Linguagem de montagem

5. Procedimentos. Caracteres, formas de endereçamento

Prof. John L. Gardenghi
Adaptado dos slides do livro

Chamada a procedimentos

- Conceitos preliminares
  - *caller*: programa que chama o procedimento
  - *callee*: procedimento chamado
  - *program counter (PC)*: registrador especial que possui o endereço de memória da instrução que o processador está executando
  - *Procedimento folha*: um procedimento que não faz chamada a algum outro (*alusão do nome: chamada de procedimentos como uma árvore*).

Chamada a procedimentos

- Fluxo de chamada e execução de procedimentos
  1. Coloque os parâmetros nos registradores
  2. Desvie a execução para o procedimento
  3. Ajuste o armazenamento para o procedimento
  4. Execute o procedimento
  5. Salve o resultado no registrador de retorno
  6. Retorne ao procedimento chamador

Uso dos registradores

- $a0 – $a3: argumentos (reg's 4 – 7)
- $v0, $v1: valores de retorno (reg's 2 and 3)
- $t0 – $t9: temporários
  - Podem ser sobrescritos pelo *callee*
- $s0 – $s7: salvos
  - Devem ser salvos e restaurados pelo *callee*
- $gp: ponteiro global para dados estáticos (r. 28)
- $sp: ponteiro para a pilha (reg 29)
- $fp: ponteiro para o *frame* (reg 30)
- $ra: endereço de retorno (reg 31)

Instruções para chamadas

- **Chamada a procedimento**: *jump and link*

  `jal ProcedureLabel`

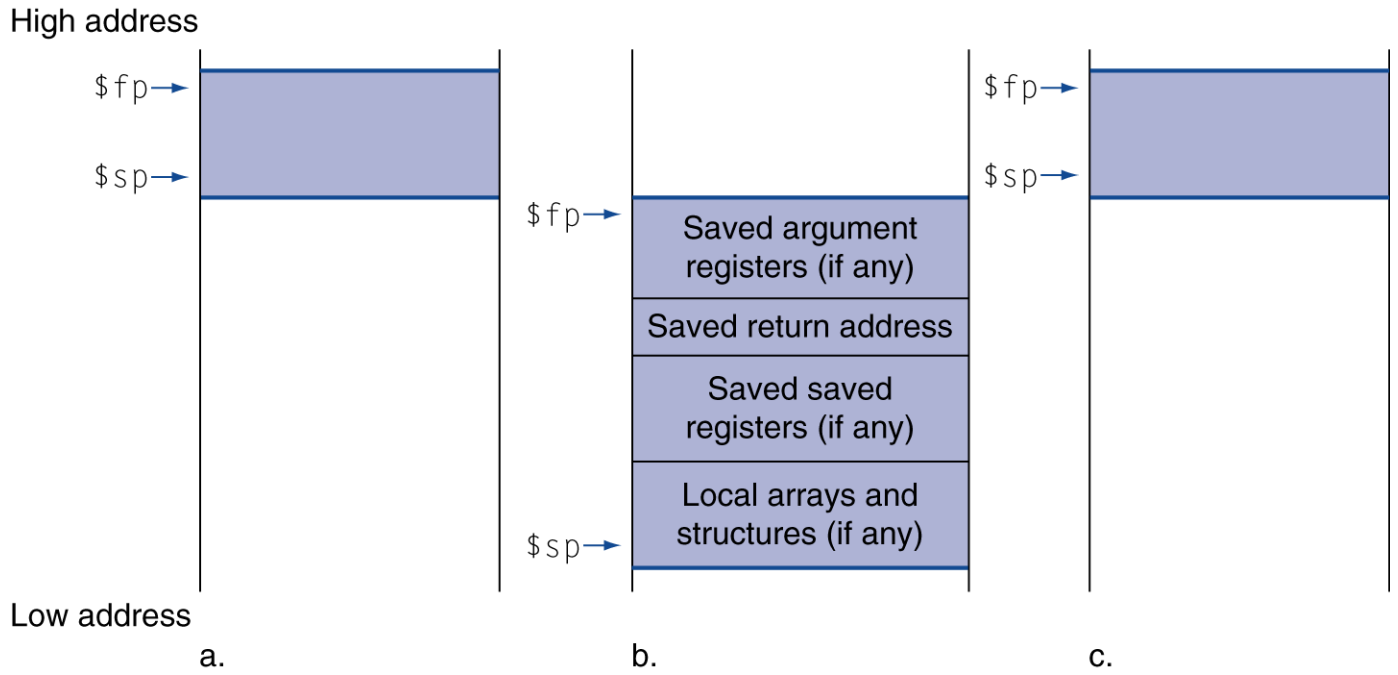
  - Endereço da próxima instrução salvo em $ra
  - Desvia para `ProcedureLabel`

- **Retorno do procedimento**: *jump register*

  `jr $ra`

  - Copia o conteúdo de $ra para o PC
  - Desvia o fluxo de volta ao *caller*

A pilha



- Dados locais do *callee*
  - e.g., variáveis automáticas do C
- Frame de procedimento (registro de ativação)
  - Espaço entre $fp e $sp
  - Usado por alguns compiladores para gerenciar o armazenamento de um procedimento

Usando a pilha

- Como usar a pilha num procedimento
  1. Alocar espaço na pilha
  2. Salvar os dados
  3. Restaurar os dados
  4. Desalocar espaço na pilha
- Como alocar espaço na pilha
  - Subtrair 4 do registrador $sp para cada palavra a ser armazenada
  - Ex: `addi $sp, $sp, -8` aloca espaço para duas palavras na pilha

Usando a pilha

- Como salvar dados na pilha
  - Salvar palavra por palavra em cada posição alocada
  - Deslocar `4 x posição` no `$sp`
- Exemplo
  - `addi $sp, $sp, -8 # aloca espaço para 2 pal.`
  - `sw $s0, 8($sp)  # salva $s0 na posição 2`
  - `sw $s1, 4($sp)  # salva $s1 na posição 1`

Exemplo de proc. folha

- Código C:

```
int exemplo_folha (int g, h, i, j) {
   int f;
   f = (g + h) - (i + j);
   return f;
}
```

  - Argumentos g, …, j em $a0, …, $a3
  - f em $s0 (importante: salvar $s0 na pilha)
  - Resultado em $v0

Exemplo de proc. folha

- Código MIPS:

```
exemplo_folha:
   addi $sp, $sp, -4
   sw   $s0, 0($sp)
   add  $t0, $a0, $a1
   add  $t1, $a2, $a3
   sub  $s0, $t0, $t1
   add  $v0, $s0, $zero
   lw   $s0, 0($sp)
   addi $sp, $sp, 4
   jr   $ra
```

Aloca 1 posição na pilha

Salva $s0 na pilha

Operações do procedimento

Resultado

Restaura o valor de $s0

Desaloca o espaço na pilha

Retorna ao *caller*

Procedimentos não-folha

- Procedimentos que chamam outros
- Antes de chamar outro procedimento, é necessário salvar
  - O endereço de retorno ($ra)
  - Quaisquer argumentos e temporários necessários após a chamada
- Restaura dados da pilha depois da chamada

Exemplo de proc. não-folha

- Código C:

```
int fat (int n) {
  if (n < 1) return 1;
  else return n * fat(n - 1);
}
```

  - Argumento n em $a0
  - Resultado no $v0

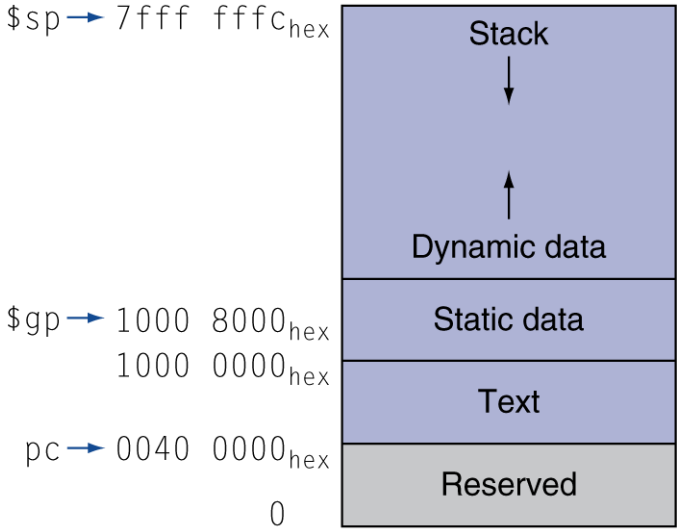
Exemplo de proc. não-folha

- Código MIPS:

```
fat:
    addi $sp, $sp, -8       # adjust stack for 2 items
    sw   $ra, 4($sp)        # save return address
    sw   $a0, 0($sp)        # save argument
    slti $t0, $a0, 1        # test for n < 1
    beq  $t0, $zero, L1
    addi $v0, $zero, 1      # if so, result is 1
    addi $sp, $sp, 8        #   pop 2 items from stack
    jr   $ra                #   and return
L1: addi $a0, $a0, -1       # else decrement n
    jal  fact               # recursive call
    lw   $a0, 0($sp)        # restore original n
    lw   $ra, 4($sp)        #   and return address
    addi $sp, $sp, 8        # pop 2 items from stack
    mul  $v0, $a0, $v0      # multiply to get result
    jr   $ra                # and return
```

Padrão da memória

- Text: código de programas
- Static data: variáveis globais
  - e.g., variáveis estáticas em C, arrays de tamanho fixo e strings
  - $gp é inicializado “no meio” para permitir offsets positivos e negativos
- Dados dinâmicos: heap
  - E.g., malloc em C, new no Java
- Stack: pilha



Caracteres

- Conjunto de caracteres byte-encoded
  - ASCII: 128 caracteres
    - 95 gráficos, 33 de controle
  - Latin-1: 256 characters
    - ASCII, +96 caracteres gráficos
- Unicode: conjunto de caracteres de 32-bit
  - Usado no Java, C++, …
  - Representa a maioria dos alfabetos do mundo, mais os símbolos
  - UTF-8, UTF-16: codificações de tamanho variável

Instruções Byte/Halfword

- Operações bit-a-bit
- MIPS byte/halfword load/store
  - Mais usado no processamento de strings
- Instruções

```
lb rt, offset(rs)      lh rt, offset(rs)
```

  - Extensão de sinal para 32 bits em rt

```
lbu rt, offset(rs)     lhu rt, offset(rs)
```

  - Unsigned int: complete com zero os 32 bits em rt

```
sb rt, offset(rs)      sh rt, offset(rs)
```

  - Armezena o byte/halfword mais à direita no reg.

Exemplo: função `strcpy`

- Código em C (simplificado):
  - String terminada com ‘\0’.

```
void strcpy (char x[], char y[])
{ int i;
  i = 0;
  while ((x[i]=y[i])!='\0')
    i += 1;
}
```

  - Endereços de x, y em $a0, $a1
  - i em $s0

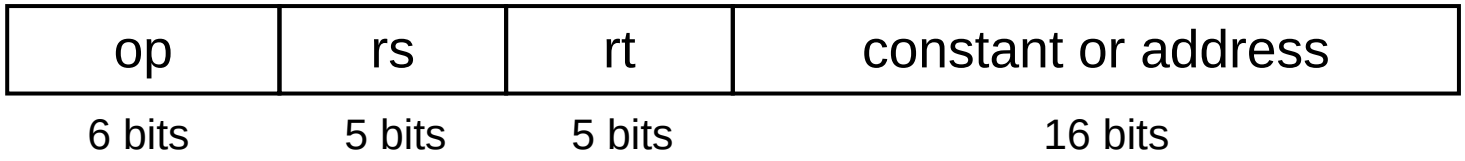
Exemplo: função strcpy

- MIPS code:

```
strcpy:
    addi $sp, $sp, -4       # adjust stack for 1 item
    sw   $s0, 0($sp)        # save $s0
    add  $s0, $zero, $zero  # i = 0
L1: add  $t1, $s0, $a1      # addr of y[i] in $t1
    lbu  $t2, 0($t1)        # $t2 = y[i]
    add  $t3, $s0, $a0      # addr of x[i] in $t3
    sb   $t2, 0($t3)        # x[i] = y[i]
    beq  $t2, $zero, L2     # exit loop if y[i] == 0
    addi $s0, $s0, 1        # i = i + 1
    j    L1                 # next iteration of loop
L2: lw   $s0, 0($sp)        # restore saved $s0
    addi $sp, $sp, 4        # pop 1 item from stack
    jr   $ra                # and return
```

Constantes de 32 bits

- A maioria das constantes são pequenas
  - Os 16 bits de um imediato costuma ser suficiente



- Para carregar uma constante de 32 bits

```
lui rt, constant
```

  - Copia a constante de 16 bits para os bits à esquerda do registrador rt
  - Define os bits à direita como zero

Constantes de 32 bits

- Exemplo: como carregar o valor 4.000.000 num registrador?
  - Em binário:

0000 0000 0011 1101 0000 1001 0000 0000

1. Carrega 61_{10} = 0000 0000 0011 1101_2 à esquerda:

```
lui $s0, 61
```

0000 0000 0011 1101	0000 0000 0000 0000

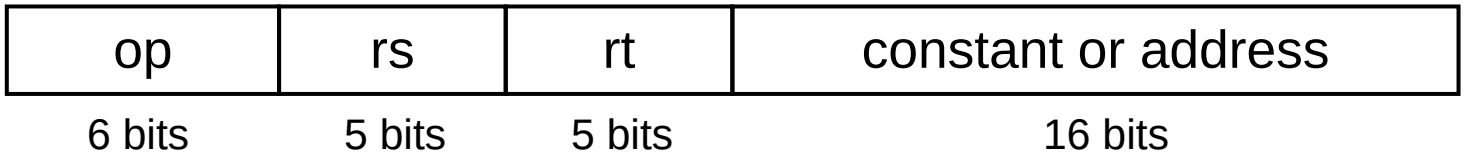
2. Carrega 2304_{10} = 0000 1001 0000 0000_2 à esquerda usando a instrução `ori`

```
ori $s0, $s0, 2304
```

0000 0000 0011 1101	0000 1001 0000 0000

Endereçamento de desvio

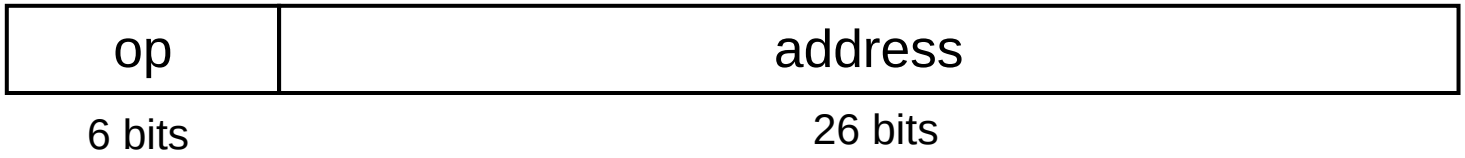
- Nas instruções de desvio especifica-se
  - opcode, dois registradores e destino do desvio
- A maioria dos desvios estão próximos às instruções
  - Para frente ou para trás



  - Endereçamento relativo ao PC
    - destino = PC + offset × 4
  - Capacidade de 4 x (-2^{15} a $2^{15} - 1$)

Endereçamento no jump

- Os destinos das instruções Jump (`j` and `jal`) podem estar em qualquer lugar no código
  - O endereço é representado no formato tipo J:



- Endereçamento pseudodireto
  - destino = $PC_{31...28}$: (address × 4)
- Capacidade de: 0 a 2^{31} - 1

Exemplo de endereçamento

- Exemplo:
  - Suponha que o Loop está no endereço 80000

```
Loop: sll  $t1, $s3, 2      80000
      add  $t1, $t1, $s6    80004
      lw   $t0, 0($t1)      80008
      bne  $t0, $s5, Exit   80012
      addi $s3, $s3, 1      80016
      j    Loop             80020
Exit: …                     80024
```

0	0	19	9	4	0
0	9	22	9	0	32
35	9	8	0		
5	8	21	2		
8	19	19	1		
2	20000				

Um exemplo completo

- Bubble sort implementado em assembly
- Procedimento Swap (troca dois elementos de um vetor - *folha*)

```
void swap(int v[], int k)
{
  int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

  - v em $a0, k em $a1, temp em $t0

A função swap

```
swap: sll $t1, $a1, 2    # $t1 = k * 4
      add $t1, $a0, $t1  # $t1 = v+(k*4)
                         #   (address of v[k])
      lw $t0, 0($t1)     # $t0 (temp) = v[k]
      lw $t2, 4($t1)     # $t2 = v[k+1]
      sw $t2, 0($t1)     # v[k] = $t2 (v[k+1])
      sw $t0, 4($t1)     # v[k+1] = $t0 (temp)
      jr $ra             # return to calling routine
```

Bubblesort em C

- Não é folha (faz chamada a swap)

```
void sort (int v[], int n)
{
  int i, j;
  for (i = 0; i < n; i += 1) {
    for (j = i - 1;
         j >= 0 && v[j] > v[j + 1];
         j -= 1) {
      swap(v,j);
    }
  }
}
```

  - v em $a0, k em $a1, i em $s0, j em $s1

A função em assembly MIPS

```
          move $s2, $a0            # save $a0 into $s2
          move $s3, $a1            # save $a1 into $s3
          move $s0, $zero          # i = 0
for1tst:  slt  $t0, $s0, $s3       # $t0 = 0 if $s0 ≥ $s3 (i ≥ n)
          beq  $t0, $zero, exit1   # go to exit1 if $s0 ≥ $s3 (i ≥ n)
          addi $s1, $s0, -1        # j = i - 1
for2tst:  slti $t0, $s1, 0         # $t0 = 1 if $s1 < 0 (j < 0)
          bne  $t0, $zero, exit2   # go to exit2 if $s1 < 0 (j < 0)
          sll  $t1, $s1, 2         # $t1 = j * 4
          add  $t2, $s2, $t1       # $t2 = v + (j * 4)
          lw   $t3, 0($t2)         # $t3 = v[j]
          lw   $t4, 4($t2)         # $t4 = v[j + 1]
          slt  $t0, $t4, $t3       # $t0 = 0 if $t4 ≥ $t3
          beq  $t0, $zero, exit2   # go to exit2 if $t4 ≥ $t3
          move $a0, $s2            # 1st param of swap is v (old $a0)
          move $a1, $s1            # 2nd param of swap is j
          jal  swap                # call swap procedure
          addi $s1, $s1, -1        # j -= 1
          j    for2tst             # jump to test of inner loop
exit2:    addi $s0, $s0, 1         # i += 1
          j    for1tst             # jump to test of outer loop
```

Move params

Outer loop

Inner loop

Pass params & call

Inner loop

Outer loop

O procedimento completo

```
sort:   addi $sp,$sp, -20       # make room on stack for 5 registers
        sw $ra, 16($sp)         # save $ra on stack
        sw $s3,12($sp)          # save $s3 on stack
        sw $s2, 8($sp)          # save $s2 on stack
        sw $s1, 4($sp)          # save $s1 on stack
        sw $s0, 0($sp)          # save $s0 on stack
        …                       # procedure body
        …
        exit1: lw $s0, 0($sp)   # restore $s0 from stack
        lw $s1, 4($sp)          # restore $s1 from stack
        lw $s2, 8($sp)          # restore $s2 from stack
        lw $s3,12($sp)          # restore $s3 from stack
        lw $ra,16($sp)          # restore $ra from stack
        addi $sp,$sp, 20        # restore stack pointer
        jr $ra                  # return to calling routine
```