

## Aula 2: Instruções - A linguagem de máquina - Parte II

Professor(a): Virgínia Fernandes Mota

<http://www.dcc.ufmg.br/~virginiaferm>

OCS (TEORIA) - SETOR DE INFORMÁTICA



- Suporte a Procedimentos no Hardware do Computador
- Comunicando-se com as Pessoas
- Endereçamento no MIPS para Operandos Imediatos e Endereços de 32 bits

- Procedimento (ou função)
  - Sub-rotina armazenada que realiza uma tarefa com base nos parâmetros com os quais ela é provida
  - Usada para estruturar programas
  - Permite que código seja reutilizado
- Seis etapas precisam ser seguidas
  - Colocar parâmetros onde procedimento possa acessá-los
  - Transferir controle para o procedimento
  - Adquirir recursos necessários para o procedimento
  - Realizar tarefa desejada
  - Colocar valor de retorno onde código que o chamou possa acessá-lo
  - Retornar controle para o ponto de origem

- Registradores usados para a chamada de procedimento:
  - \$a0 - \$a3 : usados para passar parâmetros
  - \$v0 - \$v1 : usados para valores de retorno
  - \$ra : registrador de endereço de retorno, usado para retornar ao ponto de origem
- Instrução apenas para os procedimentos:
  - **jal** EndereçoProcedimento # jal (jump and link)
  - Desvia para endereço e simultaneamente salva endereço da instrução seguinte no registrador \$ra
  - Instrução seguinte dada por  $PC + 4$ 
    - PC (contador de programa): registrador que contém o endereço da instrução que está sendo executada

- Instrução **jump register (jr)** permite fazer desvio incondicional para endereço especificado em um registrador jr \$ra
- Código que chama procedimento (caller)
  - Coloca valores dos parâmetros em \$a0-\$a3
  - Utiliza jal X para desviar para o procedimento X
- Código invocado (callee)
  - Executa instruções
  - Coloca resultados em \$v0 - \$v1
  - Retorna controle para caller usando jr \$ra

- Durante execução, procedimento pode precisar usar registradores
- Mas e se os registradores estavam sendo usados pelo caller?
  - Depois de completar a chamada, caller espera que os valores originais dos seus registradores sejam mantidos
- Se registradores usados durante execução do procedimento, valores originais devem ser salvos e depois restaurados

- Conceito de spilling registers utilizado
  - Variáveis menos utilizadas armazenadas na memória
- Pilha usada para implementar spilling registers
  - Organizada como fila do tipo "último a entrar, primeiro a sair"
  - Ponteiro para endereço mais recentemente alocado na pilha
    - Chamado de stack pointer (\$sp)
    - Indica onde valores antigos estão localizados e/ou
    - Mostra onde próximo procedimento deverá alocar os spilling registers

# Suporte a Procedimentos no Hardware do Computador

- Push coloca dados na pilha
- Pop retira dados
- Convenção histórica: pilhas "crescem" para baixo
  - Endereços maiores para menores
  - Uma word para cada registrador salvo ou restaurado
  - Valores levados para pilha subtraindo do valor do stack pointer
  - Valores retirados da pilha somando ao valor do stack pointer



- Exemplo: Qual o código assembly do MIPS compilado para a seguinte função?

```
1  int exemplo (int g, int h, int i, int j) {  
2      int f;  
3      f = (g+h) - (i+j);  
4      return f;  
5  }
```

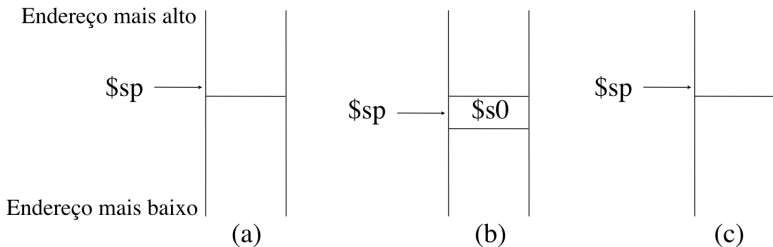
- Primeiro passo:
  - Variáveis g, h, i e j correspondem aos registradores \$a0, \$a1, \$a2 e \$a3
  - Variável f corresponde a \$s0
  - Procedimento começa com rótulo (label) com o nome do procedimento

- Próximo passo: Salvar registradores usados pelo procedimento
  - Valores antigos empilhados
  - Importante: registradores temporários não precisam ser preservados em uma chamada de procedimento
  - Só precisamos salvar registrador \$s0
  - Precisamos primeiro criar espaço na pilha
  - Subtraímos de \$sp o valor que precisamos

- Subtração:  
addi \$sp, \$sp, -4 # ajustamos a pilha  
sw \$s0, 0 (\$sp) # salva registrador para uso posterior
- Corpo do procedimento  
add \$t0, \$a0, \$a1 #  $t0 = g + h$   
add \$t1, \$a2, \$a3 #  $t1 = i + j$   
sub \$s0, \$t0, \$t1 #  $f = t0 - t1$   
add \$v0, \$s0, \$zero # retorna  $f$  ( $v0 = s0 + 0$ )?
- Restauramos valores dos registradores antes do retorno  
lw \$s0, 0 (\$sp) # restaura registrador utilizado  
addi \$sp, \$sp, 4 # exclui um item da pilha

# Suporte a Procedimentos no Hardware do Computador

- Procedimento termina com salto para endereço de retorno  
jr \$ra # retorna para código que chamou procedimento



Valores da pilha (a) antes, (b) durante e (c) depois da chamada de procedimento

- Procedimentos aninhados.
- Procedimentos que não chamam outros procedimentos são chamados folha
- Procedimentos podem chamar outros procedimentos?
  - O que ocorreria se seguinte seqüência ocorresse:
    - Programa principal chama procedimento A, passando 3 como argumento
    - Procedimento A chama procedimento B, passando 7 como argumento

- Poderíamos perder valores originais de \$a0 e de \$ra.
- **Solução:** empilhar todos os registradores que precisam ser preservados
- Neste caso, incluindo qualquer registrador temporário que precise ser utilizado posteriormente
  - Caller empilha registradores de argumento (\$a0-\$a3) e temporários (\$t0-\$t9) que sejam necessários após chamada
  - Callee empilha \$ra e quaisquer registradores \$s0-\$s7 usados por ele
- Stack pointer atualizado para refletir total de registradores salvos na pilha
- No retorno, registradores restaurados e \$sp reajustado

- Exemplo: Qual o código assembly para a função recursiva abaixo?

```
1 int fact (int n) {  
2     if (n < 1) return 1;  
3     else return (n * fact (n-1));  
4 }
```

- Parâmetro n corresponde ao registrador \$a0
- Função começa com rótulo, e depois salva registradores na pilha



- Código:  
fact:  
    addi \$sp, \$sp, -8 # ajusta pilha para dois itens  
    sw \$ra, 4(\$sp) # salva endereço de retorno  
    sw \$a0, 0(\$sp) # salva argumento n
- Primeira vez que fact chamado: salva endereço do código que o chamou
- Instruções seguintes testam se n é menor que 1  
    slti \$t0, \$a0, 1 #teste se  $n < 1$   
    beq \$t0, \$zero, L1 # se  $n \geq 1$ , vai para L1

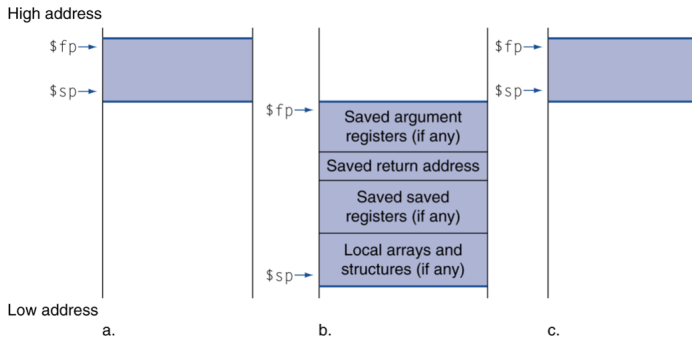
- Se  $n < 1$ , fact retorna 1 e restaura valores da pilha  
addi \$v0, \$zero, 1 # retorna 1  
addi \$sp, \$sp, 8 # retira 2 itens da pilha  
jr \$ra # retorna para depois de jal
- Porque não restauramos \$a0 e \$ra? → Porque seus valores não mudaram!
- Se n não é maior que 1, argumento decrementado, e fact chamado com valor decrementado  
L1: addi \$a0, \$a0, -1 #  $n \geq 1$  : argumento recebe (n-1)  
jal fact # chama fact com (n-1)

- Próxima instrução é onde fact retorna: Endereço de retorno antigo, argumento antigo e stack pointer restaurados  
lw \$a0, 0 (\$sp) # retorna de jal, restaura argumento n  
lw \$ra, 4 (\$sp) # restaura endereço de retorno  
addi \$sp, \$sp, 8 # ajusta stack pointer para retirar 2 itens
- Operação final antes do retorno: cálculo do produto do argumento antigo pelo valor retornado  
mul \$v0, \$a0, \$v0 # retorna  $n * \text{fact}(n - 1)$ ?
- Salto para endereço de retorno original  
jr \$ra # retorna para que chamou

# Suporte a Procedimentos no Hardware do Computador

- Alocando espaço para novos dados na pilha
- Pilha também usada para armazenar variáveis locais ao procedimento que não cabem nos registradores
- Segmento da pilha que contém registradores salvos e variáveis locais de um procedimento chamado de registro de ativação (ou frame de procedimento)
  - \$fp aponta para primeira palavra do registro de ativação
    - Como \$sp pode mudar ao longo da execução do código, referência a variáveis locais poderia ter offsets diferentes, dependendo de onde estiverem no código
    - Evitaremos seu uso

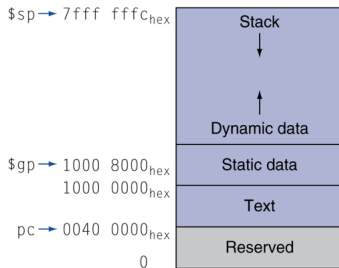
# Suporte a Procedimentos no Hardware do Computador



- Alocando espaço para novos dados na heap
- Programas C possuem dois tipos de variáveis
  - Automáticas: Locais a um procedimento e descartadas quando este termina. Escopo local.
  - Estáticas: Variáveis declaradas fora de um procedimento ou com uso da palavra reservada static. Escopo global.
- Registrador \$gp simplifica acesso aos dados estáticos

# Suporte a Procedimentos no Hardware do Computador

- Variáveis estáticas e estruturas de dados dinâmicas alocadas segundo uma convenção.



- Pilha começa na parte alta da memória e "cresce" para baixo, como já visto
- Primeira parte da extremidade baixa da memória é reservada
- Código de máquina do MIPS na sequência
  - Denominado segmento de texto
- Segmento seguinte chamado de segmento de dados estáticos
  - Local para constantes e outras variáveis estáticas
- Heap na sequência
  - Armazena dados dinâmicos (malloc do C e new do Java)



- E se houver mais de quatro parâmetros?
  - Quatro primeiros parâmetros alocados nos registradores
  - Parâmetros extras alocados na pilha

## Vamos rodar o exemplo do fatorial no simulador MIPS32!

Faça uma versão iterativa e uma versão recursiva do fatorial em assembly e rode no simulador. Mostre tanto o código em C quando o código em assembly.

<http://rivoire.cs.sonoma.edu/cs351/wemips/>

Entrega até 12/03.

## E nosso problema do while?

<code>while ( &lt;cond&gt; ) {</code>	<code>L1: if ( &lt;cond&gt; ) {</code>
<code>    &lt;while-body&gt;</code>	<code>    &lt;while-body&gt;</code>
<code>}</code>	<code>    goto L1 ;</code>
	<code>}</code>

If condition is true, execute body and go back, otherwise do next statement.

<code>while ( i &lt; j ) {</code>	<code>L1: if ( i &lt; j ) {</code>
<code>    k++ ;</code>	<code>    k++ ;</code>
<code>    i = i * 2 ;</code>	<code>    i = i * 2 ;</code>
<code>}</code>	<code>    goto L1 ;</code>
	<code>}</code>

<code>L1: bge \$s1, \$s2, DONE</code>	<code># branch if ! ( i &lt; j )</code>
<code>addi \$s3, \$s3, 1</code>	<code># k++</code>
<code>add \$s1, \$s1, \$s1</code>	<code># i = i * 2</code>
<code>j L1</code>	<code># jump back to top of loop</code>
<code>DONE:</code>	

- Maioria dos computadores usam 1 byte para representar caracteres: ASCII é a representação mais utilizada (American Standard Code for Information Interchange).
- Palavra no MIPS tem 4 bytes.
- Uso de texto freqüente: MIPS oferece instruções para mover bytes
  - lb lê byte da memória, armazenando-o nos 8 bits mais à direita do registrador. Ex: lb \$t0, 0 (\$sp).
  - sb armazena byte mais à direita do registrador na memória. Ex: sb \$t0, 0 (\$gp).

- Caracteres normalmente combinados na forma de strings → Cadeias de caracteres de tamanho variável
- Como representá-la? Três opções:
  - Primeira posição da string reservada para indicar seu tamanho
  - Como um estrutura, com um campo para indicar seu tamanho
  - Última posição da string ocupada por caracter especial, que marca seu final
  - C utiliza esta opção, com byte null (0) indicando fim de string

- Exemplo: Código abaixo copia string y para x. Qual o código assembly equivalente no MIPS?

```
1 void strcpy (char x[], char y[]) {  
2     int i = 0;  
3     while ((x[i] = y[i]) != '\0')?  
4         i+=1;  
5 }
```

- Considere que i será armazenado em \$s0

- O primeiro passo é salvar o valor original do registrador \$s0  
strcpy:  
    addi \$sp, \$sp, -4 # ajusta pilha para mais um item  
    sw \$s0, 0 (\$sp) # salva \$s0
- Em seguida iniciamos i com 0  
    add \$s0, \$zero, \$zero # i = 0
- Início do loop. Endereço de y[i] formado inicialmente pela soma de i com base do vetor  
    L1: add \$t1, \$s0, \$a1 # endereço de y[i] em \$t1
- Porque não multiplicamos i por 4, como nos outros exemplos?

- Pois y é um array de caracteres, portanto um array de bytes, e não de words!
- Carregando o caracter de y[i]  
`lb $t2, 0 ($t1) # $t2 = y[i]`
- Atribuição do caracter a x[i]  
`add $t3, $s0, $a0 # endereço de x[i] em $t3`  
`sb $t2, 0 ($t3) # x[i] = y[i]`
- Atingimos último caracter da string?  
`beq $t2, $zero, L2 # se y[i] == 0, vai para L2`



- Se não for último caracter, incrementamos i e voltamos para o início do loop  
addi \$s0, \$s0, 1 #  $i = i + 1$   
j L1 # volta para o início do loop
- Se foi o último caracter, restauramos valores originais de \$sp e \$s0 e retornamos  
L2: lw \$s0, 0(\$sp) # restauramos \$s0  
addi \$sp, \$sp, 4  
jr \$ra

# Comunicando-se com as Pessoas

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

- Repare que uso de \$s0 não é necessário
  - Procedimento folha
  - Porque não usar registradores temporários?
  - Evita necessidade de salvar \$s0
  - Registradores \$s0-\$s7 usados apenas quando registradores temporários esgotados
- Algumas codificações, como Unicode, usam 16 bits para representar caracteres
  - MIPS possui instruções para carregar (lh) e armazenar (sh) meia palavra (16 bits)

# Endereçamento no MIPS para Operandos Imediatos e Endereços de 32 Bits

- Em algumas ocasiões conveniente ter endereços ou constantes de 32 bits
- Mas formato das instruções é fixo...

R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	Endereço / constante de 16 bits		
J	op	endereço de 26 bits				

# Endereçamento no MIPS para Operandos Imediatos e Endereços de 32 Bits

- MIPS inclui instruções para permitir operandos imediatos de 32 bits
  - Constantes grandes desmembradas em partes
  - Partes remontadas em um registrador
  - Montador usa registrador temporário \$at para este fim

# Endereçamento no MIPS para Operandos Imediatos e Endereços de 32 Bits

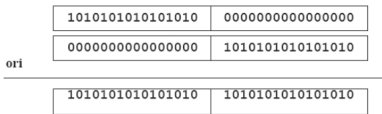
- Instruções **lui** (load upper immediate) e **ori** (or immediate) usadas para este fim

lui \$t0, 1010101010101010



preenchido com zeros

ori \$t0, \$t0, 1010101010101010



# Endereçamento no MIPS para Operandos Imediatos e Endereços de 32 Bits

- No caso do endereçamento do desvio, temos dois casos:
  - Desvios condicionais são instruções tipo I: 16 bits para endereço de desvio
  - Instruções de salto são do tipo J: 26 bits para endereço de desvio
- Logo limitam, no pior caso, endereços a 16 bits: Nenhum programa poderia ser maior do que  $2^{16}$

# Endereçamento no MIPS para Operandos Imediatos e Endereços de 32 Bits

- **Solução:** Contador de programa = registrador + endereço de desvio
- Nova questão: que registrador usar?
- Estatísticas mostram que maior parte dos desvios vão para locais a menos de 16 instruções a partir da posição atual de PC
  - Endereçamento relativo ao PC → Na realidade, é relativo ao endereço da instrução seguinte ao PC
  - Usado para todos os desvios condicionais



# Endereçamento no MIPS para Operandos Imediatos e Endereços de 32 Bits

- Instruções de jump and link chamam procedimentos
  - Não têm motivo para estarem próximos à chamada
  - Utilizam outra forma de endereçamento
- Ambas as instruções de desvio fazem saltos relativos à words, e não bytes
  - Permite salto para distâncias 4 vezes maiores

# Endereçamento no MIPS para Operandos Imediatos e Endereços de 32 Bits

- Modos de endereçamento do MIPS:
  - Endereçamento em registrador: operando é um registrador
  - Endereçamento de base ou deslocamento: operando está no local de memória cujo endereço é a soma de um registrador e uma constante na instrução
  - Endereçamento imediato: onde o operando é uma constante dentro da própria instrução
  - Endereçamento relativo ao PC: onde o endereçamento é a soma do PC e uma constante na instrução
  - Endereçamento pseudo-direto: onde o endereço de jump são os 26 bits da instrução concatenados com os bits mais altos do PC

# Endereçamento no MIPS para Operandos Imediatos e Endereços de 32 Bits

## 1. Endereçamento imediato

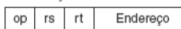


## 2. Endereçamento em registrador

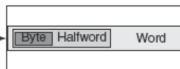


Registradores  
Registrador

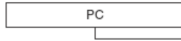
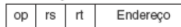
## 3. Endereçamento de base



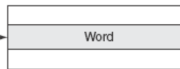
Memória



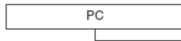
## 4. Endereçamento relativo ao PC



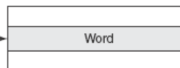
Memória



## 5. Endereçamento pseudodireto



Memória



# Resumo

Nome	Número do registrador	Uso
\$zero	0	O valor constante 0
\$v0-\$v1	02-03	Valores para resultados e avaliação de expressões
\$a0-\$a3	04-07	Argumentos
\$t0-\$t7	08-15	Temporários
\$s0-\$s7	16-23	Valores salvos
\$t8-\$t9	24-25	Mais temporários
\$gp	28	Ponteiro global
\$sp	29	Ponteiro de pilha
\$fp	30	Ponteiro de quadro
\$ra	31	Endereço de retorno

Registrador 1 (\$at) reservado para o assembler, 26-27 para o sistema operacional

Assembly do MIPS				
Categoria	Instrução	Exemplo	Significado	Comentários
Aritmética	add	add \$s1,\$s2,\$s3	$\$s1 = \$s2 + \$s3$	Três operandos; dados nos registradores
	subtract	sub \$s1,\$s2,\$s3	$\$s1 = \$s2 - \$s3$	Três operandos; dados nos registradores
	add immediate	addi \$s1,\$s2,100	$\$s1 = \$s2 + 100$	Usada para somar constantes
Transferência de dados	load word	lw \$s1,100(\$s2)	$\$s1 = \text{Memória}[\$s2 + 100]$	Dados da memória para o registrador
	store word	sw \$s1,100(\$s2)	$\text{Memória}[\$s2 + 100] = \$s1$	Dados do registrador para a memória
	load byte	lb \$s1,100(\$s2)	$\$s1 = \text{Memória}[\$s2 + 100]$	Byte da memória para registrador
	store byte	sb \$s1,100(\$s2)	$\text{Memória}[\$s2 + 100] = \$s1$	Byte de um registrador para memória
	load upper immed.	lui \$s1,100	$\$s1 = 100 * 2^{16}$	Carrega constante nos 16 bits mais altos
Desvio condicional	branch on equal	beq \$s1,\$s2,25	if ( $\$s1 == \$s2$ ) go to PC + 4 + 100	Testa igualdade; desvio relativo ao PC
	branch on not equal	bne \$s1,\$s2,25	if ( $\$s1 \neq \$s2$ ) go to PC + 4 + 100	Testa desigualdade; relativo ao PC
	set on less than	slt \$s1,\$s2,\$s3	if ( $\$s2 < \$s3$ ) $\$s1 = 1$ ; else $\$s1 = 0$	Compara menor que; usado com beq, bne
	set less than immediate	sli \$s1,\$s2,100	if ( $\$s2 < 100$ ) $\$s1 = 1$ ; else $\$s1 = 0$	Compara menor que constante
Desvio incondicional	jump	j 2500	go to 10000	Desvia para endereço de destino
	jump register	jr \$ra	go to \$ra	Para switch e retorno de procedimento
	jump and link	jal 2500	$\$ra = PC + 4$ . go to 10000	Para chamada de procedimento

## Instruções - A linguagem de máquina - Parte III