas a seguir.

5.5.1 *brz*

sintaxe:
`brz <operando1>, <operando2>`
semântica:
se *operando*$_2$ for zero, a próxima instrução a ser executada será a que estiver no endereço simbólico dado por *operando*$_1$, caso contrário, será a do endereço atual mais um.

5.5.2 *brnz*

sintaxe:
`brnz <operando1>, <operando2>`
semântica:
se *operando*$_2$ for diferente de zero, a próxima instrução a ser executada será a que estiver no endereço simbólico dado por *operando*$_1$, caso contrário, será a do endereço atual mais um.

5.5.3 *jump*

sintaxe:
`jump <operando>`

semântica:
a próxima instrução a ser executada será a que estiver no endereço simbólico dado por *operando*.

5.6 PILHA

O interpretador possui uma pilha global de símbolos. As instruções a seguir operam sobre esta pilha.

5.6.1 ***push***

sintaxe:
`push <operando>`
semântica:
coloca *operando* no topo da pilha global.

5.6.2 ***pop***

sintaxe:
`pop <destino>`
semântica:
remove o símbolo no topo da pilha global e atribui seu valor a *destino* (as considerações sobre tipos feitas no início deste capítulo devem ser observadas).

5.7 SUBROTINAS

O interpretador suporta diretamente o conceito de subrotina. Quando uma subrotina é chamada, um novo contexto é criado e empilhado, o contador de temporários é reiniciado e todos os temporários alocados a partir deste momento pertencerão ao novo contexto.

Os parâmetros de função são apelidos para posições na pilha, já que os parâmetros de uma subrotina são sempre passados pela pilha global.

5.7.1 ***param***

sintaxe:
`param <operando>`

semântica:
coloca *operando* no topo da pilha global, para ser utilizado como parâmetro; esta chamada é equivalente a `push`.

5.7.2 *call*

sintaxe:
`call <operando> [, <n>]`
semântica:
chama a subrotina cujo endereço simbólico é dado por *operando*, possivelmente com *n* parâmetros, onde *n*, quando fornecido, é uma constante inteira não negativa; reserva a seção de pilha onde estão os parâmetros para esta rotina, cria um novo contexto e armazena o endereço atual mais um como endereço de retorno.

5.7.3 *return*

sintaxe:
`return [, <operando>]`
semântica:
desaloca todos os temporários e remove remove todos os elementos da pilha relativos ao contexto atual, retorna o contador de programa para o endereço de retorno armazenado neste contexto, muda o contexto atual para o último que estava no topo da pilha de contextos e, se *operando* for fornecido, armazena-o no topo da pilha global.

5.8 CHAMADAS DE SISTEMA

Estas instruções são utilitárias, e realizam funções relacionadas ao sistema operacional.

5.8.1 *print*

sintaxe:
`print <operando>`
semântica:
imprime *operando* na saída padrão.

5.8.2 *println*

sintaxe:
`println [<operando>]`
semântica:
se *operando* for fornecido, imprime-o na saída padrão, em seguida imprime um caractere de nova linha.

5.8.3 *scanc*

sintaxe:
`scanc <destino>`
semântica:
lê da entrada padrão um único caractere e armazena-o em *destino*.

5.8.4 *scani*

sintaxe:
`scani <destino>`
semântica:
lê da entrada padrão um número inteiro (equivalente à função `scanf` da linguagem C) e armazena-o em *destino*.

5.8.5 *scanf*

sintaxe:
`scanf <destino>`
semântica:
lê da entrada padrão um número em ponto flutuante (equivalente à função `scanf` da linguagem C) e armazena-o em *destino*.

5.8.6 *mema*

sintaxe:
`mema <destino>, <operando>`
semântica:
interpreta *operando* como um inteiro sem sinal e aloca dinamicamente esta quantidade de símbolos contíguos, salvando em *destino* o endereço simbólico do primeiro símbolo; se não for possível alocar, *destino* recebe o valor 0;

para fins de conversão de tipo, símbolos alocados dinamicamente se comportam como temporários.

5.8.7 *memf*

sintaxe:
`memf <operando>`
semântica:
desaloca o(s) símbolo(s) previamente alocado(s), cujo endereço simbólico (do primeiro símbolo) é dado por *operando*.

5.8.8 *rand*

sintaxe:
`rand <destino>`
semântica:
gera um número inteiro pseudo-aleatório entre 0 e 2.147.483.647 e armazena-o em *destino*.

Tabela 2: Versões da instrução `mov`

Sintaxe	Semântica
`mov d, s`	onde *d* é um destino e *s* é um operando, simplesmente copia o valor de *s* para *d*
`mov d, *s`	onde tando *d* quanto *s* são destinos, trata *s* como um endereço simbólico e atribui o valor do elemento apontado por este endereço a *d*
`mov d, &s`	onde *d* é um destino e *s* é um endereçável, atribui o endereço simbólico de *s* a *d*
`mov d, s[i]`	onde *d* e *s* são destinos, e *i* é um operando, trata *s* como um endereço simbólico e atribui o valor do elemento apontado por este endereço mais o deslocamento *i* a *d*
`mov *d, s`	onde *d* é um destino e *s* é um operando, trata *d* como um endereço simbólico e atribui ao elemento apontado por este endereço o valor *s*
`mov *d, *s`	onde *d* e *s* são destinos, trata ambos como endereços simbólicos e atribui ao elemento apontado por *d* o valor do elemento apontado por *s*
`mov *d, &s`	onde *d* é um destino e *s* é um endereçável, trata *d* como um endereço simbólico e atribui ao elemento apontado por este endereço o endereço simbólico de *s*
`mov *d, s[i]`	onde *d* e *s* são destinos, e *i* é um operando, trata *d* e *s* como endereços simbólicos e atribui ao elemento apontado *d* o valor do elemento apontado por *s* mais o deslocamento *i*
`mov d[i], s`	onde *d* é um destino e *s* e *i* são operandos, trata *d* como um endereço simbólico e atribui ao elemento apontado *d* mais o deslocamento *i* o valor *s*
`mov d[i], *s`	onde *d* e *s* são destinos e *i* é um operando, trata *d* e *s* como endereços simbólicos e atribui ao elemento apontado *d* mais o deslocamento *i* o valor do elemento apontado por *s*
`mov d[i], &s`	onde *d* é um destino, *s* é um endereçável e *i* é um operando, atribui o endereço simbólico de *s* ao elemento apontado por *d* mais o deslocamento *i*

EXEMPLOS

6.1 FIBONACCI

```
.table
int n = 10
.code
fibonacci:
// if first parameter is less than 1, set $0 to 1, otherwise to 0
slt $0, #0, 1
brz L1, $0 // if $0 is zero, goto L1
return 0
L1:
// if first parameter is less than 2, set $0 to 1, otherwise to 0
slt $0, #0, 2
brz L2, $0 // if $0 is zero, goto L2
return 1 L2:
// call fibonacci with value (#0 - 1)
sub $0, #0, 1
param $0
call fibonacci, 1
pop $0 // get result on $0
// call fibonacci with value (#0 - 2)
sub $1, #0, 2
param $1
call fibonacci, 1
pop $1 // get result on $1
// adds $0 and $1 and returns
add $0, $0, $1
return $0
main:
param n
call fibonacci, 1
pop $0
println $0
```

6.2 QUICKSORT

```
// Quicksort algorithm, as proposed by Cormen et.  al
.table
int v[12] = {12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1}
int size = 12
.code
// swaps two array elements
swap:
mov $0, #0[#1]
mov $1, #0[#2]
mov #0[#1], $1
mov #0[#2], $0
return
//partition(A, p, r)
//     x = A[r]
//     i = p - 1
//     for j = p to r - 1
//          if A[i] <= x
//          i = i + 1
//          swap(A, i, j)
//     swap(A, i + 1, r)
//     return i + 1
partition:
mov $0, #0[#2]
sub $1, #1, 1
sub $3, #2, 1
P0:
sleq $2, #1, $3
brz P1, $2 mov $2, #0[#1]
sleq $2, $2, $0
brz P2, $2
add $1, $1, 1
param #0
param $1
param #1
call swap, 3
P2:
add #1, #1, 1
```

```
jump P0
P1:
param #0
add $1, $1, 1
param $1
param #2
call swap, 3
return $1
//quicksort(A, p, r)
//     if p < r
//          q := partition(A, p, r)
//          quicksort(A, p, q - 1)
//          quicksort(A, q + 1, r)
quick:
slt $0, #1, #2
brz Q1, $0
param #0
param #1
param #2
call partition, 3
pop $0
param #0
param #1
sub $1, $0, 1
param $1
call quick, 3
param #0
add $1, $0, 1
param $1
param #2
call quick, 3
Q1:
return
main:
// call quick(&v, 0, size - 1)
mov $0, &v
param $0
param 0
sub $0, size, 1
```

```
param $0
call quick, 3
// $0 = 0, $1 = size - 1
mov $0, 0
sub $1, size, 1
// while $0 < size
L1:
slt $2, $0, size
brz L2, $2
// print v[$0]
mov $2, &v
mov $2, $2[$0]
print $2
// if $0 < size - 1, print ", "
slt $2, $0, $1
brz L3, $2
print ','
print ' '
L3:
add $0, $0, 1
jump L1 // loop
L2:
println
```

BIBLIOGRAFIA

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986. ISBN 0-201-10088-6.