Arquitetura e Organização de Computadores

Cap 2. A Linguagem dos Computadores

AP 2 - Instruções para tomada de decisão

Prof. Dr. João Fabrício Filho

Universidade Tecnológica Federal do Paraná
Campus Campo Mourão
2024

Qual a diferença entre um computador e uma calculadora?

Que tipo de funcionalidades de alto nível ainda não vimos em MIPS?

O que é uma decisão?

Operações de desvio

- Desvio para uma instrução se a condição for verdadeira
  - Caso contrário, continue sequencialmente

```
beq rs, rt, L1 #branch if equal
```

- se rs == rt desvia para a instrução marcada como L1

```
bne rs, rt, L1 #branch if not equal
```

- se rs != rt desvia para a instrução marcada como L1

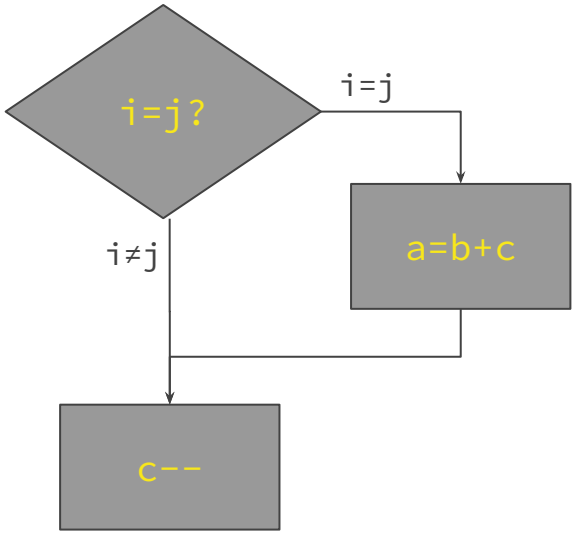
```
j L1 #jump to L1
```

- salta incondicionalmente para instrução marcada L1

Compilando if

Código em C

```
if (i==j)
    a = b+c
c--;
...
```

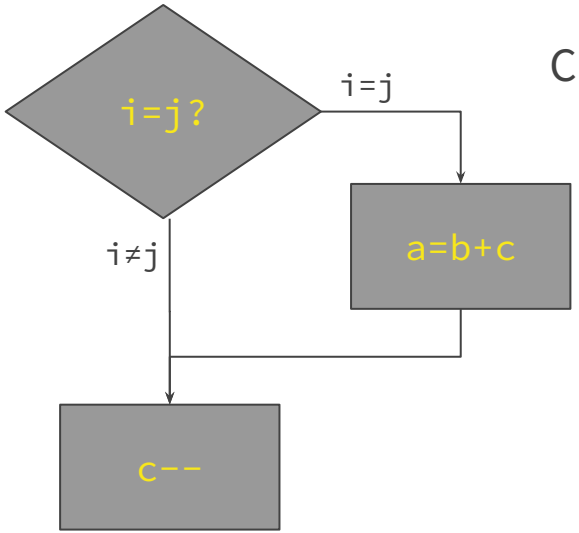


considere a, b, c em $s0, $s1, $s2 e i, j em $t0, $t1

Compilando if

Código em C

```
if (i==j)
    a = b+c
c--;
...
```



Código MIPS

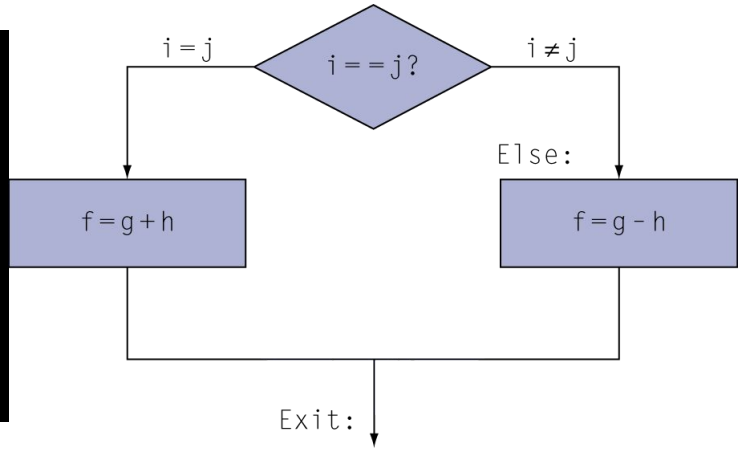
```
bne $t0, $t1, L1
add $s0, $s1, $s2
L1: addi $s2, $s2, -1
    ...
```

considere a, b, c em $s0, $s1, $s2 e i, j em $t0, $t1

Compilando if-else

Código em C

```
if (i==j)
    f = g+h
else
    f = g-h
```

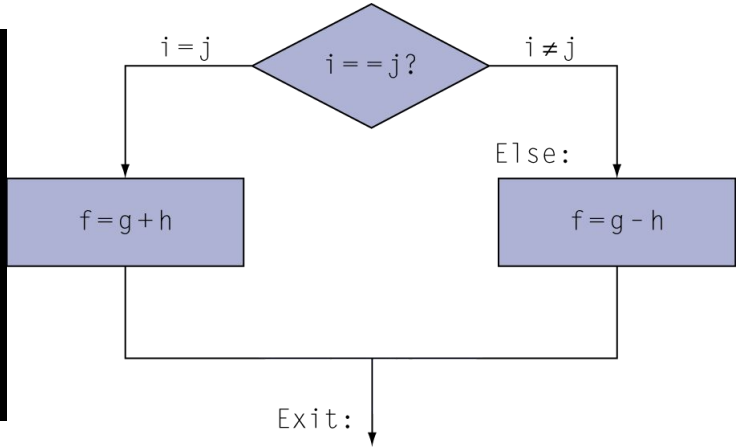


considere f, g, h, i, j em $s0, $s1, $s2, $s3, $s4

Compilando if-then-else

Código em C

```
if (i==j)
    f = g+h
else
    f = g-h
```



Código MIPS

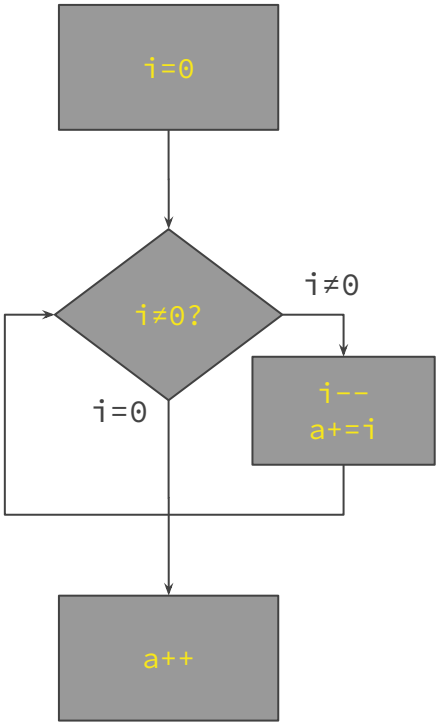
```
bne $s3, $s4, ELSE
add $s0, $s1, $s2
j EXIT
ELSE: sub $s0, $s1, $s2
EXIT:
    ...
```

considere f, g, h, i, j em $s0, $s1, $s2, $s3, $s4

Compilando loops

Código em C

```
i=0;
while (i!=0)
    i--
    a+=i
a++
```

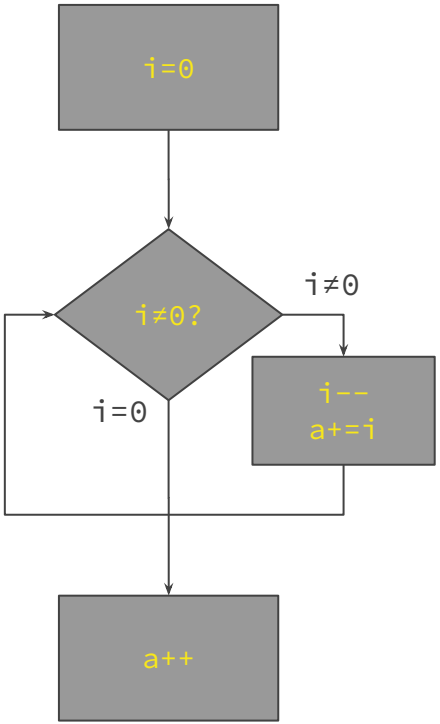


considere i em $t0 e a em $s1

Compilando loops

Código em C

```
i=0;
while (i!=0)
    i--
    a+=i
a++
```



Código MIPS

```
add $t0, $0, $0
LOOP: beq $t0, $0, EXIT
    addi $t0, $t0, -1
    add $s1, $s1, $t0
    j LOOP
EXIT: addi $s1, $s1, 1
```

considere i em $t0 e a em $s1

Mais operações condicionais

Definir o resultado como 1 se a condição for verdadeira

- Caso contrário, resetar (definir como zero)
- Se (rs < rt) então rd=1 senão rd=0

```
slt rd, rs, rt #set on less than
```

- Se (rs < constante) rd=1 senão rd=0

```
slti rd, rs, constante
```

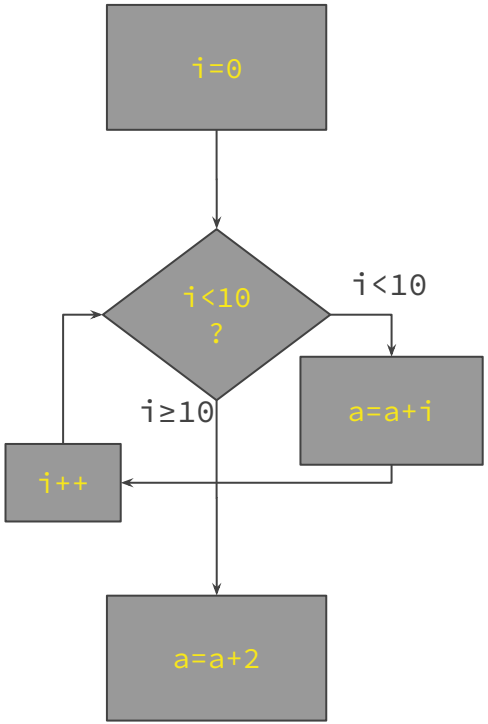
- Usado em combinação com beq, bne

```
slt $t0, $s1, $s2 #if $s1<$s2
bne $t0, $zero, L #desvia para L
```

Compilando loops

Código em C

```
for (i=0; i<10; i++)
    a += i
a = a + 2
```

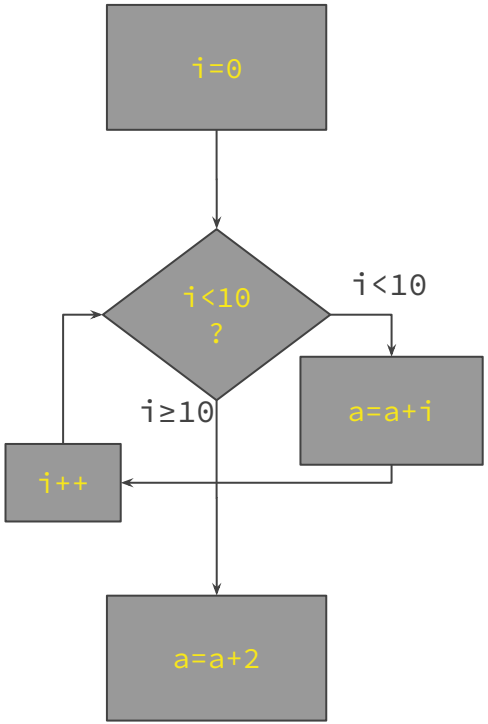


considere i em $t0 e a em $s1

Compilando loops

Código em C

```
for (i=0; i<10; i++)
    a += i
a = a + 2
```




```
add $t0, $0, $0
FOR: slti $t1, $t0, 10
    beq $t1, $0, EXIT
    add $s1, $s1, $t0
    addi $t0, $t0, 1
    j FOR
EXIT: addi $s1, $s1, 2
    …
```

considere i em $t0 e a em $s1

Compilando loops

Código em C

```
while (save[i]==k)
    i += 1
```

considere i em $s3, k em $s5 e o endereço de save em $s6

Compilando loops

Código em C

```
while (save[i]==k)
    i += 1
```

considere i em $s3, k em $s5 e o endereço de save em $s6

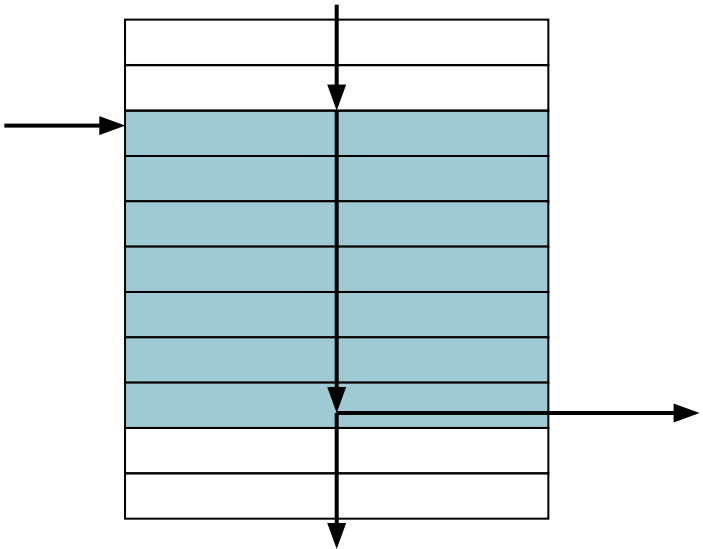
Código MIPS

```
LOOP:
    sll $t1, $s3, 2
    add $t1, $t1, $s6
    lw $t0, 0($t1)
    bne $t0, $s5, EXIT
    addi $s3, $s3, 1
    j LOOP
EXIT:
    …
```

Blocos básicos

Um bloco básico é uma sequência de instruções:

- Sem desvios (exceto no final)
- Nenhum alvo de desvio (exceto no início)



- Um compilador identifica blocos básicos para otimização
- Um processador avançado pode acelerar a execução de blocos básicos

Projeto de instruções de desvio

Por que não blt ou bge?

- blt: branch on less than
- bge: branch on greater or equal

Hardware para fazer comparação <, > é mais lento do que para = ou ≠

- Combinar com desvio envolve mais trabalho por instrução
- Exige um clock mais lento
- Todas as instruções seriam penalizadas

beq e bne são os casos mais comuns

- Compromisso de projeto!

Sinalizados vs não-sinalizados

- Comparação sinalizada: slt, slti
- Comparação não-sinalizada: sltu, sltui
- Exemplo
  - $s0 = 1111 1111 1111 1111 1111 1111 1111 1111
  - $s1 = 0000 0000 0000 0000 0000 0000 0000 0001

```
slt $t0, $s0, $s1 #signed
```

```
sltu $t0, $s0, $s1 #unsigned
```

Chamada de procedimentos

Etapas necessárias

1. Colocar parâmetros nos registradores ($a0-$a3)
2. Transferir o controle para o procedimento (jal)
3. Salvar registradores $s que usar na pilha
4. Realizar a tarefa desejada
5. Colocar o resultado nos registradores de retorno ($v0, $v1)
6. Restaurar os registradores $s e retornar ao local da chamada
    - jr $ra

Registradores utilizados

- **$a0 - $a3**: argumentos (reg's 4 - 7)
- **$v0, $v1**: valores de resultado (reg's 2 - 3)
- **$t0 - $t9**: temporários
  - Pode ser substituído pelo procedimento chamado
- **$s0 - $s7**: variáveis locais
  - Deve ser salvo/restaurado pelo procedimento chamado
- **$gp**: ponteiro global para dados estáticos (reg 28)
- **$sp**: ponteiro da pilha (reg 29)
- **$fp**: ponteiro do frame (reg 30)
- **$ra**: endereço de retorno (reg 31)

Instruções - Chamada de procedimento

- Chamada de procedimento: jump and link

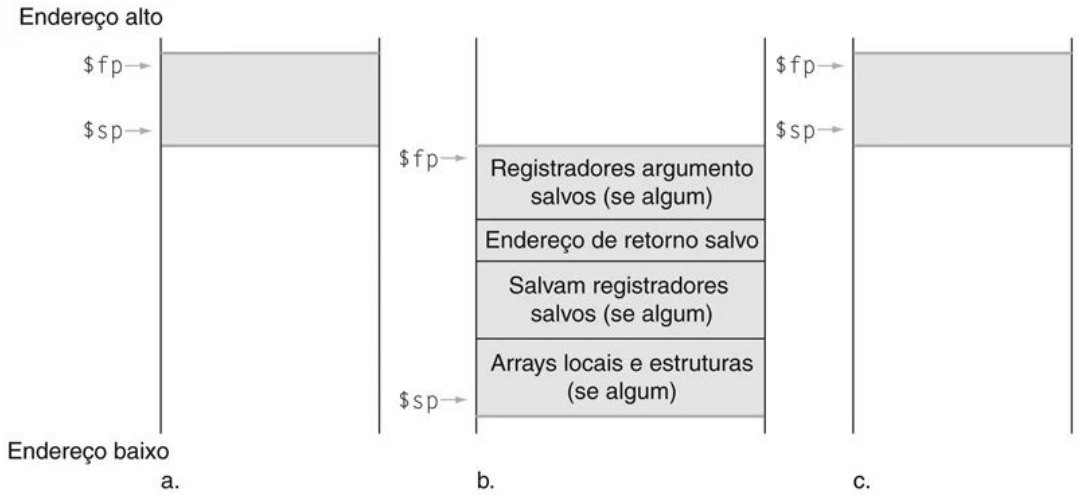
```
jal ENDERECO_PROCEDIMENTO
```

  - Salva o endereço da instrução seguinte em $ra
  - Salta para o endereço de destino
- Retorno do procedimento: jump register

```
jr $ra
```

- Copia $ra para pc (program counter)
- Também pode ser usado para saltos computados

Dados locais na pilha



- Variáveis locais alocadas pelo procedimento chamado
  - Como arrays ou estruturas locais
- Frame de procedimento (registro de ativação)
  - Alguns compiladores usam o frame pointer ($fp) para apontar para a primeira palavra do registro de ativação

Exemplo - procedimentos não aninhados

Código em C

```
int soma(int a,int b) {
    int s;
    s = a+b;
    return s;
}
```

- Argumentos a,b em $a0,$a1
- s em $s0
- Resultado em $v0

Exemplo - procedimentos não aninhados

Código em C

```c
int soma(int a,int b) {
    int s;
    s = a+b;
    return s;
}
```

- Argumentos a,b em $a0,$a1
- s em $s0
- Resultado em $v0

Código MIPS

```
soma:
    addi $sp, $sp, -4
    sw $s0, 0($sp)
    add $s0, $a0, $a1
    add $v0, $s0, $zero
    lw $s0, 0($sp)
    addi $sp, $sp, 4
    jr $ra
```

Exemplo - procedimentos não aninhados

Código em C

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

- Argumentos g, …, j em $a0, …, $a3
- f em $s0
- Resultado em $v0

Exemplo - procedimentos não aninhados

Código em C

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

- Argumentos g, …, j em $a0, …, $a3
- f em $s0
- Resultado em $v0

Código MIPS

```
exemplo:
    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
```

Exemplo - procedimentos não aninhados

Código em C

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

- Argumentos g, …, j em $a0, …, $a3
- f em $s0
- Resultado em $v0

Código MIPS

```
exemplo:
    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
```

salva $s0 na pilha

corpo do procedimento

resultado em $v0

restaura $s0

retorno

Procedimentos aninhados

- Procedimentos que chamam outros procedimentos
- Para chamadas aninhadas, o procedimento deve salvar na pilha:
  - Seu endereço de retorno
  - Quaisquer argumentos e temporários necessários após a chamada
- Restaurar a pilha após a chamada

Exemplo de proc. não-folha

Código em C

```
int mult (int n, int m) {
    int f=0;
    for (int i=0; i<m; i++)
      f = soma(f, n);
    return f;
}
```

- Argumentos n,m em $a0, $a1
- Resultado em $v0

Exemplo de proc. não-folha

Código em C

```c
int mult (int n, int m) {
    int f=0;
    for (int i=0; i<m; i++)
      f = soma(f, n);
    return f;
}
```

- Argumentos n,m em $a0, $a1
- Resultado em $v0

```
mult:
    addi $sp, $sp, -16
    sw $a0, 16($sp)
    sw $a1, 8($sp)
    sw $ra, 4($sp)
    sw $s0, 0($sp)
    add $s0, $0, $0
    add $t0, $0, $0
FOR: slt $t1, $t0, $a1
    add $a0, $s0, $0
    addi $a1, $a0, $0
    jal soma
    lw $a0, 16($sp)
    lw $a1, 8($sp)
    add $s0, $v0, $0
    j FOR
    add $v0, $s0, $0
    lw $ra, 4($sp)
    lw $s0, 0($sp)
    addi $sp, $sp, 16
    jr $ra
```

Convenção de registradores no MIPS

Registrador 1, $at, é reservado para o montador

Registradores 26-27, $k0-$k1, são reservados para o SO

Nome	Número do registrador	Uso	Preservado na chamada?
$zero	0	O valor constante 0	n.a.
$v0-$v1	2–3	Valores para resultados e avaliação de expressões	não
$a0-$a3	4–7	Argumentos	não
$t0-$t7	8–15	Temporários	não
$s0-$s7	16–23	Valores salvos	sim
$t8-$t9	24–25	Mais temporários	não
$gp	28	Ponteiro global	sim
$sp	29	Stack pointer	sim
$fp	30	Frame pointer	sim
$ra	31	Endereço de retorno	sim

Dados e caracteres

- Conjunto de caracteres codificados em byte
  - ASCII: 128 caracteres
    - 95 gráficos, 33 controle
  - Latin-1: 256 caracteres
    - ASCII + 96 caracteres gráficos adicionais
- Unicode: conjunto de caracteres de 32bits
  - Usado em Java, C++
  - A maioria dos alfabetos do mundo, mais símbolos
  - UTF-8, UTF-16: codificações de comprimento variável

Tabela ASCII

Valor ASCII	Sinal	Valor ASCII	Sinal	Valor ASCII	Sinal	Valor ASCII	Sinal	Valor ASCII	Sinal	Valor ASCII	Sinal
32	Espaç	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL

Operações de byte/halfword

- Poderia utilizar operações bit a bit
- MIPS byte/halfword load/store
  - Processamento de string é caso comum

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

- Sinal estendido para 32 bits em rt

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

- Zero estendido para 32 bits em rt

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

- Armazena somente byte/halfword mais à direita

Exemplo de cópia de string

Código em C

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

- Endereços x e y em $a0 e $a1
- i em $s0

```
strcpy:
    addi    $sp, $sp, -4          #ajusta pilha para 1 arg
    sw      $s0, 0($sp)           #salva o valor utilizado
    add     $s0, $zero, $zero     #i=0
L1:
    add     $t1, $s0, $a1         #t1 = &y[i]
    lbu     $t2, 0($t1)           #t2 =  y[i]
    add     $t3, $s0, $a0         #t3 = &x[i]
    sb      $t2, 0($t3)           #x[i]=y[i]
    beq     $t2, $0, L2           #se y[i]==0, vai para L2
    addi    $s0, $s0, 1           #i++
    j       L1                    #vai para L1
L2:                               #y[i]==0: fim da string
    lw      $s0, 0($sp)           #restaura $s0 antigo
    addi    $sp, $sp, 4           #retira 1 palavra da pilha
    jr      $ra                   #retorna para o último link
```

Constantes de 32 bits

- A maioria das constantes são pequenas
  - Imediato de 16 bits é suficiente
- Para constantes de 32 bits que aparecem ocasionalmente

```
lui rt, constant
```

  - copia a constante de 16 bits para os 16 bits mais significativos de rt
  - zera os 16 bits menos significativos de rt

```
lui $s0, 61
```

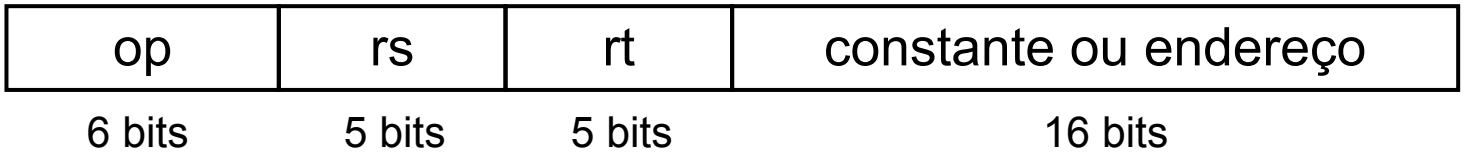
0000 0000 0011 1101 0000 0000 0000 0000

```
ori $s0, $s0, 4
```

0000 0000 0011 1101 0000 0000 0000 0100

Endereçamento de desvios condicionais

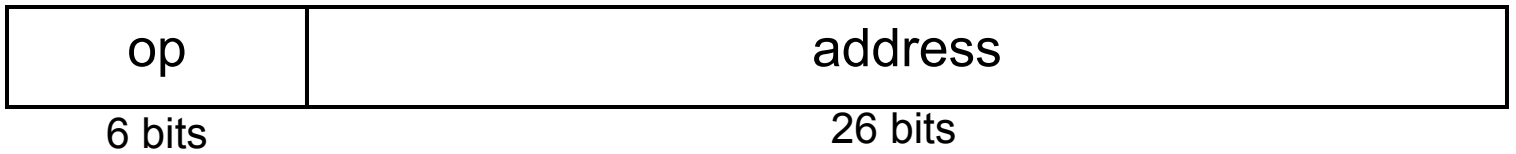
- Instruções de desvios especificam
  - opcode, dois registradores, endereço alvo
- A maioria dos desvios são para perto
  - Para frente ou para trás



- Endereçamento relativo ao PC
  - Endereço de destino = PC + (endereço x 4)
  - número de palavras para a próxima instrução

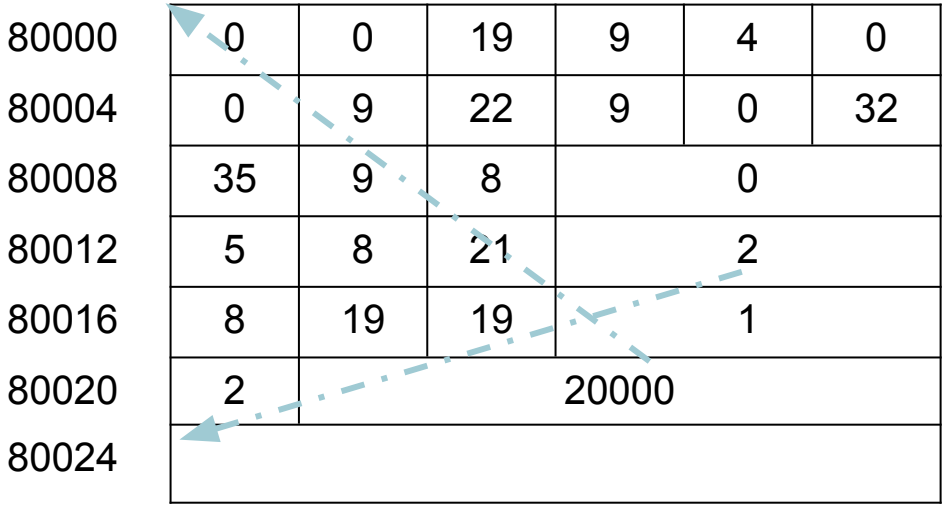
Endereçamento de jump

- Alvos do jump (j e jal) pode ser qualquer no segmento de código
  - Codifica o endereço na instrução
- Endereçamento (pseudo)direto
  - Endereço alvo: $PC_{31..28}$: address x 4
  - O endereço alvo concatena os 26 bits da instrução com MSB do PC



Exemplo de endereçamento de destino

```
LOOP:
    sll   $t1, $s3, 2
    add   $t1, $t1, $s6
    lw    $t0, 0($t1)
    bne   $t0, $s5, EXIT
    addi  $s3, $s3, 1
    j     LOOP
EXIT:
    …
```



Assumindo LOOP no endereço 80000

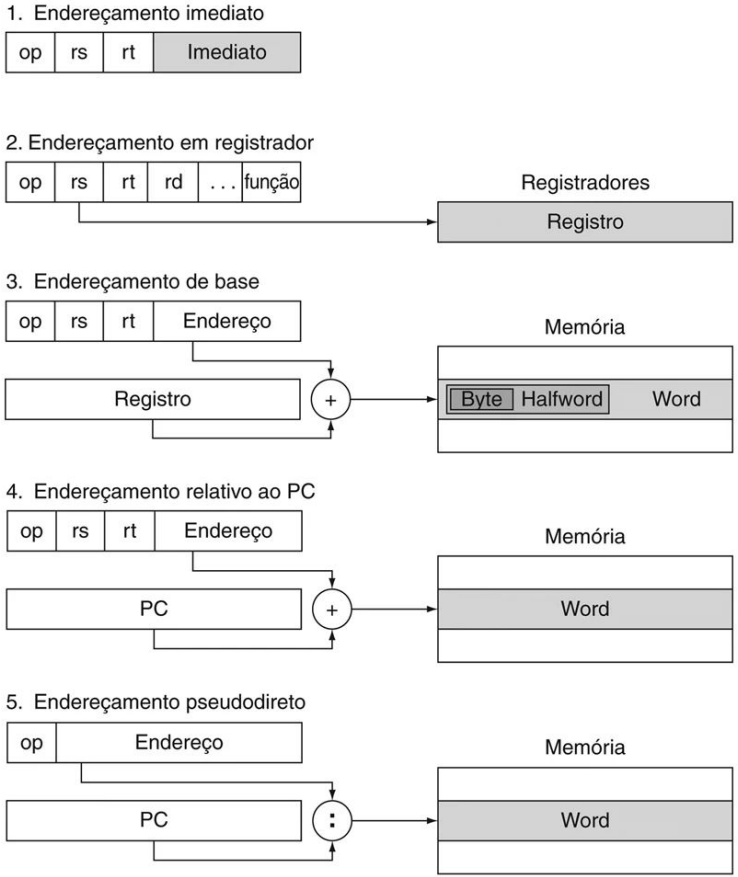
Desvio para mais longe

- Se o destino dos desvios é muito longe para codificar em 16 bits, o montador reescreve o código

```
beq $s0, $s1, L1
```

```
bne $s0, $s1, L2
j L1
L2: …
```

Resumo dos modos de endereçamento do MIPS



Capítulo 2 - A linguagem dos computadores

Capítulo 2 - A linguagem dos computadores

Implementações disponíveis no moodle

Exercícios sugeridos

2.23, 2.24, 2.26, 2.28, 2.30, 2.[32-34], 2.[37-42]

Referências

Materiais disponibilizados pelos professores Lucas Wanner, Paulo Gonçalves, Ricardo Pannain e Rogério Ap. Gonçalves.

Patterson, David A. Hennessy, John L. Organização e Projeto de Computadores. Disponível em: Minha Biblioteca, (5a. edição). Grupo GEN, 2017.

Slides do livro PATTERSON, David A. e HENNESSY, John L. *Computer Organization and Design Risc-V Edition: The Hardware Software Interface*. Estados Unidos, Ed. Morgan Kauffman, 2020.