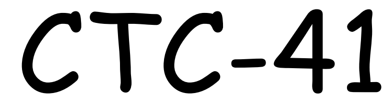
CTC-41

Compiladores

Carlos Alberto Alonso Sanches



8) Máquina virtual Tiny

Geração de código na TM

Tiny Machine (TM)

- Kenneth C. Louden (http://www.cs.sjsu.edu/~louden/cmptext/) desenvolveu a *Tiny Machine* (TM), que é uma máquina virtual que lê e simula a execução de um código de montagem simplificado.
- Há um simulador visual da TM disponível em http://david-white.net/tmvs.html#references.
- A arquitetura básica da TM possui memória de instruções, memória de dados e um conjunto de 8 registradores de uso geral, onde o registrador 7 é o contador de programa (PC).
- Início do código da TM:

```
#define IADDR_SIZE …               /* tamanho da memória de instruções */
#define DADDR_SIZE …               /* tamanho da memória de dados */
#define NO_REGS 8                  /* quantidade de registradores */
#define PC_REG 7                   /* contador de programa */
Instruction iMem[IADDR_SIZE];      /* memória de instruções */
int dMem[DADDR_SIZE];              /* memória de dados */
int reg[NO_REGS];                  /* registradores */
```

Funcionamento da TM

- TM inicializa os registradores e a memória de dados com zeros.
- Em seguida, coloca em **dMem[0]** o valor do endereço legal mais alto, isto é, **DADDR_SIZE - 1**. Isso permite a possibilidade de adição de memória à TM, pois os programas podem verificar, durante a execução, o quanto de memória está disponível.
- TM executa instruções segundo o ciclo abaixo, começando em **iMem[0]**:

```
do
    currentInstruction = iMem[reg[PC_REG]++]; /* seleciona instrução */
        ...
        /* execução da instrução corrente */
        ...
while(!(halt||error));
```

- Condições de parada:
  - Execução da instrução **HALT**
  - Situações de erro: **reg[PC_REG]**<0, **reg[PC_REG]>=IADDR_SIZE**, divisão por zero, etc.

Instruções da TM

- TM possui dois formatos básicos de instruções:
  - Instruções somente de registrador (RO):
    - Formato: **`opcode r, s, t`**
    - **`r`**, **`s`** e **`t`** devem ser índices de registradores
    - São instruções aritméticas (só há inteiros) e de entrada e saída
  - Instruções de registrador-memória (RM):
    - Formato: **`opcode r, d(s)`**
    - **`r`** e **`s`** devem ser índices de registradores
    - **`d`** é um valor inteiro que corresponde a um deslocamento na memória
    - **`d(s)`** representa o endereço **`d + reg[s]`**
      - Ocorre o erro **`DMEM_ERR`** se este endereço estiver fora do limite da memória, isto é, se for negativo ou maior do que **`DADDR_SIZE - 1`**

Instruções RO

- Formato: **opcode r, s, t**

opcode	Efeito
HALT	Interrompe a execução, ignorando os três operandos
IN	**reg[r]** recebe valor inteiro lido na entrada (**s** e **t** são ignorados)
OUT	Retorna valor de **reg[r]** à saída padrão (**s** e **t** são ignorados)
ADD	**reg[r] = reg[s] + reg[t]**
SUB	**reg[r] = reg[s] - reg[t]**
MUL	**reg[r] = reg[s] * reg[t]**
DIV	**reg[r] = reg[s] / reg[t]** (pode gerar o erro **ZERO_DIV**)

Instruções RM

- Formato: `opcode r, d(s)`
  - Seja `a = d + reg[s]`

opcode	Efeito
LD	`reg[r] = dMem[a]` (carrega `reg[r]` com o valor de memória em `a`)
LDA	`reg[r] = a` (carrega e endereça `a` diretamente em reg[r])
LDC	`reg[r] = d` (carrega valor `d` diretamente em `reg[r]`, e `s` é ignorado)
ST	`dMem[a] = reg[r]` (armazena valor de `reg[r]` na posição de memória `a`)
JLT	`if(reg[r]<0) reg[PC_REG] = a` (salta para a instrução em `a` se `reg[r]` for negativo)
JLE	`if(reg[r]<=0) reg[PC_REG] = a` (salta para a instrução em `a` se `reg[r]` for menor ou igual a 0)
JGE	`if(reg[r]>=0) reg[PC_REG] = a` (salta para a instrução em `a` se `reg[r]` for maior ou igual a 0)
JGT	`if(reg[r]>0) reg[PC_REG] = a` (salta para a instrução em `a` se `reg[r]` for positivo)
JEQ	`if(reg[r]==0) reg[PC_REG] = a` (salta para a instrução em `a` se `reg[r]` for igual a 0)
JNE	`if(reg[r]!=0) reg[PC_REG] = a` (salta para a instrução em `a` se `reg[r]` for diferente de 0)

Algumas características da TM

- Em todas as instruções são necessários os 3 operandos, mesmo quando são ignorados
- Todas as operações aritméticas devem ser realizadas em registradores
- Não há operações ou registradores para ponto-flutuantes
- Somente operações de carga e de armazenamento têm acesso à memória
- Há 3 modos de endereçamento: indireto (**`LD`**), direto (**`LDA`**) e imediato (**`LDC`**)
- Não há restrições para o uso do PC, que permanece disponível no registrador 7
- Não existe uma pilha de execução, nem FP ou SP: ambiente de execução deve ser mantido "manualmente"

Instruções para salto (*jump*)

- Salto incondicional
  - **`LDA 7, d(s)`**
  - Efeito: **`reg[7] = d + reg[s]`**, ou seja, salta para o endereço **`d + reg[s]`**
- Salto indireto
  - **`LD 7, 0(s)`**
  - Efeito: **`reg[7] = dMem[0 + reg[s]]`**, ou seja, salta para o endereço armazenado em **`reg[s]`**
- Salto condicional relativo
  - **`JEQ 1, 4(7)`**
  - Efeito: **`if reg[0]==0 reg[7] = 4 + reg[7]`**, ou seja, salta 5 instruções para a frente se o registrador for nulo
- Salto incondicional relativo
  - **`LDA 7, -4(7)`**
  - Efeito: **`reg[7] = -4 + reg[7]`**, ou seja, salta três instruções para trás

Ativação de funções

- TM não oferece uma instrução específica para a ativação (chamada) de funções
- Uma ativação pode ser realizada através da instrução abaixo:
  - `LD 7, d(s)`
  - Efeito: `reg[7] = dMem[d + reg[s]]`, supondo que seja esse o endereço do início do código da função
- Neste caso, é necessário guardar previamente o endereço de retorno. Por exemplo:
  - `LDA 0, 1(7)`
  - Efeito: coloca no registrador 0 o valor subsequente do PC corrente, supondo que a instrução seguinte seja o salto para a execução da função

Exemplo: cálculo de fatorial

```
{ Programa Tiny }
read x;
if 0 < x then
   fact := 1;
   repeat
      fact := fact * x;
      x := x - 1
   until x = 0;
   write fact
end
```

Linhas em branco ou que começam com asterisco são ignoradas.

As linhas de código precisam ser numeradas: são como rótulos de cada instrução, chamados de localização. No arquivo de entrada, não precisam estar em ordem crescente.

No código ao lado, a linha 4 corresponde ao início do laço.

```
0: IN 0,0,0
* reg[0] = read

1: JLE 0,6(7)
* if reg[0]<=0 reg[7] = 6 + reg[7] (if 0 < x then)

2: LDC 1,1,0
* reg[1] = 1

3: LDC 2,1,0
* reg[2] = 1

4: MUL 1,1,0
* reg[1] = reg[1]*reg[0]

5: SUB 0,0,2
* reg[0] = reg[0]-reg[2]

6: JNE 0,-3(7)
* if reg[0]!=0 reg[7] = -3 + reg[7] (until x = 0)

7: OUT 1,0,0
* write reg[1]

8: HALT 0,0,0
```

Comandos do simulador TM

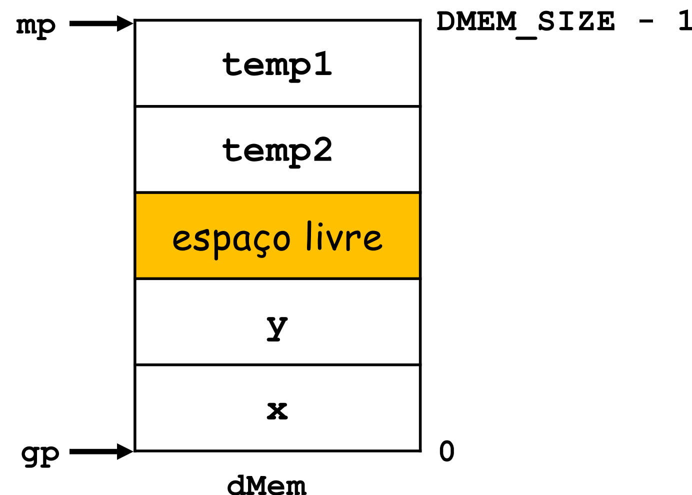
Comando	Efeito
`s <n>`	Execute **n** instruções TM (o *default* é 1)
`g`	Execute as instruções TM até instrução **HALT**
`r`	Imprima o conteúdo dos registradores
`i <b <n>>`	Imprima **n** localizações de memória de **iMem**, começando em **b**
`d <b <n>>`	Imprima **n** localizações de memória de **dMem**, começando em **b**
`t`	Alterne rastreamento de instrução
`p`	Alterne impressão de total de instruções executadas
`c`	Reinicie o simulador para uma nova execução do programa
`h`	*Help* (imprime esta lista de comandos)
`q`	Termine a simulação

Supondo que o arquivo anterior tenha nome **fact.tm**, ele pode ser executado do seguinte modo:

```
tm fact
TM simulation (enter h for help) ...
Enter command: g
Enter value for IN instruction: 7
OUT instruction prints: 5040
HALT 0,0,0
Halted
Enter command: q
Simulation done
```

Interface para geração de código

- Kenneth C. Louden disponibilizou, nos arquivos **code.h** e **code.c**, uma interface para a geração de código na TM.
- Nesta implementação, as localizações de variáveis e temporários podem ser vistas como endereços absolutos. Como a instrução **LD** exige o uso de um registrador, serão utilizados os registradores **mp** (ponteiro para o topo da memória) e **gp** (ponteiro para o início da memória global).
- Exemplo:



Variável	Endereço
temp1	0(mp)
temp2	-1(mp)
y	1(gp)
x	0(gp)

- **mp** é o registrador 5 e **gp** é o registrador 6. Também são utilizados dois acumuladores: **ac** (registrador 0) e **ac1** (registrador 1). Por simplicidade, os demais registradores (2, 3 e 4) não serão usados.

Funções para emissão de código

- **emitComment**: imprime parâmetro como comentário em linha separada, se **TraceCode** estiver ativada
- **emitRO** e **emitRM**: emissão de código para instruções RO e RM
  - Também imprime seu último parâmetro (que é um comentário) se **TraceCode** estiver ativada
- **emitSkip(n)**: pula **n** unidades na numeração (ou localização) corrente das instruções, retornando seu valor anterior
  - **emitSkip(0)**: retorna a numeração atual das instruções sem pular nada
- **emitBackup(loc)**: emissão de código para numeração **loc**
  - **emitRestore**: volta para a numeração anterior à execução de **emitBackup**
- **emitRM_Abs**: utilizada para emitir código de salto para localização obtida através de **emitSkip**
  - Transforma um endereço de código absoluto em endereço relativo ao valor corrente do PC

Esqueleto do gerador de código

- Kenneth C. Louden também disponibilizou, nos arquivos **`cgen.h`** e **`cgen.c`**, o esqueleto de um gerador de código na TM.
- Função **`void codeGen(void)`**:
  - Escreve comentários iniciais
  - Gera um prólogo padrão para ajustar o ambiente no início da execução:
    - Carrega o registrador **`mp`** com a localização de memória legal mais alta (valor colocado por TM no endereço zero no início da execução)
    - Apaga o endereço zero
  - Chama a função **`cGen`** para percorrer a árvore sintática
  - No final, gera a instrução **`HALT`**

```
* TINY Compilation to TM code
* File: xxxx.tm
```

```
0: LD 6,0(0)
```

```
1: ST 0,0(0)
```

```
xx: HALT 0,0,0
```

Função `cGen`

- A função **`cGen`** percorre a árvore sintática, realizando a geração de código.
- Esta árvore possui dois tipos de nós:
  - Declaração (*if*, *repeat*, *assign*, *read*, *write*): é chamado **`genStmt`**
  - Expressão (identificador, constante, operador): é chamado **`genExp`**
- As listas de irmãos são percorridas da esquerda para a direita, com chamadas recursivas para cada elemento.
- Tanto **`genStmt`** como **`genExp`** são implementados com **`switch`**, para diferenciar os possíveis casos.
- O código gerado para uma subexpressão deve deixar resultado no acumulador **`ac`**, para acesso subsequente.
- O acesso a uma variável na tabela de símbolos é realizado com **`loc = lookup(tree->attr.name)`**
  - `loc` será utilizado como deslocamento com o registrador `gp`

Código para expressão

- Considere uma expressão de operador, onde **p1** é o operando esquerdo e **p2** é o operando direito.
- Neste caso, o operando à esquerda deverá ser armazenado em um temporário, para depois ser calculado o operando à direita.
- Seja **tmpOffset** uma variável estática com valor inicial zero, usada como deslocamento da localização temporária do topo da pilha, apontada por **mp**.
- Geração de código:

```
cGen(p1);
emitRM("ST", ac, tmpOffset--, mp, "op: push left");
cGen(p2);
emitRM("LD", ac1, ++tmpOffset, mp, "op: load left");
```

  - **tmpOffset** é decrementado após cada armazenamento e incrementado após cada carga.
  - **emitRM** corresponde a colocar ou retirar da pilha
  - Operando esquerdo fica em **ac1** e direito em **ac**
  - Instrução RO pode ser gerada em seguida

Código para operador de comparação

- É utilizada uma abordagem mais geral, aplicável também para linguagens com operações lógicas. Como na linguagem C, 0 é considerado *false*, e 1 é *true*.
- Conforme o resultado da comparação, a constante 0 ou a constante 1 deve ser carregada no **ac**, isto é, no registrador 0.
- Exemplo de código gerado para o operador lógico "menor que" (<), onde seu operando esquerdo foi computado no registrador 1, e seu operando direito no registrador 0:

```
SUB 0,1,0     reg[0] = reg[1] - reg[0]
JLT 0,2(7)    if(reg[0]<0) reg[7] = 2 + reg[7]
LDC 0,0(0)    reg[0] = 0
LDA 7,1(7)    reg[7] = 1 + reg[7]
LDC 0,1(0)    reg[0] = 1
```

- Se o resultado for *true*, a segunda instrução provoca um salto para a última instrução, que coloca 1 em **ac**; caso contrário, a terceira e a quarta instruções são executadas, saltando a última.

Código para declaração *if*

- A avaliação da expressão lógica deixou 0 ou 1 no acumulador **`ac`**. Portanto, será preciso uma instrução **`JEQ`** para a geração do código da parte *else*.
- No entanto, a localização da parte *else* ainda é desconhecida, pois é necessário gerar antes o código da parte *then*.
- Utiliza-se a seguinte tática: **`savedLoc1 = emitSkip(1)`**. Deste modo, pula-se a próxima instrução, guardando-se a sua localização.
- É gerado o código da parte *then*, que será seguido de um salto incondicional sobre a parte *else*. Para isso, será preciso: **`savedLoc2 = emitSkip(1)`**.
- Depois de gerar o código da parte *else*, é possível gerar a instrução com localização **`savedLoc1`**, que faz o salto para essa parte:

```
currentLoc = emitSkip(0);
emitBackup(savedLoc1);
emitRM_Abs("JEQ", ac, currentLoc, "if: jmp to else");
emitRestore();
```

- Finalmente, é gerado na localização **`savedLoc2`** o salto sobre a parte *else* :

```
currentLoc = emitSkip(0);
emitBackup(savedLoc2);
emitRM_Abs("LDA", pc, currentLoc, "jmp to end");
emitRestore();
```

Exemplo: cálculo de fatorial

Código gerado com **`TraceCode`** ativada

```
{ Programa Tiny }
read x;
if 0 < x then
   fact := 1;
   repeat
      fact := fact * x;
      x := x - 1
   until x = 0;
   write fact
end
```

```
* TINY Compilation to TM code
* File: xxx.tm
* Standard prelude:
  0:   LD  6,0(0)   load maxaddress from location 0
  1:   ST  0,0(0)   clear location 0
* End of standard preludie.
  2:   IN  0,0,0    read integer value
  3:   ST  0,0(5)   read store value
* -> if
* -> Op
* -> Const
  4:  LDC  0,0(0)   load const
* <- Const
  5:   ST  0,0(6)   op: push left
* -> Id
  6:   LD  0,0(5)   load id value
* <- Id
  7:   LD  1,0(6)   op: load left
  8:  SUB  0,1,0    op <
  9:  JLT  0,2(7)   br if true
 10:  LDC  0,0(0)   false case
 11:  LDA  7,1(7)   unconditional jmp
 12:  LDC  0,1(0)   true case
* <- Op
* if: jump to else belongs here
* -> assign
```

Continuação do código gerado

```
{ Programa Tiny }
read x;
if 0 < x then
   fact := 1;
   repeat
      fact := fact * x;
      x := x - 1
   until x = 0;
   write fact
end
```

```
* -> Const
  14: LDC 0,1(0)   load const
* <- Const
  15:  ST 0,1(5)   assign: store value
* <- assign
* -> repeat
* repeat: jump after body comes back here
* -> assign
* -> Op
* -> Id
  16:  LD 0,1(5)   load id value
* <- Id
  17:  ST 0,0(6)   op: push left
* -> Id
  18:  LD 0,0(5)   load id value
* <- Id
  19:  LD 1,0(6)   op: load left
  20: MUL 0,1,0    op
* <- Op
  21:  ST 0,1(5)   assign: store value
* <- assign
* -> assign
* -> Op
* -> Id
  22:  LD 0,0(5)   load id value
* <- Id
  23:  ST 0,0(6)   op: push left
* -> Const
  24: LDC 0,1(0)   load const
* <- Const
  25:  LD 1,0(6)   op: load left
  26: SUB 0,1,0    op -
```

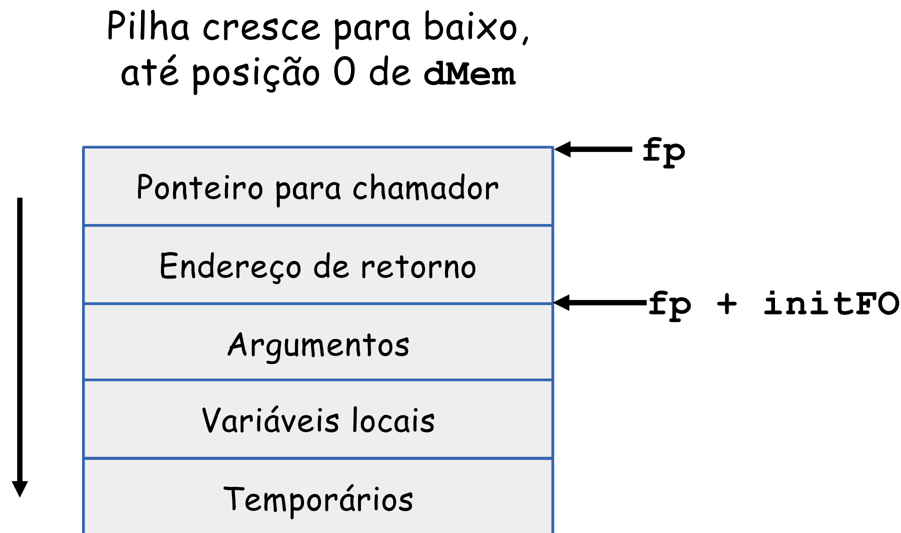
```
* <- Op
  27:  ST 0,0(5)     assign: store value
* <- assign
* -> Op
* -> Id
  28:  LD 0,0(5)     load id value
* <- Id
  29:  ST 0,0(6)     op: push left
* -> Const
  30: LDC 0,0(0)     load const
* <- Const
  31:  LD 1,0(6)     op: load left
  32: SUB 0,1,0      op = =
  33: JEQ 0,2(7)     br if true
  34: LDC 0,0(0)     false case
  35: LDA 7,1(7)     unconditional jmp
  36: LDC 0,1(0)     true case
* <- Op
  37: JEQ 0,-22(7)   repeat: jmp back to body
* <- repeat
* -> Id
  38: LD 0,1(5)      load id value
* <- Id
  39: OUT 0,0,0      write ac
* if: jump to end belongs here
  13: JEQ 0,27(7)    if: jmp to else
  40: LDA 7,0(7)     jmp to end
* <- if
* End of execution.
  41: HALT 0,0,0
```

Projeto de geração de código

- Nos arquivos **`cgen.h`** e **`cgen.c`**, também está implementada a geração de código para declarações *repeat*, *assign*, *read* e *write*.
- No entanto, como pode ser observado, o código gerado é bastante ineficiente:
  - alocação dos registradores: poderiam ser utilizados mais do que dois;
  - uso das variáveis, que são carregadas apenas para serem posteriormente armazenadas como temporários.
- Neste curso, não iremos estudar as técnicas de otimização do código produzido, mas apenas focar em que funcione corretamente.
- O objetivo do projeto final com geração de código é alterar e complementar os arquivos **`cgen.h`** e **`cgen.c`** de tal modo que sejam incluídas as seguintes funcionalidades da linguagem C-:
  - variáveis indexadas de uma dimensão (vetores);
  - comando de repetição *while* ;
  - regras de escopo em blocos aninhados;
  - funções recursivas.
- Os códigos gerados deverão ser executados através da TM.

Ambiente de execução em C-

- Devido à recursão, haverá pilha de execução, mas sem necessidade de *heap*, pois não há alocação dinâmica.
- A área global e a pilha de execução serão alocados em **dMem**, desde a posição **DADDR_SIZE - 1** até 0.
- Formato de um registro de ativação de chamada, alocado na pilha de execução:



fp: *frame pointer* do início de cada registro de ativação

Frame Offsets: deslocamentos somados a **fp**

ofpFO = 0: *old frame pointer*

retFO = -1: *return address*

initFO = -2: início das próximas alocações na pilha

Ativação de funções

- A ativação de uma função precisa da localização inicial do seu código.
- Para tornar o código gerado potencialmente realocável, será implementado um salto relativo baseado no valor corrente de **`pc`**. A função **`emitRabs`** pode ser utilizada para isso.
- Por exemplo, considere a ativação de uma função cujo código começa na localização 27, sendo que a localização corrente é 42.
  - Código não realocável: `42: LDC pc, 27(*)`
  - Código realocável: `42: LDC pc, -16(pc)`
- A sequência de ativação é realizada em parte pelo ativador (quem chama) e em parte pelo ativado (quem é chamado), como será explicado a seguir.

Alocação de argumentos e variáveis

- Argumentos e variáveis locais são alocados na pilha.
- Exemplo:

```
int f (int x, int y)
      {  int z;

         …

      }
```

Deslocamentos somados a FP

x: -2
y: -3
z: -4

- As variáveis globais, embora sejam alocadas em posições absolutas da memória, serão referenciadas com o uso do registrador **gp**, que sempre aponta para o início dessa memória (**dMem[0]**).
- Prólogo padrão para inicializar o ambiente de execução:

```
0:  LD gp,0(ac)    * carrega endereço inicial em gp (reg[gp] = dMem[ac])
1: LDA fp,0(gp)    * copia gp em fp (reg[fp] = gp)
2:  ST ac,0(ac)    * limpa endereço 0 (dMem[ac] = reg[ac])
```

Sequência de ativação de função

- Tarefas do ativador:
  - cria um novo quadro de ativação com tamanho **frameoffset**;
  - calcula e armazena os valores dos argumentos;
  - armazena **fp** corrente em **ofpFO**;
  - faz **fp** apontar para novo quadro;
  - deixa endereço de retorno no **ac**;
  - no retorno, faz **fp** apontar para quadro anterior.
- Exemplo de chamada com 2 parâmetros:

```
<código para computar 1º argumento>
 ST ac, frameoffset+initFO(fp)      * dMem[frameoffset+initFO+fp] = ac
<código para computar 2º argumento>
 ST ac, frameoffset+initFO-1(fp)    * dMem[frameoffset+initFO-1+fp] = ac
 ST fp, frameoffset+ofpFO(fp)       * armazena fp corrente (dMem[frameoffset+ofpFO+fp] = fp)
LDA fp, frameoffset(fp)             * coloca novo quadro (fp = frameoffset+fp)
LDA ac, 1(pc)                       * salva retorno em ac (ac = 1+pc)
LDA pc, …(pc)                       * salto relativo ao início da função (pc = ... +pc)
 LD fp, ofpFO(fp)                   * retira quadro corrente (fp = dMem[ofpFO+fp])
```

- Todo código de função começa armazenando o valor do **ac** em **retFO**, e termina colocando esse conteúdo de volta em **pc**:

```
ST ac, retFO(fp)    * dMem[retFO+fp] = reg[ac]
LD pc, retFO(fp)    * reg[pc] = dMem[retFO+fp]
```

Primeira instrução de cada função

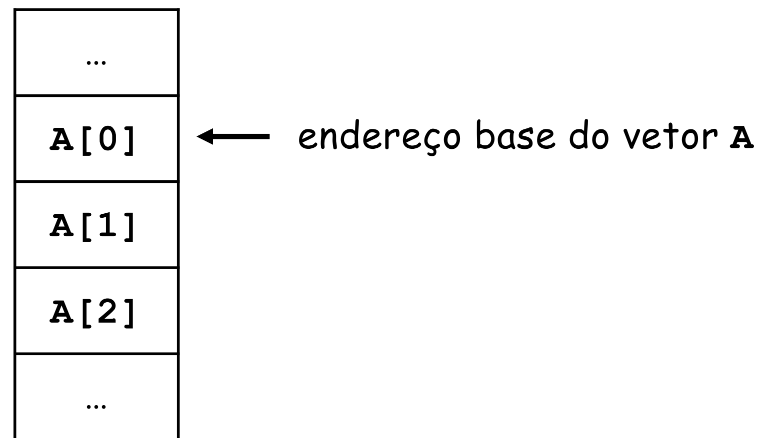
Última instrução de cada função

Computação de valores e endereços

- Considere a compilação do seguinte comando de atribuição envolvendo um vetor: `A[i] = A[i+1]`
  - `A[i]` refere-se ao endereço desta posição do vetor
  - `A[i+1]` refere-se ao valor armazenado nesta posição do vetor
- Portanto, durante a compilação, será preciso diferenciar entre endereços e valores. Isso pode ser feito através de uma *flag* (`isAddress`) passada como parâmetro na função `cGen`, que percorre a árvore sintática.
- No caso em que será preciso computar o endereço:
  - Se for variável vetor: somar o valor do índice ao endereço de base
  - Se for variável simples:
    - Se for global: somar seu deslocamento a `gp`
    - Se for local: somar seu deslocamento a `fp`
  - Em todos os casos acima, carregar o resultado em `ac`.
  - Exemplo para o caso de variável local:
    - `LDA ac, offset(fp)`

Alocação de vetores

- Os elementos dos vetores são alocados na pilha, dentro do quadro corrente, e permanecem lá enquanto o vetor existir.
- Seus valores são armazenados em ordem crescente de índices.
- Exemplo para um vetor **A**:



- Os endereços de cada elemento do são obtidos através de um simples cálculo: basta subtrair o índice do endereço base do vetor.
- Passar um vetor como parâmetro numa função significa passar seu endereço base.
  - Portanto, parâmetros de vetores são sempre passados por referência, e o cálculo do seu endereço base exige instrução **LD** ao invés de **LDA**.