



Capítulo 2

Instruções: Linguagem do Computador

Conjunto de Instrução

- O repertório de instruções de um computador
- Diferentes computadores possuem diferentes conjuntos de instrução
 - No entanto, com vários aspectos em comum
- Os primeiros computadores tinham conjuntos de instrução muito simples
 - Implementação simplificada
- Muitos computadores modernos também têm conjuntos de instruções simples

O Conjunto de Instrução do MIPS

- Usado como exemplo ao longo do livro
- Stanford MIPS comercializado pela MIPS Technologies (www.mips.com)
- Grande parte do mercado principal embutido
 - Aplicações em eletrônica de consumo, equipamento de rede/armazenamento, cameras, impressoras, ...
- Típico de vários ISAs modernos
 - Veja os dados de referência do MIPS e os Apêndices B e E

Operações Aritméticas

- Adicionar e subtrair, três operandos

- Duas fontes e um destino

add a, b, c # a gets b + c

- Todos os operadores aritméticos possuem esta forma

- *Princípio de Projeto 1*: Simplicidade favorece a regularidade

- Regularidade torna a implementação mais simples
 - Simplicidade permite maior desempenho com menor custo

Exemplo de Aritmética

- Código C:

$f = (g + h) - (i + j);$

- Código MIPS compilado:

```
add t0, g, h    # temp t0 = g + h
add t1, i, j    # temp t1 = i + j
sub f, t0, t1   # f = t0 - t1
```

Registrador Operandos

- Instruções de aritmética usam registrador como operandos
- MIPS tem um conjunto de registradores de 32 × 32-bit
 - Usa para dados acessados frequentemente
 - Numerado de 0 a 31
 - Dado de 32-bit, chamado “palavra”
- Montador (*assembler*) nomea
 - \$t0, \$t1, ..., \$t9 para valores temporários
 - \$s0, \$s1, ..., \$s7 para as variáveis salvas
- *Princípio de Projeto 2*: Quanto menor, mais rápido
 - c.f. memória principal: milhões de localizações

Exemplo de Registrador Operando

- Código C:

$f = (g + h) - (i + j);$

- f, \dots, j in $\$s0, \dots, \$s4$

- Código MIPS compilado:

add \$t0, \$s1, \$s2

add \$t1, \$s3, \$s4

sub \$s0, \$t0, \$t1

Memória Operandos

- Memória principal usada para dados compostos
 - Vetores, estruturas, dado dinâmico
- Para aplicar operações aritméticas
 - Carrega valores da memória para registradores
 - Armazena resultado do registrador para memória
- Memória é endereçado por *byte*
 - Cada endereço identifica um *byte* de 8-bit
- Palavras são alinhadas na memória
 - Endereço devem ser um múltiplo de 4
- MIPS é *Big Endian*
 - *Byte* mais significativo no menor endereço de uma palavra
 - *c.f. Little Endian: Byte* menos significativo no menor endereço

Exemplo 1: Memória Operando

- Código C:

`g = h + A[8];`

- `g` em `$s1`, `h` em `$s2`, endereço base de `A` em `$s3`

- Código MIPS compilado:

- Index 8 requer *offset* de 4×8 , ou 32: o endereço de carregamento selecionará `A[8]` e não `A[8/4]`

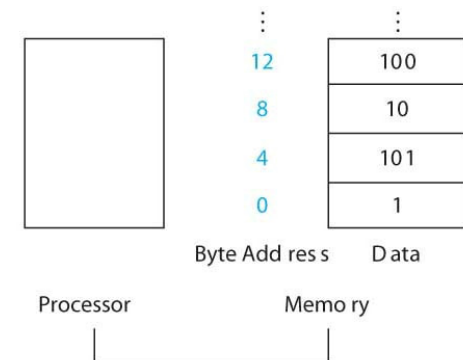
- 4 bytes por palavra

`lw $t0, 32($s3) # load word`

`add $s1, $s2, $t0`

offset

Registrador
base



Exemplo 2: Memória Operando

- Código C:

$A[12] = h + A[8];$

- h in $\$s2$, endereço base de A em $\$s3$

- Código MIPS compilado:

- Index 8 requer *offset* de 32
- Index 12 requer *offset* de 48

```
lw    $t0, 32($s3)    # load word
add   $t0, $s2, $t0
sw    $t0, 48($s3)    # store word
```

Registradores vs. Memória

- Registradores são mais rápidos para acessar do que a memória
- Operando em dados da memória requer operações de carregamento e armazenagem
 - Mais instruções a serem executadas (instrução de transferência somente lê ou escreve um operando)
 - Enquanto que uma instrução aritmética do MIPS lê dois registradores, opera sobre eles e escrever o resultado
- Compiladores devem usar registradores para variáveis tanto quanto possível
 - Passar somente para a memória variáveis que não são usadas frequentemente
 - Otimização de registrador é importante!

Operandos Imediatos

- Várias vezes um programa usará uma constante
- Dados constantes especificado em uma instrução
`addi $s3, $s3, 4`
- Nenhuma instrução imediata de subtração
 - Somente usa uma constante negativa
`addi $s2, $s1, -1`
- *Princípio de Projeto 3: Faça o caso comum rápido*
 - Pequenas constantes são comuns
 - Operando imediato evita uma instrução de carga

A Constante Zero

- Incluindo constantes dentro de instruções aritméticas, operações executam mais rápidas
 - também consomem menos energia do que se forem carregadas da memória

```
lw $t0, AddrConstant0($s1) #$t0 = 0  
add $s3, $s3, $t0 # $s3=$s3+$t0 ($t0=0)
```
- Registrador 0 (\$zero) do MIPS é a constante 0
 - Não pode ser substituído
- Útil para operações comuns
 - E.g., move conteúdo entre registradores

```
add $t2, $s1, $zero
```

Inteiros Binários sem Sinal

- Dado um número de n -bit

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

- Intervalo: 0 até $+2^n - 1$

- Exemplo

- $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
 $= 0 + \dots + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$
 $= 0 + \dots + 8 + 0 + 2 + 1 = 11_{10}$

- Usando 32 bits

- 0 to +4,294,967,295

Inteiros com Sinal Complemento de 2

- Dado um número de n-bit

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

- Intervalo: -2^{n-1} a $+2^{n-1} - 1$

- Exemplo

- $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
 $= -1 \times 2^{31} + 1 \times 2^{30} + \dots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$
 $= -2,147,483,648 + 2,147,483,644 = -4_{10}$

- Usando 32 bits

- $-2,147,483,648$ to $+2,147,483,647$

Inteiros com Sinal Complemento de 2

- Bit 31 é o bit de sinal
 - 1 para números negativos
 - 0 para números não negativos
- Números não negativos têm a mesma representação de complemento de 2 e sem sinal
- Alguns números específicos
 - 0: 0000 0000 ... 0000
 - -1: 1111 1111 ... 1111
 - O mais negativo: 1000 0000 ... 0000
 - O mais positivo: 0111 1111 ... 1111

Negação com Sinal

- Complementa e adiciona 1
 - Complemento significa $1 \rightarrow 0, 0 \rightarrow 1$

$$\begin{aligned}x + \bar{x} &= 1111 \dots 111_2 = -1 \\x + \bar{x} &= -1 \therefore \bar{x} + 1 = -x\end{aligned}$$

Exemplo: $x = 0100_2$
 $0100_2 + 1011_2 = 1111_2$

Negação com sinal consiste sempre no complemento e soma de 1

- Exemplo: negar +2
 - $+2 = 0000 \ 0000 \dots 0010_2$
 - $-2 = 1111 \ 1111 \dots 1101_2 + 1$
 $= 1111 \ 1111 \dots 1110_2$

Extensão com Sinal

- Representando um número com mais bits
 - Preserva o valor numérico
- No conjunto de instrução do MIPS
 - `addi`: estende o valor imediato
 - `lb`, `lh`: estende *byte/halfword* carregado
 - `beq`, `bne`: estende o deslocamento
- Replica o bit de sinal para a esquerda
 - c.f. valores sem sinal: estende com 0s
- Exemplos: 8-bit para 16-bit
 - `+2`: 0000 0010 => 0000 0000 0000 0010
 - `-2`: 1111 1110 => 1111 1111 1111 1110

Hexadecimal

- Base 16
 - Representação compacta de *strings* de *bit*
 - 4 bits por dígito hex

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

- Exemplo: eca8 6420
 - 1110 1100 1010 1000 0110 0100 0010 0000

Exercício 1

- Explore conversões de números a partir de números binárias com sinal e sem sinal para decimal

a. 0010 0100 1001 0010 0100 1001 0010 0100₂

b. 1101 1111 1011 1110 0100 0000 0000 0000₂

- 1) Qual é a representação dos números binários acima na base 10, assumindo um inteiro em complemento de 2?
- 2) Qual é a representação dos números binários acima na base 10, assumindo um inteiro sem sinal?
- 3) Qual é a representação dos números binários acima em hexadecimal?

Exercício 2

- Qual é o valor decimal deste número de complemento de 2 de 64 bits?

1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
1111 1111 1111 1111 1111 1000₂

- a) -4_{10}
- b) -8_{10}
- c) -16_{10}
- d) $18,446,744,073,709,551,609_{10}$

Exercício 3

- Estenda os seguintes números inteiros sem sinal para 16 bits:

a) $1000\ 0001_2$

b) $1111\ 1111_2$

Representando Instruções

- Instruções são codificadas em binário
 - Chamado código de máquina
- Instruções do MIPS
 - Codificadas como palavras de instrução de 32-bit
 - Pequeno número de formatos codificando código de operação (*opcode*), números de registrador, ...
 - Regularidade!
- Números do registrador
 - \$t0 – \$t7 são registradores 8 – 15
 - \$t8 – \$t9 são registradores reg's 24 – 25
 - \$s0 – \$s7 são registradores reg's 16 – 23

Instruções no Formato-R do MIPS

Formato-**R** significa formato para **registradores**



■ Campos de instrução

- **op**: código de operação (*opcode*)
- **rs**: primeiro número do registrador fonte
- **rt**: segundo número do registrador fonte
- **rd**: número do registrador de destino
- **shamt**: quantidade de deslocamento (00000 por agora)
- **funct**: código da função (estende *opcode*)



Exemplo do Formato-R

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

add \$t0, \$s1, \$s2

special	\$s1	\$s2	\$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

$00000010001100100100000000100000_2 = 02324020_{16}$

Instruções do Formato-I do MIPS



- Aritmética **imediata** e instruções de carregar / armazenar

- rt: número do registrador fonte ou destino
- Constante ou endereço: -2^{15} to $+2^{15} - 1$
- Endereço: *offset* adicionado ao endereço base em rs

Primeiro número do registrador fonte

- *Princípio de Projeto 4*: Um bom projeto exige bons compromissos

- Diferentes formatos complicam a decodificação, mas permitem instruções de 32-bit uniformemente
- Manter formatos o mais semelhante possível

Codificação das Instruções

- A tabela abaixo mostra os números usados em cada campo para as instruções MIPS cobertas aqui

Inst ruct ion	Format	op	rs	rt	rd	shamt	fun ct	addre ss
add	R	0	reg	reg	reg	0	32 _{ten}	n.a.
sub (subtract)	R	0	reg	reg	reg	0	34 _{ten}	n.a.
add immedi ate	I	8 _{ten}	reg	reg	n.a.	n.a.	n.a.	constant
lw (load word)	I	35 _{ten}	reg	reg	n.a.	n.a.	n.a.	address
sw (store word)	I	43 _{ten}	reg	reg	n.a.	n.a.	n.a.	address

Reg: número de registrador entre 0 e 31

Address: significa um endereço de 16-bit

n.a.: campo não aparece neste formato

Exemplo (1)

- Considere o seguinte trecho de código em C:

`A[300] = h + A[300];`

Compilado em

`lw $t0, 1200($t1) # $t0 gets A[300]`

`add $t0, $s2, $t0 # $t0 gets h + A[300]`

`sw $t0, 1200($t1) # stores h + A[300]`

back into A[300]

op	rs	rt	rd	address/shamt	funct
35	9	8		1200	
0	18	8	8	0	32
43	9	8		1200	

Exemplo (2)

- A instrução `lw` é identificada por 35 (`op`), o registrador base 9 (`$t1`) especificado no `rs` e o registrador destino 8 (`$t0`) no `rt`
- O *offset* para selecionar `A[300]` ($1200=300 \times 4$) é encontrado no endereço
 - Note a diferença entre as instruções `lw` e `sw`

100011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000 0100 1011 0000		

Exercício 1

- Qual instrução MIPS estes números representam?

Op	Rs	Rt	Rd	shamt	funct
0	8	9	10	0	34

1. sub \$t0, \$t1, \$t2
2. add \$t2, \$t0, \$t1
3. sub \$t2, \$t1, \$t0
4. sub \$t2, \$t0, \$t1

Exercício 2

- Mostre o binário e o hexadecimal para representar as seguintes instruções:
 - a) `addi $t0, $t0, 0`
 - b) `sw $t1, 32($t2)`

Exercício 3

- A tabela abaixo contém os valores de vários campos das instruções MIPS

a.	op=0, rs=8, rt=9, rd=10, shamt=0, funct=34
b.	op=0x23, rs=8, rt=9, const=0x4

- Qual tipo (Formato-I e Formato-R) de instrução estes valores representam?
- Quais são as instruções em assembly do MIPS descritas acima?
- Qual é a representação binária das instruções acima?

Exercício 4

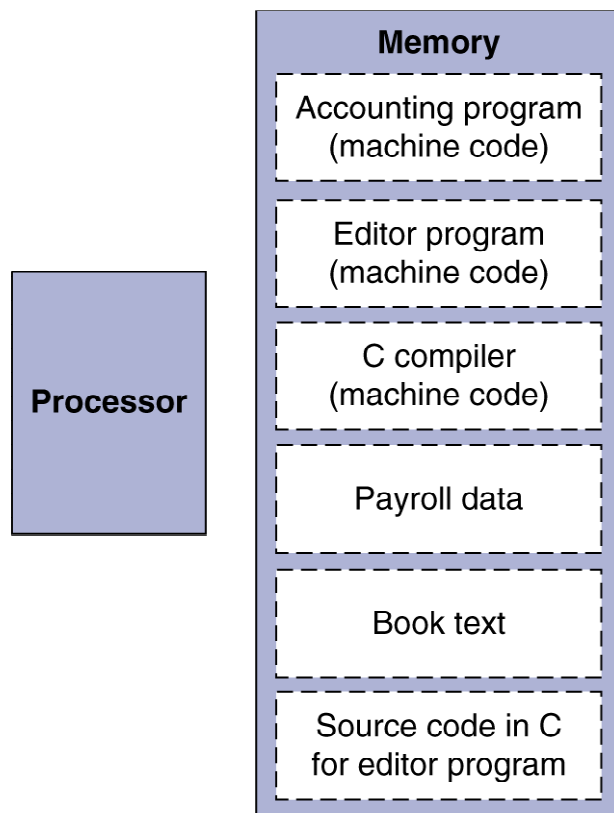
- A tabela abaixo contém os bits que representam o opcode de uma instrução

a.	0x01084020
b.	0x02538822

- Qual número binário representa o número hexadecimal acima?
- Qual número decimal representa o número hexadecimal acima?
- Qual instrução o número hexadecimal acima representa?

Armazenamento dos Programas do Computador

The BIG Picture



- Instruções representadas em binário assim como dados
- Instruções e dados armazenados na memória
- Programas podem operar em programas
 - e.g., compiladores, *linkers*, ...
- Compatibilidade binária permite que programas compilados funcionem em diferentes computadores
 - ISAs padronizados

Operações Lógicas

- Instruções para manipulação bit a bit

Operação	C	Java	MIPS
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bitwise AND	&	&	and, andi
Bitwise OR			or, ori
Bitwise NOT	~	~	nor

- Útil para extração e inserção de grupos de bits em uma palavra

Operações de Deslocamento

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- *shamt*: quantas posições para deslocar
- Lógica do deslocamento para esquerda
 - Desloca para esquerda e preenche com 0 bits
 - sll por i bits multiplica por 2^i
 - Exemplo: $0010 \rightarrow 1000$, onde $i=2$
- Lógica do deslocamento para direita
 - Desloca para direita e preenche com 0 bits
 - srl por i bits divide por 2^i (sem sinal somente)
 - Exemplo: $1000 \rightarrow 0010$, onde $i=2$

Operações de AND

- Útil para mascarar bits em uma palavra
 - Seleciona alguns bits, ajusta os demais para 0
- and \$t0, \$t1, \$t2

\$t2	0000 0000 0000 0000 0000 1101 1100 0000	→ Palavra a mascarar
\$t1	0000 0000 0000 0000 0011 1100 0000 0000	→ Máscara
\$t0	0000 0000 0000 0000 0000 1100 0000 0000	→ Resultado

Operações de OR

- Útil para incluir bits em uma palavra
 - Ajusta alguns bits para 1, deixa os demais permanecem inalterados

or \$t0, \$t1, \$t2

\$t2 0000 0000 0000 0000 0000 1101 1100 0000

\$t1 0000 0000 0000 0000 0011 1100 0000 0000

\$t0 0000 0000 0000 0000 0011 1101 1100 0000

Operações de NOT

- Útil para inverter bits em uma palavra
 - Muda de 0 para 1, e de 1 para 0
- MIPS tem instrução NOR de 3 operandos
 - $a \text{ NOR } b == \text{NOT} (a \text{ OR } b)$

`nor $t0, $t1, $zero` ←

Registrador 0:
sempre lê como
zero

Fonte \$t1 0000 0000 0000 0000 0011 1100 0000 0000

Destino \$t0 1111 1111 1111 1111 1100 0011 1111 1111

Operações Condicionais

- Desvia para uma instrução rotulada se a condição for verdadeira
 - Caso contrário, continua sequencialmente
- `beq rs, rt, L1`
 - se $(rs == rt)$ desvia para instrução rotulada L1;
- `bne rs, rt, L1`
 - se $(rs != rt)$ desvia para instrução rotulada L1;
- `j L1`
 - salto incondicional para instrução rotulada L1

Compilando Instruções *If*

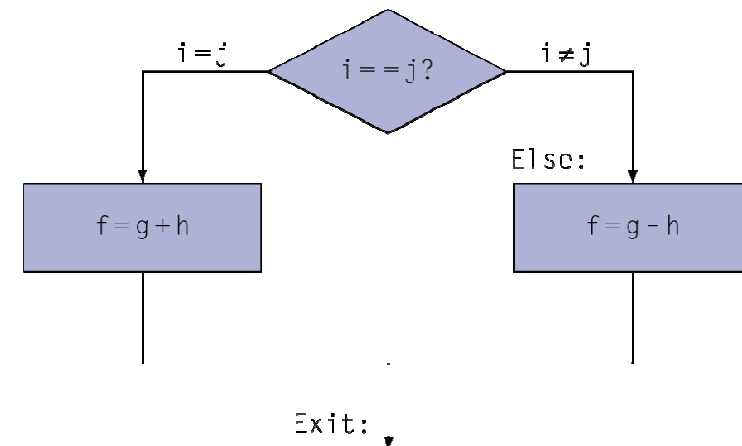
- Código C:

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

- f, g, ... in \$s0, \$s1, ...

- Código MIPS compilado:

```
        bne $s3, $s4, Else  
        add $s0, $s1, $s2  
        j   Exit  
Else:   sub $s0, $s1, $s2  
Exit:   ...
```



Montador calcula endereços

Compilando Instruções de Laço

- Código C:

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

- i em \$s3, k em \$s5, endereço de save em \$s6

offset

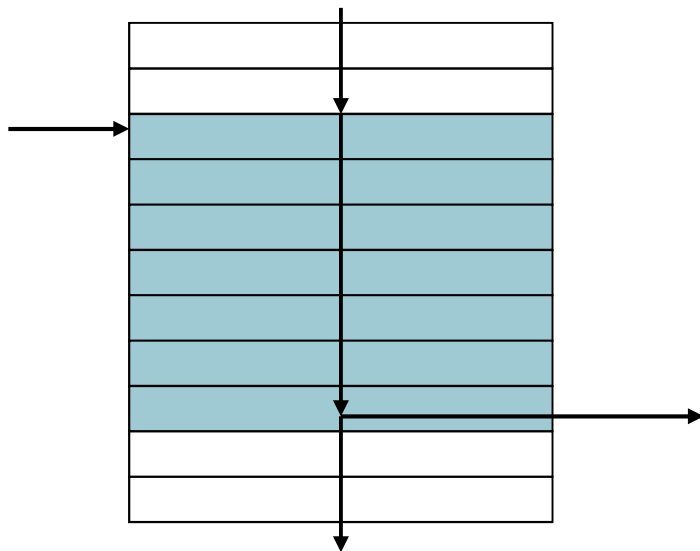
- Código MIPS compilado:

```
Loop: sll    $t1, $s3, 2          #$t1 = i * 4
      add    $t1, $t1, $s6       #$t1 = endereço
                                de save[i]
      lw     $t0, 0($t1)         #$t0 = save[i]
      bne    $t0, $s5, Exit      # vai para Exit
                                se save[i]≠k
      addi   $s3, $s3, 1         #i = i + 1
      j      Loop               #vai para loop

Exit: ...
```

Blocos Básicos

- Um bloco básico é uma sequência de instruções que consiste
 - Sem desvios embutidos (exceto no fim)
 - Sem desvios rotulados (exceto no início)



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

Mais Operações Condicionais

- Ajusta resultado para 1 se uma condição for verdadeira
 - Caso contrário, ajusta para 0
- `slt rd, rs, rt`
 - se $(rs < rt)$ $rd = 1$; senão $rd = 0$;
- `slti rt, rs, constant`
 - se $(rs < constant)$ $rt = 1$; senão $rt = 0$;
- Use em combinação com `beq`, `bne`

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

Conversão de Operadores (1)

```
if (a>b)
```

```
    L1:
```

```
else
```

```
    L2:
```

```
Exit:
```



```
slt $t0, $t1, $t2
```

```
bne $t0, $zero, L2
```

```
beq $t1, $t2, L2
```

```
L1:
```

```
j Exit
```

```
L2:
```

```
if (a>=b)
```

```
    L1:
```

```
else
```

```
    L2:
```

```
Exit:
```



```
slt $t0, $t1, $t2
```

```
bne $t0, $zero, L2
```

```
L1:
```

```
j Exit
```

```
L2:
```

Conversão de Operadores (2)

```
if (a<=b)
```

```
    L1:
```

```
else
```

```
    L2:
```

```
Exit:
```



```
slt $t0, $t1, $t2
```

```
bne $t0, $zero, L1
```

```
beq $t1, $t2, L1
```

```
L2:
```

```
j Exit
```

```
L1:
```

Projeto de Instrução de Desvio

- Por que não b1e, bge, etc?
- Hardware para \leq , \geq , ... mais lento do que $=$, \neq
 - Atendendo von Neumann sobre simplicidade, a arquitetura MIPS não inclui desvio em “<”
 - Combinando “<” com desvio envolve mais trabalho por instrução, exigindo ciclos de relógio extra ou um relógio mais rápido
- beq e bne são os casos comuns
 - Duas instruções rápidas são mais proveitosas
- Este é um bom compromisso de projeto

Com Sinal vs. Sem Sinal

- Comparação com sinal: `slt`, `slti`
- Comparação sem sinal: `sltu`, `sltui`
- Exemplo
 - 1 na parte mais significativa representa números negativos
 - `$s0 = 1111 1111 1111 1111 1111 1111 1111 1111`
 - `$s1 = 0000 0000 0000 0000 0000 0000 0000 0001`
 - `slt $t0, $s0, $s1 # signed`
 - $-1 < +1 \Rightarrow \$t0 = 1$ Número negativo é menor do que qualquer número positivo
 - `sltu $t0, $s0, $s1 # unsigned`
 - $+4,294,967,295 > +1 \Rightarrow \$t0 = 0$

Reflexão

- C contém várias instruções para laços e decisões, enquanto MIPS tem poucas. Quais das seguintes frases explicam ou não explicam este desequilíbrio
 1. Mais instruções de decisão tornam o código mais fácil de ler e entender
 2. Poucas instruções de decisão simplificam a tarefa da camada subjacente que é responsável pela execução
 3. Mais instruções de decisão significam menos linhas de código, que geralmente reduz o tempo de codificação
 4. Mais instruções de decisão significam menos linhas de código, que geralmente resulta na execução de menos operações

Exercício 1

- A tabela abaixo contém várias operações lógicas do MIPS.

a.	sll \$t2, \$t0, 1 andi \$t2, \$t2, -1
b.	andi \$t2, \$t1, 0x00F0 srl \$t2, 2

- a) Assuma que \$t0 = 0x0000A5A5 e \$t1 = 0x00005A5A. Qual é o valor de \$t2 depois das duas instruções na tabela?
- b) Assuma que \$t0 = 0xA5A50000 e \$t1 = 0xA5A50000. Qual é o valor de \$t2 depois das duas instruções na tabela?

Exercício 2

- Suponha que os registradores \$t0 e \$t1 contêm os seguintes valores, resp.:

0010 0100 1001 0010 0100 1001 0010 0100₂

0011 1111 1111 1000 0000 0000 0000 0000₂

Qual é o valor de \$t2 depois de executar as seguintes instruções:

```
slt    $t2, $t0, $t1
beq    $t2, $zero, ELSE
j      DONE
```

```
ELSE:  addi $t2, $zero, 2
```

```
DONE:
```

Exercício 3

- Considere que as variáveis f e g são inteiros de 32 bits
 - a) $f = -g - f$;
 - b) $f = g + (-f - 5)$;
- Para o código C acima, determine
 - 1) As instruções assembly do MIPS (use um número mínimo de instruções)
 - 2) O código assembly do MIPS correspondente
- Se as variáveis f e g possuem valores 1 e 2, qual é o valor final de f ?

Chamada de Procedimento

- Passos necessários
 1. Colocar parâmetros nos registradores
 2. Transferir o controle para o procedimento
 3. Adquirir armazenamento para procedimento
 4. Executar operações do procedimento
 5. Colocar resultados no registrador de chamada
 6. Retornar para o local de chamada

Uso dos Registradores do MIPS

- \$a0 – \$a3: argumentos (reg's 4 – 7)
- \$v0, \$v1: valores do resultado (reg's 2 e 3)
- \$t0 – \$t9: temporários
 - Podem ser sobre-escritos pela chamada
- \$s0 – \$s7: salvo
 - Deve ser salvo/restaurado pela chamada
- \$gp: ponteiro global para dados estáticos (reg 28)
- \$sp: ponteiro de pilha (reg 29)
- \$fp: ponteiro do quadro (reg 30)
- \$ra: endereço de retorno (reg 31)

Instruções de Chamada de Procedimento

- Chamada do procedimento (*caller*): *jump-and-link*
 - Armazena os valores dos parâmetros em \$a0–\$a3 e usa: `jal ProcedureLabel`
 - Endereço da instrução seguinte colocado em \$ra
 - Salta para o endereço de destino
- Retorno do procedimento (*callee*): *jump register*
 - Armazena os resultados em \$v0 e \$v1 e retorna o controle para o *caller* usando: `jr $ra`
 - Copia \$ra para contador de programa (PC)
 - Também pode ser usado para saltos computados
 - e.g., para instruções *case/switch*
 - A Instrução `jal` salva PC + 4 no registrador \$ra



Exemplo de Procedimento Folha (1)

- Suponha que precisamos de mais registradores para um procedimento (4–argum. e 2–val.)

- Código em C:

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

- Argumentos g, ..., j em \$a0, ..., \$a3
- f em \$s0 (daí a necessidade de salvar \$s0 na pilha)
- Resultado em \$v0

Exemplo de Procedimento Folha (2)

■ Código MIPS:

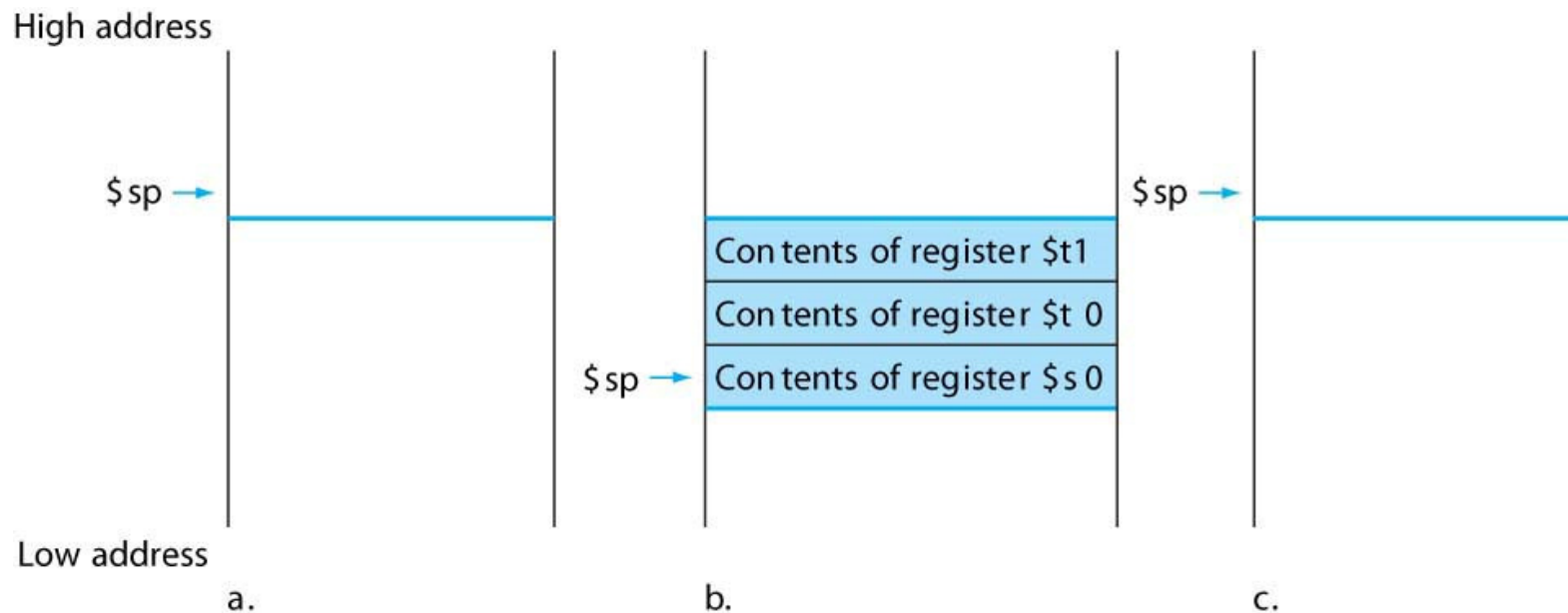
	leaf_example:	
	addi \$sp, \$sp, -4	Ajusta \$sp para armazenar um item
push →	sw \$s0, 0(\$sp)	Salva \$s0 na pilha para uso posterior
	add \$t0, \$a0, \$a1	Corpo do procedimento $f = (g+h) - (i+j)$
	add \$t1, \$a2, \$a3	
	sub \$s0, \$t0, \$t1	
	add \$v0, \$s0, \$zero	Copia resultado para \$v0
pop →	lw \$s0, 0(\$sp)	Restaura \$s0
	addi \$sp, \$sp, 4	Ajusta pilha para apagar 1 item
	jr \$ra	Salta p/ end. de retorno

Nós devemos salvar e restaurar \$s0, pois o *callee* deve assumir que o *caller* precisa deste valor



Exemplo de Procedimento Folha (3)

Os valores do ponteiro de pilha e a pilha (a) antes, (b) durante e (c) depois da chamada de procedimento



O ponteiro de pilha sempre aponta para o topo da pilha ou para a última palavra na pilha

Procedimentos Não-Folhas

- Procedimentos que chamam outros procedimentos
 - Procedimentos recursivos invocam até “clones” deles mesmo
- Para chamada aninhada, procedimento que chama (*caller*) precisa salvar na pilha:
 - O endereço de retorno
 - Quaisquer argumentos e temporários necessários após a chamada
- Restaura a partir da pilha após a chamada

Exemplo de Procedimento Não-Folha

- Código em C para calcular o fatorial:

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

- Argumento n em \$a0
- Resultado em \$v0

Exemplo de Procedimento Não-Folha

■ Código MIPS:

fact:		
push →	addi \$sp, \$sp, -8	#ajusta pilha para 2 itens
	sw \$ra, 4(\$sp)	#salva endereço de retorno
	sw \$a0, 0(\$sp)	#salva argumento
	slti \$t0, \$a0, 1	#testa para $n < 1$
	beq \$t0, \$zero, L1	
	addi \$v0, \$zero, 1	#se positivo, resultado é 1
	addi \$sp, \$sp, 8	# pop 2 itens da pilha
	jr \$ra	# e retorna para <i>caller</i>
	L1: addi \$a0, \$a0, -1	#senão decrementa n
	jal fact	#chama <i>fact</i> com (n-1)
pop →	lw \$a0, 0(\$sp)	#restaura original n
	lw \$ra, 4(\$sp)	# e endereço de retorno
	addi \$sp, \$sp, 8	#pop 2 itens da pilha
	mul \$v0, \$a0, \$v0	#mult. p/ obter resultado
	jr \$ra	#e retorna para <i>caller</i>

0
A
1
A
2
A
3
Z

O que é preservado na chamada

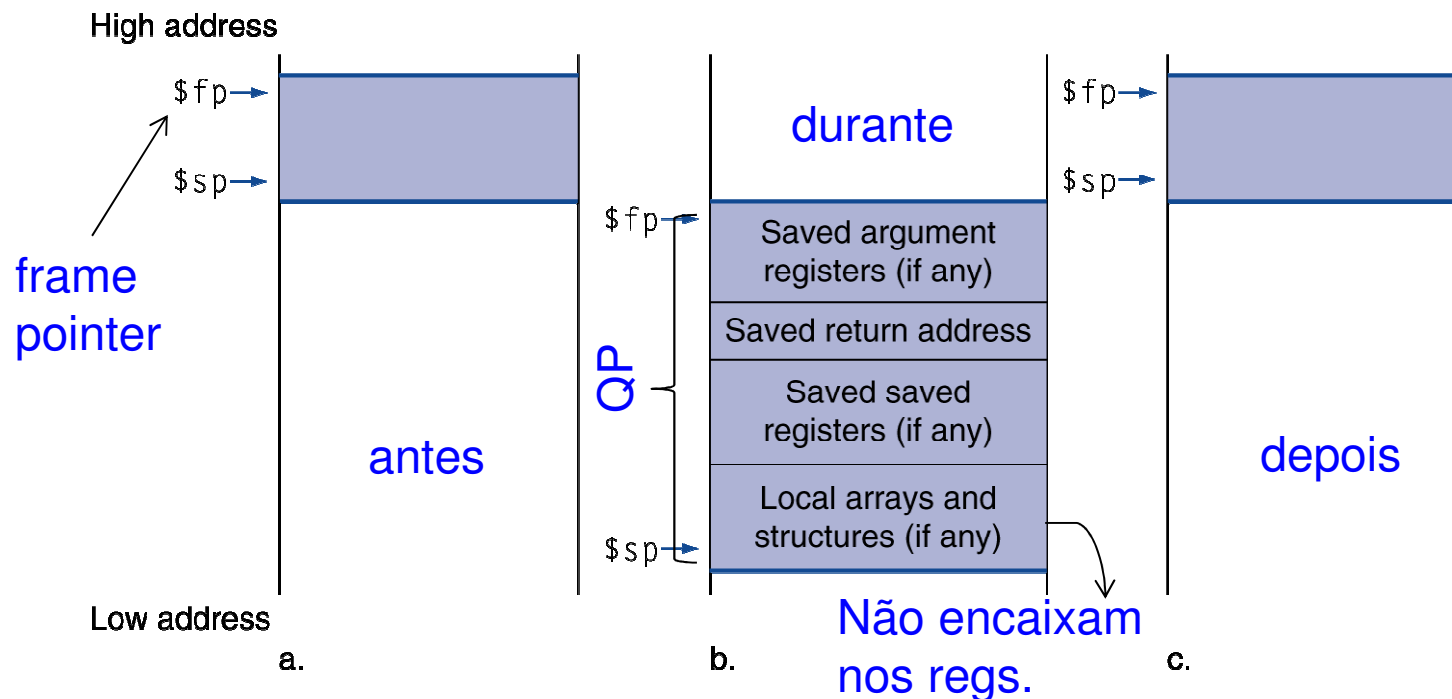
- A pilha acima de \$sp é preservada assegurando que o *callee* não escreve acima de \$sp
 - \$sp é preservado pelo callee adicionando exatamente a mesma quantidade que foi subtraído
 - Outros registradores são preservados salvando na pilha e restaurando a partir dela

Preservado	Não preservado
Reg. salvo: \$s0-\$s7	Reg. temporários: \$t0-\$t9
Reg. do ponteiro de pilha: \$sp	Reg. de argumento: \$a0-\$a3
Reg. do retorno de endereço: \$ra	Reg. valor de retorno: \$v0-\$v1
Pilha acima do ponteiro de pilha	Pilha abaixo do ponteiro de pilha

Variáveis em C

- Uma variável em C é geralmente uma localização no armazenamento
 - Depende do tipo e classe de armazenamento (*automatic* e *static*)
 - Exemplos incluem inteiro e caracter
- Variáveis automáticas são locais para um procedimento
 - São descartadas quando o procedimento termina
- Variáveis estáticas mantêm o valor de uma chamada da função para a outra
 - Variáveis declaradas fora do procedimento são consideradas estáticas

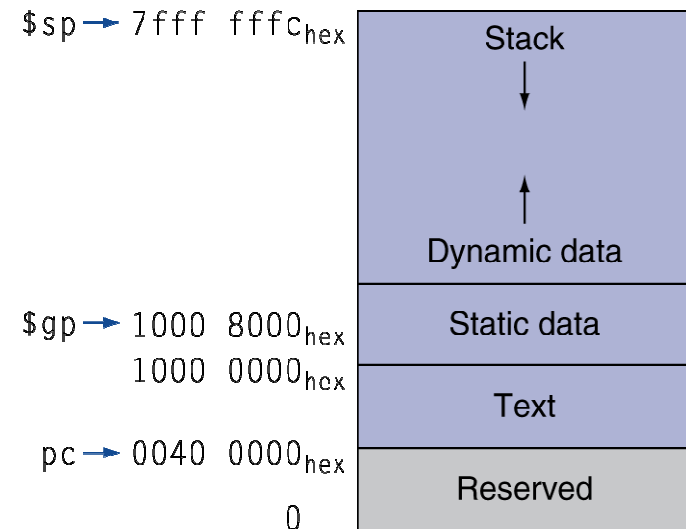
Alocando Espaço na Pilha



- Dados locais alocados pelo proc. chamado
 - e.g., variáveis automáticas C
- Quadro de procedimento (registro de ativação)
 - Usado por alguns compiladores para gerenciar o armazenamento da pilha

Alocando Espaço no Heap

- Texto: código de máquina
- Dado estático: variáveis globais
 - e.g., variáveis estáticas em C, vetores e strings constantes
 - \$gp inicializado para o endereço permitindo \pm offsets dentro deste segmento
- Dado dinâmico: heap
 - E.g., *malloc* (C), *new* (Java)
- Pilha: armazenamento automático



Uso eficiente da memória
a medida que os dois
segmentos (pilha e *heap*)
aumentam e diminuem

Reflexão

- Quais das seguintes frases sobre C e Java são geralmente verdadeiras?
 1. Programadores C gerenciam dados explicitamente, enquanto em Java isto é automático
 2. C leva a mais erros de ponteiros (e.g., *double free*) e vazamento de memória do que em Java

Exemplo (1)

- Considere o seguinte trecho de código

```
int sum(int n, int acc) {  
    if (n>0)  
        return sum(n-1, acc+n);  
    else  
        return acc;  
}
```

- Qual é o resultado das chamadas recursivas para `sum(3,0)`?
 - `sum(3,0) => sum(2,3) => sum(1,5) => sum(0,6)`

Exemplo (2)

■ Código MIPS

```
sum: slti $t0, $a0, 1      #testa se n <= 0
     bne $t0, $zero, sum_e #salta para sum_e se
                           #n<=0
     add $a1, $a1, $a0     #adiciona n para acc
     addi $a0, $a0, -1     #subtrai 1 de n
     j sum                 #salta para sum
sum_e:
     add $v0, $a1, $zero  #retorna valor do acumulador
     jr $ra               #retorna para o caller
```

Convenção dos Regs. Do MIPS

- Reg. 1, \$at, é reservado para o montador
- Reg. 26-27, \$k0-\$k1, são reservados para o SO

Name	Register number	Usage	Preserved on call?
\$zero	0	The constant value 0	n.a.
\$v0-\$v1	2-3	Values for results and expression evaluation	no
\$a0-\$a3	4-7	Arguments	no
\$t0-\$t7	8-15	Temporaries	no
\$s0-\$s7	16-23	Saved	yes
\$t8-\$t9	24-25	More temporaries	no
\$gp	28	Global pointer	yes
\$sp	29	Stack pointer	yes
\$fp	30	Frame pointer	yes
\$ra	31	Return address	yes

Exercício 1

- Responda as seguintes perguntas para este código C:

```
for(i=0; i<a; i++)  
    a += b;
```

- a) Qual é grafo de fluxo de controle?
- b) Traduza o código C para o código em assembly do MIPS. Use um número mínimo de instruções. Assuma que os valores de a, b e i estão nos registradores \$s0, \$s1 e \$t0, resp.
- c) Se as variáveis a e b forem inicializadas para 10 e 1, quantas instruções do MIPS serão executadas para completar o loop?

Exercício 2

- Qual é o código MIPS para o procedimento **fib**?

```
int fib(int n) {  
    if (n==0)  
        return 0;  
    else if (n == 1)  
        return 1;  
    else  
        return fib(n-1) + fib(n-2);  
}
```

Dados de Caracter

- Conjuntos de caracteres codificados por **8-bits**
 - ASCII: 128 caracteres
 - 95 de gráfico, 33 de controle
 - Latin-1: 256 caracteres
 - ASCII, +96 caracteres gráficos
- Unicode: Conjunto de caractere de **32-bits**
 - Usado em Java, caracteres amplos do C++, ...
 - A maioria dos alfabetos do mundo, mais símbolos
 - UTF-8, UTF-16: codificação de comprimento variável

Tabela ASCII

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	 	Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	\$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	((72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051))	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[[123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174	|	
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135]]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177		DEL

Source: www.LookupTables.com



<http://cs.stanford.edu/~miles/iso8859.html>

Operações de Byte/Halfword

- MIPS **b**yte (8 bits)/**h**alfword (16 bits) load/store

- Processamento de string é um caso comum

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

- Extensão com sinal para 32 bits em rt

- Copia o bit do sinal repetidamente para preencher o resto do registrador

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

- Extensão com zero para 32 bits em rt

- Preenche com 0s para a esquerda do dado

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

- Armazena somente byte/halfword mais da direita

Caacteres e String em Java

- Java usa Unicode para caracteres
 - Usa 16 bits para representar o caractere
- Strings são uma classe padrão em Java
 - Métodos pré-definidos para concatenação, comparação e conversão
- Java inclui um método que fornece o comprimento da string

Reflexão

- Quais das seguintes frases sobre caracteres e strings em C e Java, são verdadeiras?
 1. Uma string em C consome mais memória do que uma string em Java
 2. Strings são somente um nome informal para vetores de caracteres uni-dimensional em C e Java
 3. String em C e Java usam NULL (0) para marcar o final de uma string
 4. Operações em strings, como comprimento, são mais rápidas em C do que em Java

Exemplo de Cópia de *String*

- Código em C (naïve):

- String com final NULL

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

- Endereço de x, y em \$a0, \$a1
- i em \$s0

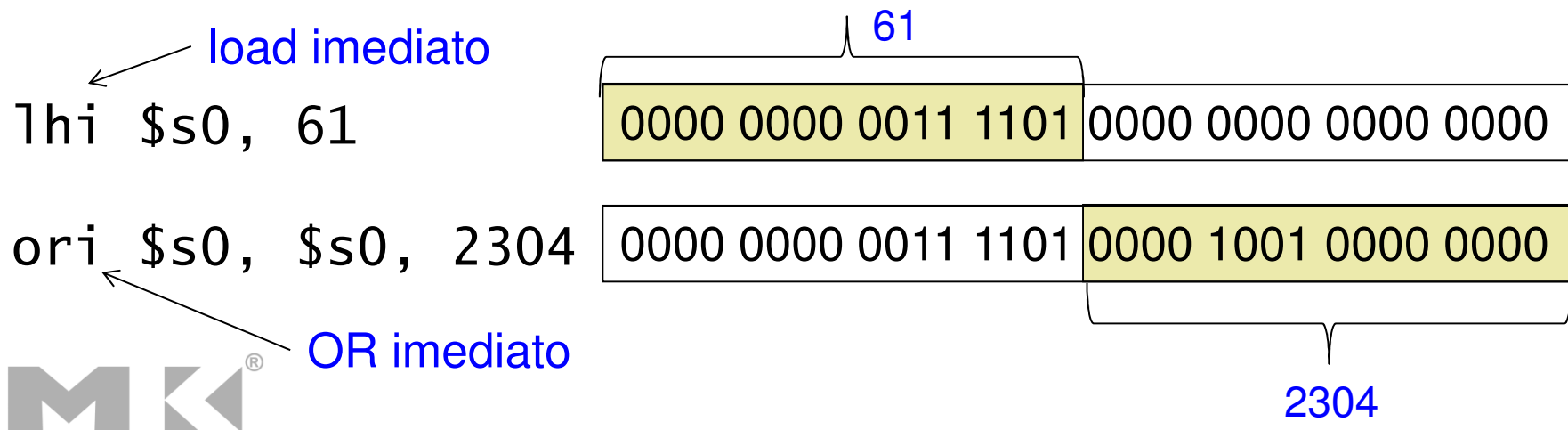
Exemplo de Cópia de *String*

- Código em MIPS:

strcpy:		
	addi \$sp, \$sp, -4	# ajusta pilha para 1 item
	sw \$s0, 0(\$sp)	# salva \$s0
	add \$s0, \$zero, \$zero	# i = 0
L1:	add \$t1, \$s0, \$a1	# endereço de y[i] em \$t1
	lb \$t2, 0(\$t1)	# \$t2 = y[i]
	add \$t3, \$s0, \$a0	# endereço de x[i] em \$t3
	sb \$t2, 0(\$t3)	# x[i] = y[i]
	beq \$t2, \$zero, L2	# sai do laço se y[i] == 0
	addi \$s0, \$s0, 1	# i = i + 1
	j L1	# próxima iteração do laço
L2:	lw \$s0, 0(\$sp)	# restaura \$s0
	addi \$sp, \$sp, 4	# remove 1 item da pilha
	jr \$ra	# e retorna

Constantes de 32-bit

- A maioria das constantes são pequenas
 - Imediato 16-bit é suficiente
- Porém, para uma constante de 32-bit (ocasional)
`lui rt, constant`
 - Copia 16-bit constante para a esquerda dos 16 bits de `rt`
 - Ajusta os 16 bits da direita para zero



Endereçamento de Desvio

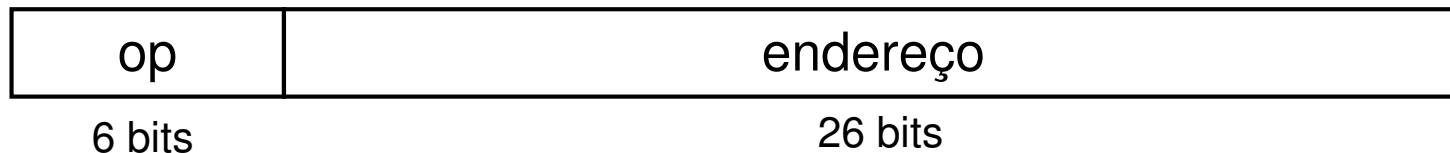
- Instruções de desvio especificam
 - Opcode, dois registradores, endereço alvo
- A maioria dos alvos de desvio são desvios próximos
 - Para frente ou para trás



- Endereçamento relativo do PC
 - $\text{Endereço alvo} = \text{PC} + \text{offset} \times 4$
 - PC já incrementado por 4 nesta altura

Endereçamento de Salto

- Salta (j e jal) alvos poderia estar em qualquer lugar do segmento de texto
 - Codifica endereço completo na instrução



- (Pseudo) Endereçamento de salto direto
 - $\text{Endereço alvo} = PC_{31...28} : (\text{endereço} \times 4)$

Exemplo de Endereçamento Alvo

- Código do laço de um exemplo anterior
 - Assume laço na localização 80000

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

Shift Left Logical

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

k

i

save

Temp reg \$t1 = 4*i
\$t1 = endereço de save[i]
Temp reg \$t0 = save[i]
vai para Exit se save[i]!=k
i = i + 1
vai para Loop

```
Exit: ...
```

Exemplo de Endereçamento Alvo

■ Código do laço de um exemplo anterior

■ Assume laço na localização 80000

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

\$t0 – \$t7 são regs 8 – 15

\$t8 – \$t9 são regs 24 – 25

\$s0 – \$s7 são regs 16 – 23

Shift Left Logical

Loop: sll \$t1, \$s3, 2
 add \$t1, \$t1, \$s6
 lw \$t0, 0(\$t1)
 bne \$t0, \$s5, Exit
 addi \$s3, \$s3, 1
 j Loop
 Exit: ...

i
 save
 k

80000
 80004
 80008
 80012
 80016
 80020
 80024

opcode	rs	rt	rd	shamt	funct
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				

$$80016 + (4 \times 2) = 80024; 4 \times 2000 = 8000$$

Desvio distante

- Se o alvo do desvio for muito distante para codificar com um *offset* de 16 bits, o montador reescreve o código
- Exemplo:

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

```
L2:  ...
```



```
bne $s0,$s1, L2
```

```
j  L1
```

```
L2:  ...
```

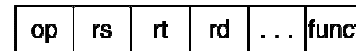
Resumo do Modo de Endereçamento

add, sub, ...

1. Immediate addressing



2. Register addressing



Registers

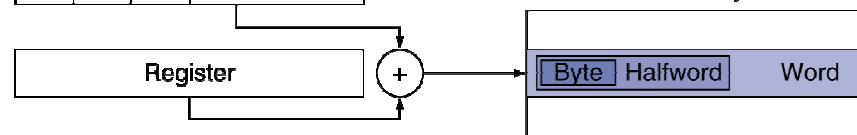
Register

addi, lw, ...

3. Base addressing

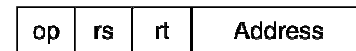


Memory

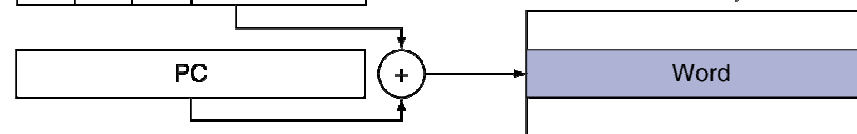


beq, bne, ...

4. PC-relative addressing

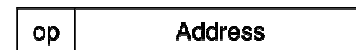


Memory

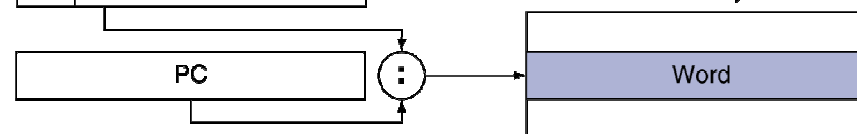


j, jal, ...

5. Pseudodirect addressing



Memory



Sincronização

- Dois processadores compartilhando uma mesma área de memória
 - P1 escreve, então P2 lê
 - Corrida de dado se P1 e P2 não sincronizarem
 - Resultado depende da ordem de acessos
- Suporte de hardware necessário
 - Operações atômicas de memória leitura/escrita
 - Nenhum outro acesso ao local permitido entre ler e escrever
- Poderia ser uma única instrução
 - E.g., troca atômica de R/W: registrador \leftrightarrow memória
 - Ou um par atômico de instruções



Sincronização no MIPS

- Carregamento acoplado: `ll rt, offset(rs)`
- Armazenamento condicional: `sc rt, offset(rs)`
 - Sucede se a localização não mudou desde a execução de `ll`
 - Retorna 1 em `rt`
 - Falha se a localização for alterada
 - Retorna 0 em `rt`
- Exemplo: troca atômica entre `$s1` e `$s4` (para testar/ajustar a variável *lock*)

```
try: add $t0,$zero,$s4 ;copia valor de troca
      ll  $t1,0($s1)    ;carregamento acoplado
      sc  $t0,0($s1)    ;armazenamento condicional
      beq $t0,$zero,try ;desvia se o
                        ;armazenamento falhar
      add $s4,$zero,$t1 ;coloca valor de carga
                        em $s4
```

Reflexão

- Quando você deve usar primitivas como carregamento acoplado e armazenamento condicional?
 1. Quando os processos são independentes
 2. Quando os processos cooperativos de um programa em paralelo precisam sincronizar para obter o comportamento adequado para leitura e escrita de dados compartilhados
 3. Quando processos cooperativos em um único processador precisam sincronizar para escrever e ler dados compartilhados

Exercício (1)

- Qual é a representação em C da string “Cal”?

Resp.: 67, 97, 108, 0 → 0x43, 0x61, 0x6c, 0x00
0100 0011 0110 0001 0110 1100 0000 0000₂

Note que strings possuem tamanho variável. Outras formas de representação:

1. Guardar na primeira posição o tamanho da string
2. Adicionar uma variável adicional para guardar o tamanho da string (estrutura)

Exercício (2)

- A seguinte tabela mostra os valores de caractere ASCII em hexadecimal

a.	41 44 44
b.	4D 49 50 53

Traduz o valores do ASCII hexadecimal para texto

Exercício (3)

- Qual é o intervalo de endereços para desvios condicionais no MIPS (K=1024)
 - a) Endereços entre 0 e 64K-1
 - b) Endereços entre 0 e 256K-1
 - c) Endereços até aprox. 32K antes do desvio e aprox. 32K depois
 - d) Endereços até aprox. 128K antes do desvio e aprox. 128K depois

Exercício (4)

- Qual é o intervalo de endereços para *jump* e *jump-and-link* no MIPS (M=1024K)
 - a) Endereços entre 0 e 64M-1
 - b) Endereços entre 0 e 256M-1
 - c) Endereços até aprox. 32M antes do desvio e aprox. 32M depois
 - d) Endereços até aprox. 128M antes do desvio e aprox. 128M depois
 - e) Qualquer lugar dentro de um bloco de 64M de endereços onde o PC fornece os 6 bits mais significativos
 - f) Qualquer lugar dentro de um bloco de 256M de endereços onde o PC fornece os 4 bits mais significativos

Exercício (5)

- Qual é a instrução da linguagem assembly do MIPS correspondente a instrução de máquina com o valor 0000 0000_{hex} ?
 - a) j
 - b) formato-R
 - c) addi
 - d) sll
 - e) mfc0 (*move from coprocessor 0*)
 - f) Opcode indefinido: não existe instrução legal que corresponde a 0

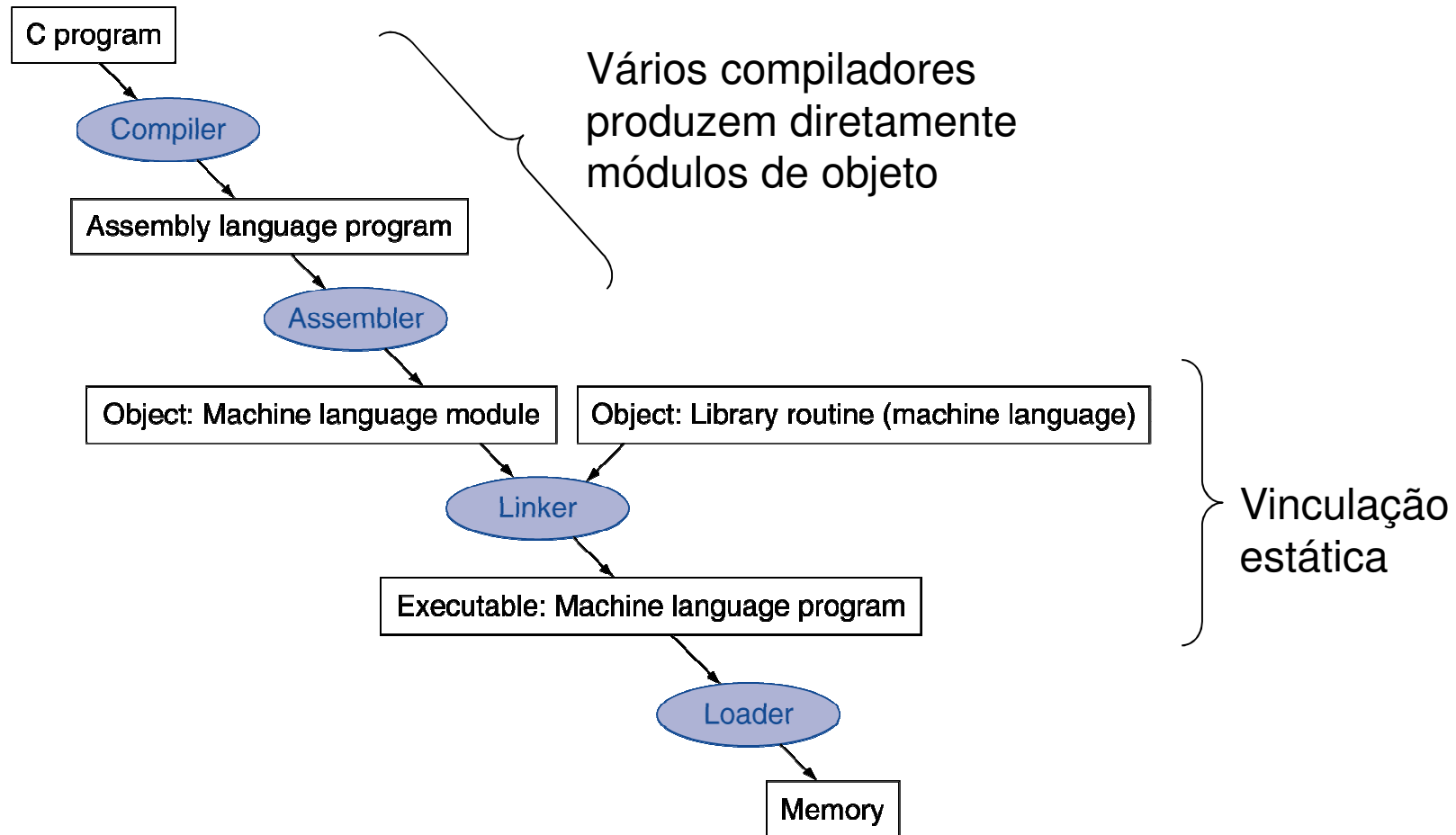
Exercício (6)

- Para os seguintes problemas, considere:

a.	lui \$t0, 0x1234 addi \$t0, \$t0, 0x5678
b.	lui \$t0, 0x1234 andi \$t0, \$t0, 0x5678

Qual é o valor do registrador \$t0 depois de executar a sequência de código da tabela acima?


Tradução e *Startup*



Montador Pseudo-instruções

- A maioria das instruções do montador representam instruções de máquina um-para-um
- Pseudo-instruções: fruto da imaginação do montador (não estão implementadas em HW)

`move $t0, $t1` → `add $t0, $zero, $t1`

 `blt $t0, $t1, L` → `slt $at, $t0, $t1`
`bne $at, $zero, L`
branch on less than

- `$at` (registrador 1): montador temporário

- As instruções `move` e `blt` são aceitas no MIPS e simplificam a tradução e programação

Produzindo um Módulo de Objeto (1)

- Montador (ou compilador) traduz programa em instruções de máquina
- Fornece informações para construir um programa completo a partir de pedaços
 - **Header**: descreve o tamanho e posição dos outros pedaços do arquivo de objeto
 - **Segmento de texto**: contém o código da linguagem de máquina
 - **Segmento de dados estático**: dados coletados para o ciclo de vida do programa
 - **Informação de realocação**: identifica instruções e palavras de dado que dependem dos endereços absolutos quando o programa é carregado na memória

Produzindo um Módulo de Objeto (2)

- **Tabela de símbolos:** contém os rótulos restantes que não estão definidos, tais como referências externas
- **Info de depuração:** contém uma descrição concisa de como os módulos foram compilados
 - Um depurador associa instruções de máquina com os arquivos fontes em C
 - Desta forma, pode transformar dados estruturados em algo legível

Vinculando Módulos de Objeto (1)

- Re-tradução completa do código é um desperdício de recursos computacionais
- Solução: compilar e montar cada procedimento independentemente
 - Mudança para uma linha requer compilar e montar somente um procedimento
- O vinculador produz uma imagem executável
 1. Coloca código e dados simbolicamente na memória
 2. Determina o endereço dos rótulos de instruções e dados
 3. Cria uma conexão entre as referências internas e externas

Vinculando Módulos de Objeto (2)

- O vinculador usa informações de relocação e a tabela de símbolos em cada módulo do objeto
 - Resolve rótulos não definidos
 - Ocorrem em instrução de desvio, salto e endereços de dados
- O vinculador encontra o endereço antigo e o substitui pelo novo endereço
- É mais rápido “remendar” o código do que re-compilar e re-montar
- O vinculador determina as localizações de memória que cada módulo ocupará
 - Se todas as referências externas estiverem resolvidas

Exemplo de Vinculação (1)

- Vincule os dois arquivos de objeto ao lado. Mostre os endereços atualizados das primeiras instruções do arquivo executável completo

A instruções devem ser números
 Texto em azul deve ser atualizado

Object file header			
	Name	Procedure A	
	Text size	100 _{hex}	
	Data size	20 _{hex}	
Text segment	Address	Instruction	
	0	lw \$a0, 0(\$gp)	
	4	jal 0	
	
Data segment	0	(X)	
	
Relocation information	Address	Instruction type	Dependency
	0	lw	X
	4	jal	B
Symbol table	Label	Address	
	X	-	
	B	-	
Object file header			
	Name	Procedure B	
	Text size	200 _{hex}	
	Data size	30 _{hex}	
Text segment	Address	Instruction	
	0	sw \$a1, 0(\$gp)	
	4	jal 0	
	
Data segment	0	(Y)	
	
Relocation information	Address	Instruction type	Dependency
	0	sw	Y
	4	jal	A
Symbol table	Label	Address	
	Y	-	
	A	-	

Exemplo de Vinculação (2)

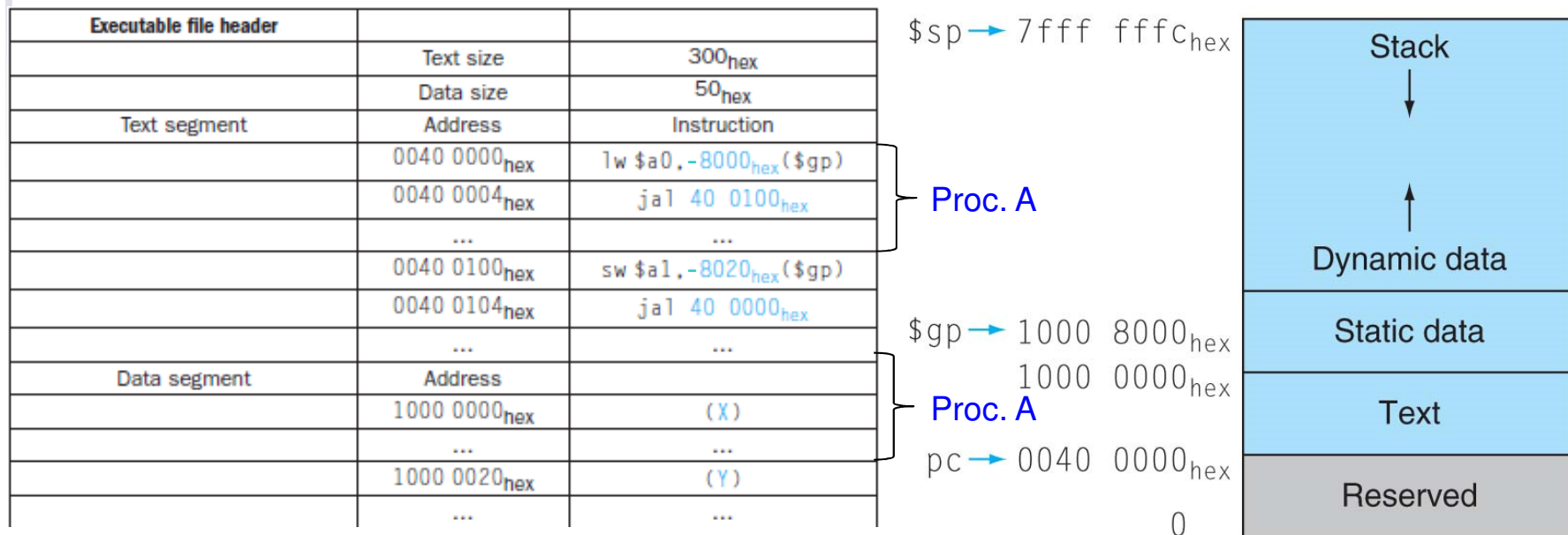
- Procedimento A precisa encontrar o endereço para a variável rotulada X
 - Para colocar na instrução lw
 - Encontrar o endereço do procedimento B para colocar na instrução jal
- Procedimento B precisa do endereço da variável rotulada Y
 - Para colocar na instrução sw
 - Encontrar o endereço do procedimento A para a instrução jal

Exemplo de Vinculação (3)

- Assumir que o segmento de texto inicia em $40\ 0000_{\text{hex}}$ e o seg. de dado em $1000\ 0000_{\text{hex}}$
- O texto do procedimento A é colocado no endereço $40\ 0000_{\text{hex}}$ e o dado em $1000\ 0000_{\text{hex}}$
- O header do proc. A menciona que o tamanho do texto é de 100_{hex} bytes e o dado é de 20_{hex} bytes
 - O endereço inicial para o texto do proc. B é $40\ 0100_{\text{hex}}$ e o dado inicia em $1000\ 0020_{\text{hex}}$

Exemplo de Vinculação (4)

- Ponteiro global inicia em $1000\ 8000_{\text{hex}}$
 - Para obter $1000\ 0000_{\text{hex}}$ (endereço de X), nós colocamos -8000_{hex} na instrução `lw`



$\$gp$: ponteiro global para dados estáticos (reg 28)

Carregando um Programa

- Carregar de um arquivo em disco para memória
 1. Lê o *header* para determinar o tamanho dos segmentos (texto e dados)
 2. Cria um espaço de endereço grande o suficiente para o texto e os dados
 3. Copia as instruções e dados para a memória
 4. Configura argumentos na pilha
 5. Inicializa registradores (incluindo *\$sp*, *\$fp*, *\$gp*)
 - Ajusta *\$sp* para a primeira posição livre
 6. Salta para rotina de inicialização
 - Copia argumentos para *\$a0*, ... e chama a *main*
 - Quando a *main* retorna, chama *exit syscall*

Vinculação Estática x Dinâmica

- Na vinculação **estática**, rotinas da **biblioteca** se tornam **parte do código** executável
 - Se uma nova versão da *lib* for liberada, o programa vinculado estaticamente mantém a versão anterior
 - Carrega todas as rotinas da biblioteca, mesmo se não forem usadas
- Vinculação **dinâmica** somente **vincula/carrega** o proc. da biblioteca **quando for chamado**
 - Requer código de procedimento a ser relocável
 - Evita o inchaço da imagem causado pela vinculação estática de todas as bibliotecas referenciadas
 - Capta automaticamente novas versões de bibliotecas

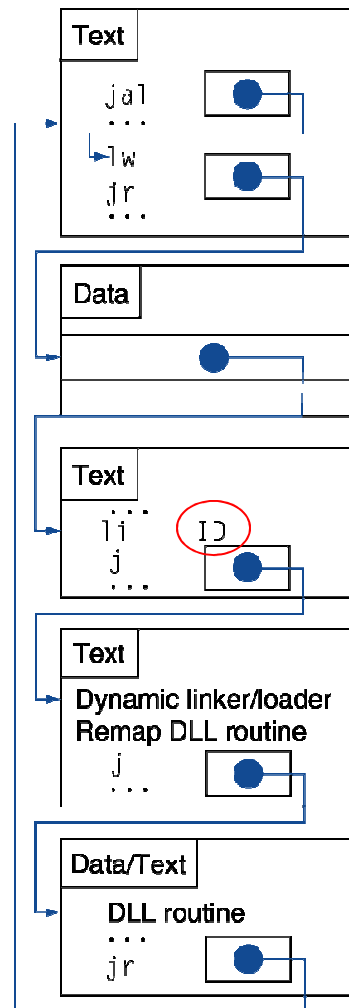
Vinculação Preguiçosa

Tabela de vias indiretas

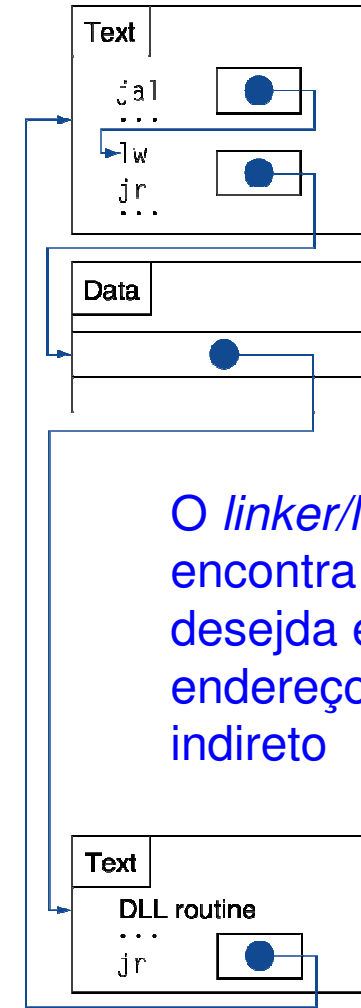
Stub: carrega ID da rotina,
Salta para *linker/loader*

código do *Linker/loader*

Código mapeado
dinamicamente



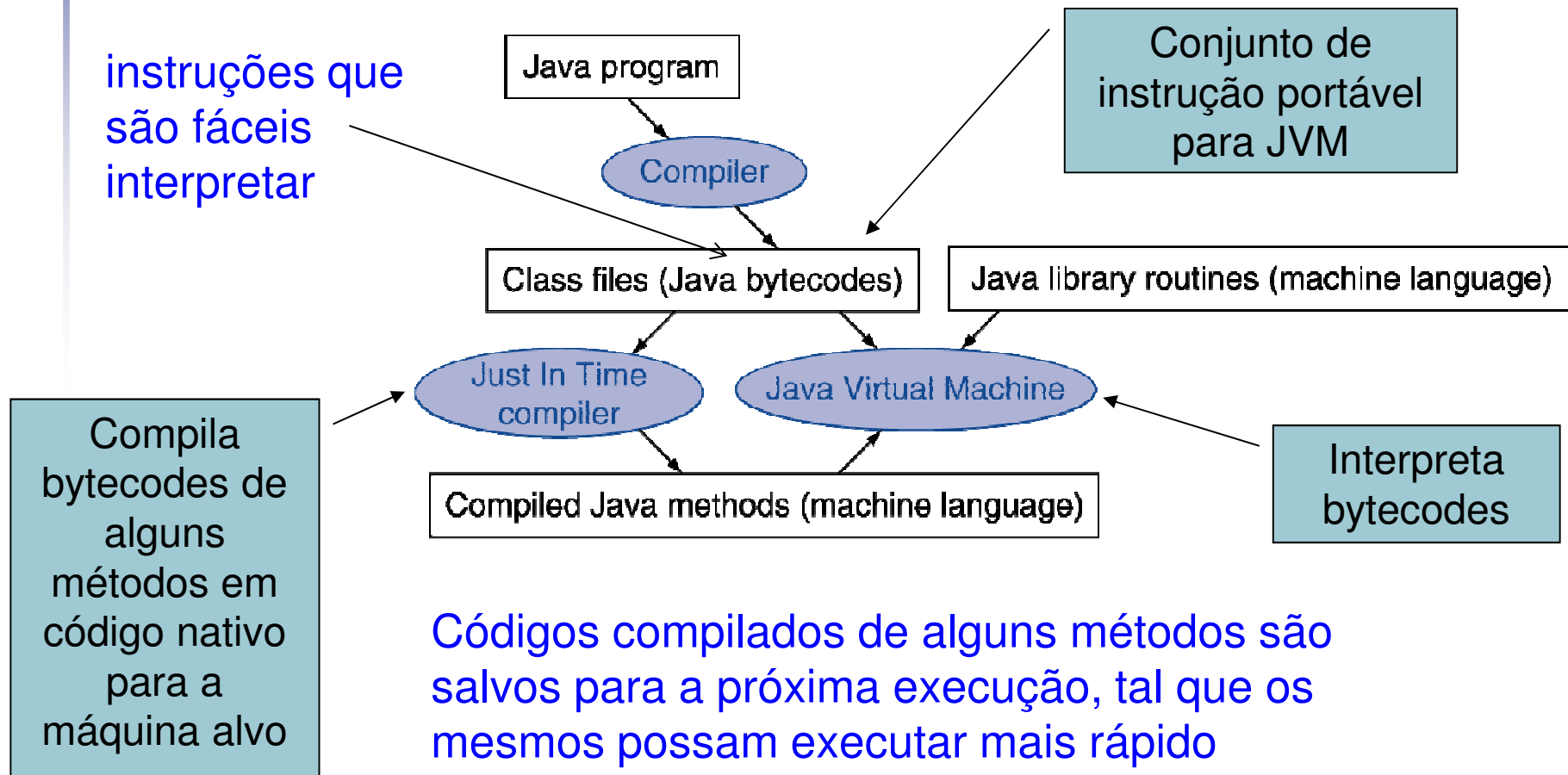
a. First call to DLL routine



b. Subsequent calls to DLL routine

O *linker/loader*
encontra a rotina
desejada e atualiza o
endereço do salto
indireto

Iniciando Aplicações Java



Reflexão

- Quais das vantagens, de um interpretador sobre um compilador, você acredita que foi mais importante para os projetistas do Java?
 1. Facilidade de escrever um interpretador
 2. Melhores mensagens de erro
 3. Menores códigos de objeto
 4. Independência da máquina

Exemplo de Ordenação em C

- Ilustrar o uso de instruções assembly para uma função de ordenação usando o método bolha em C
- Procedimento swap (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

O Procedimento *swap*

swap: sll \$t1, \$a1, 2	# \$t1 = k * 4
add \$t1, \$a0, \$t1	# \$t1 = v+(k*4)
	# (endereço 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	# voltar a rotina que chamou

O Procedimento de Ordenação em C

- Não folha (chama troca)

```
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, *n* em \$a1, *i* em \$s0 e *j* em \$s1

O Corpo do Procedimento

	move \$s2, \$a0	# salva \$a0 em \$s2	Move params
	move \$s3, \$a1	# salva \$a1 em \$s3	
for1tst:	move \$s0, \$zero	# i = 0	Outer loop
	slt \$t0, \$s0, \$s3	# \$t0 = 0 if \$s0 ≥ \$s3 (i ≥ n)	
	beq \$t0, \$zero, exit1	# vai p/ 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)	Inner loop
	bne \$t0, \$zero, exit2	# vai p/ 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	# vai p/ exit2 if \$t4 ≥ \$t3	
	move \$a0, \$s2	# 1st param de troca é v (old \$a0)	Pass params & call
	move \$a1, \$s1	# 2nd param de troca é j	
	jal swap	# chama procedimento swap	
	addi \$s1, \$s1, -1	# j -= 1	Inner loop
	j for2tst	# jump to test of inner loop	
exit2:	addi \$s0, \$s0, 1	# i += 1	Outer loop
	j for1tst	# jump to test of outer loop	

O Procedimento Completo

sort:	addi \$sp,\$sp, -20	# abrir espaço na pilha para 5 regs.
	sw \$ra, 16(\$sp)	# salva \$ra na pilha
	sw \$s3,12(\$sp)	# salva \$s3 na pilha
	sw \$s2, 8(\$sp)	# salva \$s2 na pilha
	sw \$s1, 4(\$sp)	# salva \$s1 na pilha
	sw \$s0, 0(\$sp)	# salva \$s0 na pilha
	...	# corpo do procedimento
	...	
	exit1: lw \$s0, 0(\$sp)	# restaura \$s0 da pilha
	lw \$s1, 4(\$sp)	# restaura \$s1 da pilha
	lw \$s2, 8(\$sp)	# restaura \$s2 da pilha
	lw \$s3,12(\$sp)	# restaura \$s3 da pilha
	lw \$ra,16(\$sp)	# restaura \$ra da pilha
	addi \$sp,\$sp, 20	# restaura ponteiro de pilha
	jr \$ra	# retorna para o procedimento que chamou

Exercício 1

- Vincule os dois diferentes procedimentos mostrados na tabela abaixo
 - Proc. A: tamanho de texto 0x140 e de dado 0x40
 - Proc. B: tamanho de texto 0x300 e de dado 0x50

a.	Procedure A				Procedure B			
	Text Segment	Address	Instruction		Text Segment	Address	Instruction	
		0	lbu \$a0, 0(\$gp)			0	sw \$a1, 0(\$gp)	
		4	jal 0			4	jal 0	
	Data Segment	0	(X)		Data Segment	0	(Y)	
		
	Relocation Info	Address	Instruction Type	Dependency	Relocation Info	Address	Instruction Type	Dependency
		0	lbu	X		0	sw	Y
		4	jal	B		4	jal	A
	Symbol Table	Address	Symbol		Symbol Table	Address	Symbol	
		—	X			—	Y	
		—	B			—	A	

Exercício 2

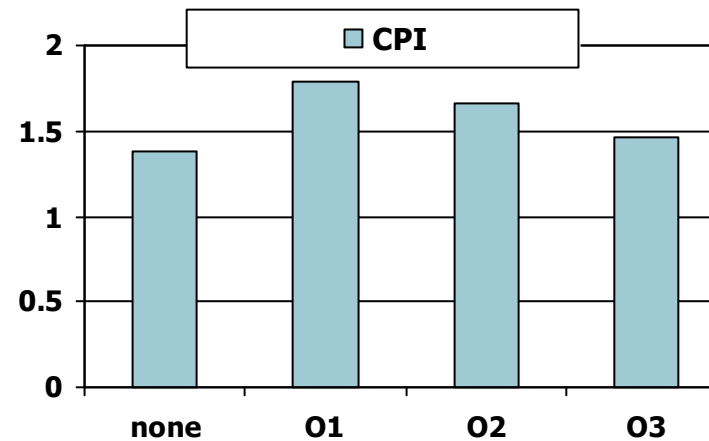
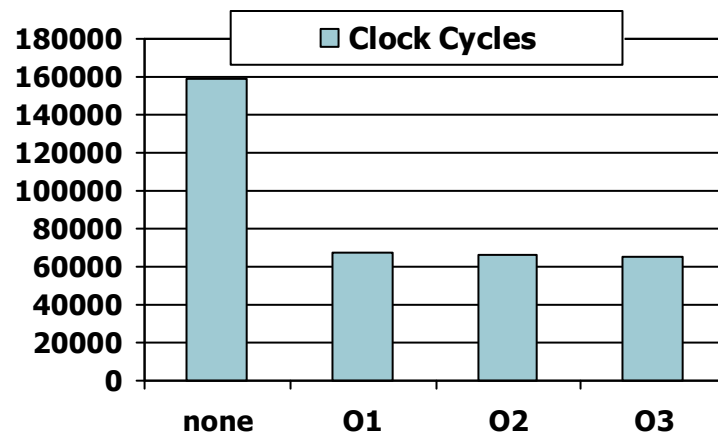
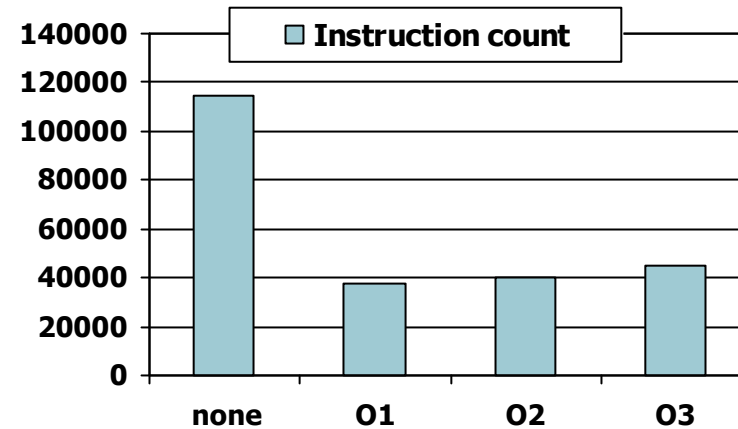
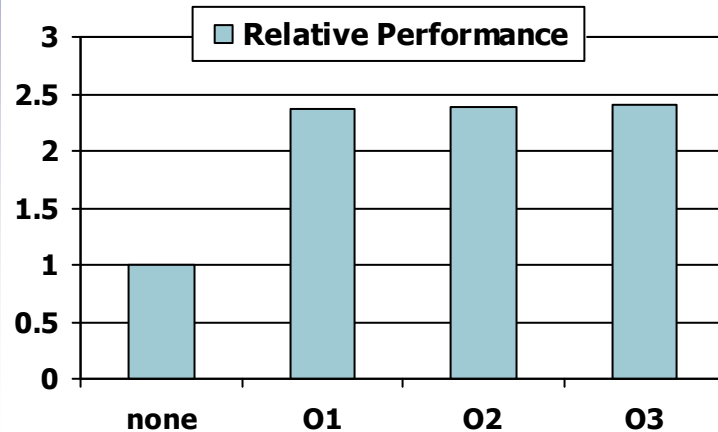
- Escreva o código MIPS para o seguinte algoritmo

```
void selectionSort() {  
    int i, j, temp, min;  
    for (j = 0; j < 2; j++) {  
        min = j;  
        for( i = j+1; i < 3 ; i++)  
            if( array[i] < array[ min])  
                min = i;  
        temp = array[ j];  
        array[ j] = array[ min];  
        array[ min] = temp;  
    }  
}
```

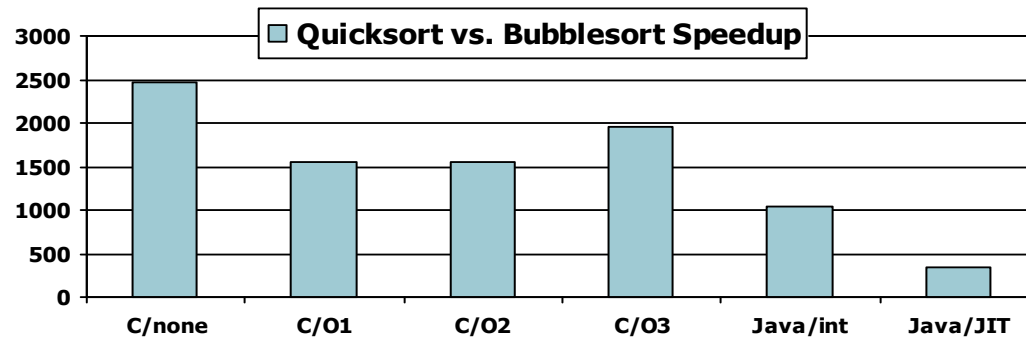
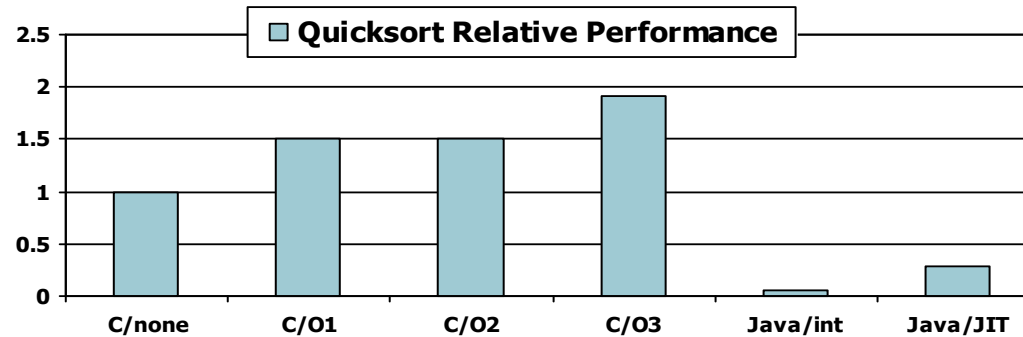
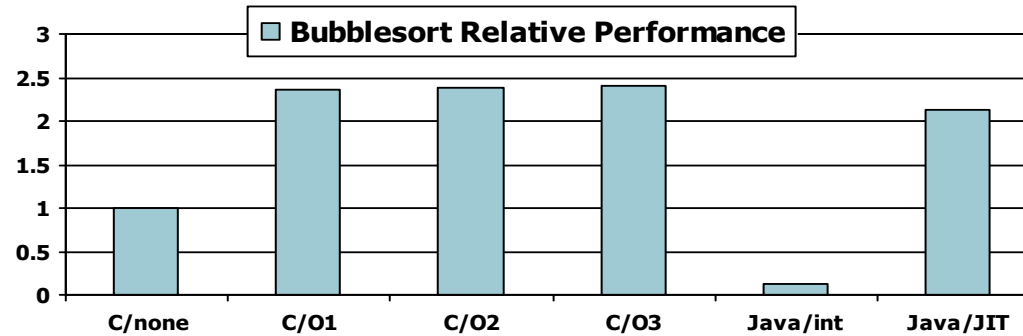
- array em \$a0, i em \$s0, j em \$s1, temp em \$s2, min em \$s3

Efeito de Otimização de Compilador

Compilado com gcc para Pentium 4 usando Linux



Efeito da Linguagem e do Algoritmo



Comparação de Desempenho

■ Níveis de otimização do GCC

gcc optimization	Relative performance	Clock cycles (millions)	Instruction count (millions)	CPI
None	1.00	158,615	114,938	1.38
O1 (medium)	2.37	66,990	37,470	1.79
O2 (full)	2.38	66,521	39,993	1.66
O3 (procedure integration)	2.41	65,747	44,993	1.46

■ Comparação entre GCC e Java/JIT

Language	Execution method	Optimization	Bubble Sort relative performance	Quicksort relative performance	Speedup Quicksort vs. Bubble Sort
C	Compiler	None	1.00	1.00	2468
	Compiler	O1	2.37	1.50	1562
	Compiler	O2	2.38	1.50	1555
	Compiler	O3	2.41	1.91	1955
Java	Interpreter	–	0.12	0.05	1050
	JIT compiler	–	2.13	0.29	338

Lições Aprendidas

- Contagem de instrução e CPI não são bons indicadores de desempenho em isolamento
- As otimizações do compilador são sensíveis ao algoritmo
- Código compilado Java/JIT é significativamente mais rápido do que o JVM interpretado
 - Comparável a código C otimizado em alguns casos
- Pouco pode ser feito para corrigir um algoritmo ineficiente!

Arranjos vs. Ponteiros

- Indexação de arranjos envolve
 - Multiplicar o index pelo tamanho do elemento
 - Adicionar ao endereço base do arranjo
- Ponteiros correspondem diretamente aos endereços de memória
 - Pode evitar a complexidade da indexação

Exemplo: Limpando um Arranjo

```
clear1(int array[], int size) {  
    int i;  
    for (i = 0; i < size; i += 1)  
        array[i] = 0;  
}
```

```
        move $t0,$zero    # i = 0  
loop1: sll $t1,$t0,2      # $t1 = i * 4  
        add $t2,$a0,$t1  # $t2 =  
                        # &array[i]  
        sw $zero, 0($t2) # array[i] = 0  
        addi $t0,$t0,1   # i = i + 1  
        slt $t3,$t0,$a1  # $t3 =  
                        # (i < size)  
        bne $t3,$zero,loop1 # if (...)  
                        # goto loop1
```

```
clear2(int *array, int size) {  
    int *p;  
    for (p = &array[0]; p < &array[size];  
        p = p + 1)  
        *p = 0;  
}
```

```
        move $t0,$a0      # p = & array[0]  
        sll $t1,$a1,2      # $t1 = size * 4  
        add $t2,$a0,$t1  # $t2 =  
                        # &array[size]  
loop2: sw $zero,0($t0) # Memory[p] = 0  
        addi $t0,$t0,4     # p = p + 4  
        slt $t3,$t0,$t2  # $t3 =  
                        # (p < &array[size])  
        bne $t3,$zero,loop2 # if (...)  
                        # goto loop2
```

Assumir *array* em *\$a0* e *size* em *\$a1*



Comparação de Arranjo vs. Ptr

- Multiplicar “com custo reduzido” usando deslocamento (*strength reduction*)
- A versão do arranjo requer deslocamento dentro do laço
 - Para calcular o índice através do incremento de i
 - c.f. incrementando ponteiro (reduz de 6 para 4 instr.)
- Compilador pode alcançar o mesmo efeito como o uso manual de ponteiros
 - Eliminação da variável de indução
 - Eliminando o cálculo do endereço do arranjo dentro do laço
 - Melhor fazer programa mais claro e mais seguro

Similaridades do ARM & MIPS

- ARM: o núcleo embarcado mais popular
- Conjunto básico de instruções similar ao MIPS

	ARM	MIPS
Data anunciada	1985	1985
Tamanho da instrução	32 bits	32 bits
Espaço de endereço	32-bit	32-bit
Alinhamento de dados	Alinhado-4 bytes	Alinhado-4 bytes
Modos de end. de dados	9	3
Registradores	15 × 32-bit	31 × 32-bit
Entrada/saída	Memória mapeada	Memória mapeada

Modos de Endereçamento

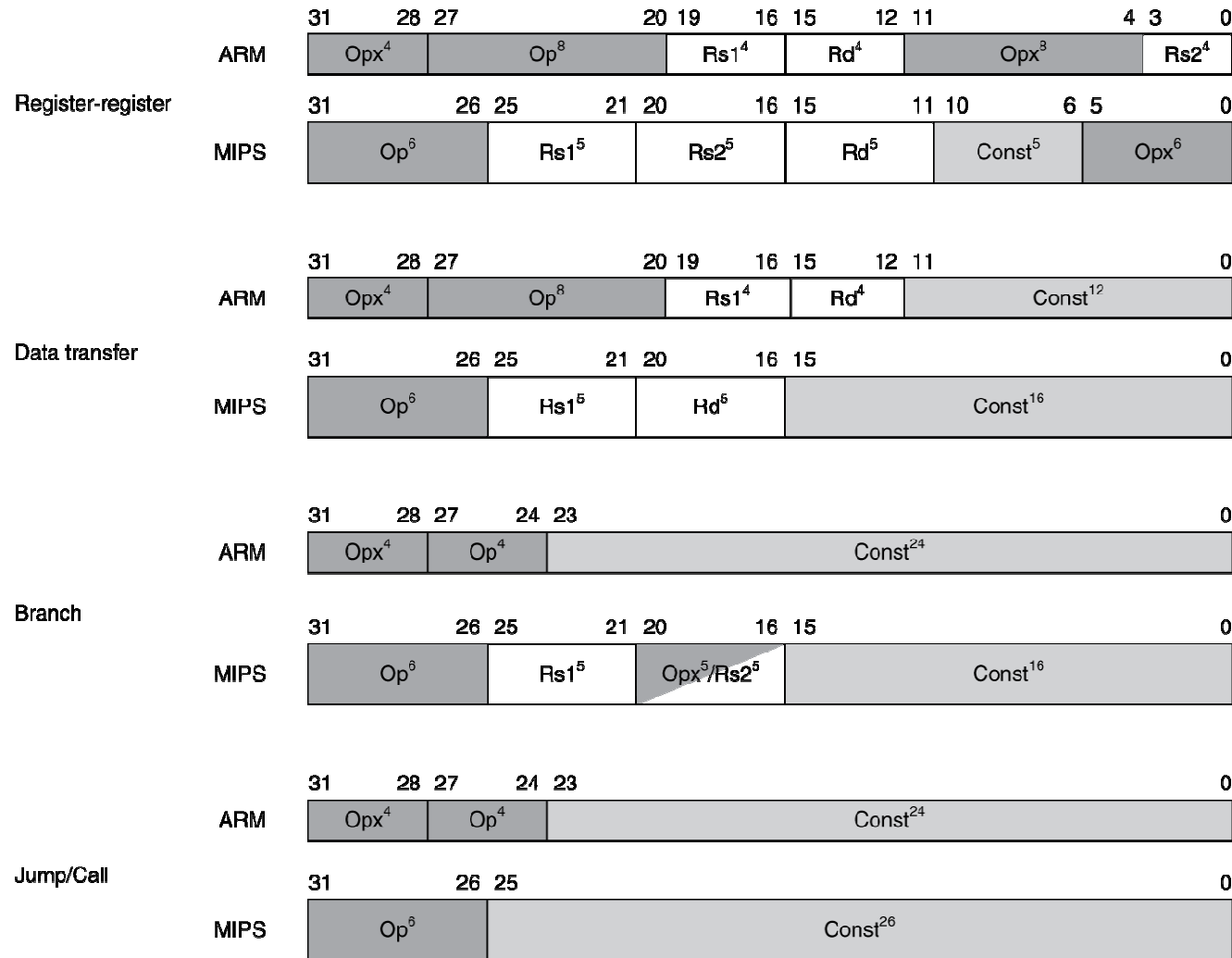
- ARM tem nove modos de endereçamento
 - ARM não reserva um registrador para o conteúdo 0

Addressing mode	ARM v.4	MIPS
Register operand	X	X
Immediate operand	X	X
Register + offset (displacement or based)	X	X
Register + register (indexed)	X	—
Register + scaled register (scaled)	X	—
Register + offset and update register	X	—
Register + register and update register	X	—
Autoincrement, autodecrement	X	—
PC-relative data	X	—

Comparação e Desvio no ARM

- MIPS usa os conteúdos dos registradores para avaliar os desvios condicionais
- Usa os 4 bits dos códigos de condição armazenado na palavra de status do programa
 - *Negative, zero, carry, overflow*
 - *Definido em qualquer instrução aritmética ou lógica*
- Cada instrução pode ser condicional
 - Top 4 bits da palavra de instrução: valor de condição
 - Determina se atuará como *nop* (*no operation*) ou como uma instrução real
 - Deste modo, pode evitar desvios sob instruções individuais

Codificação da Instrução



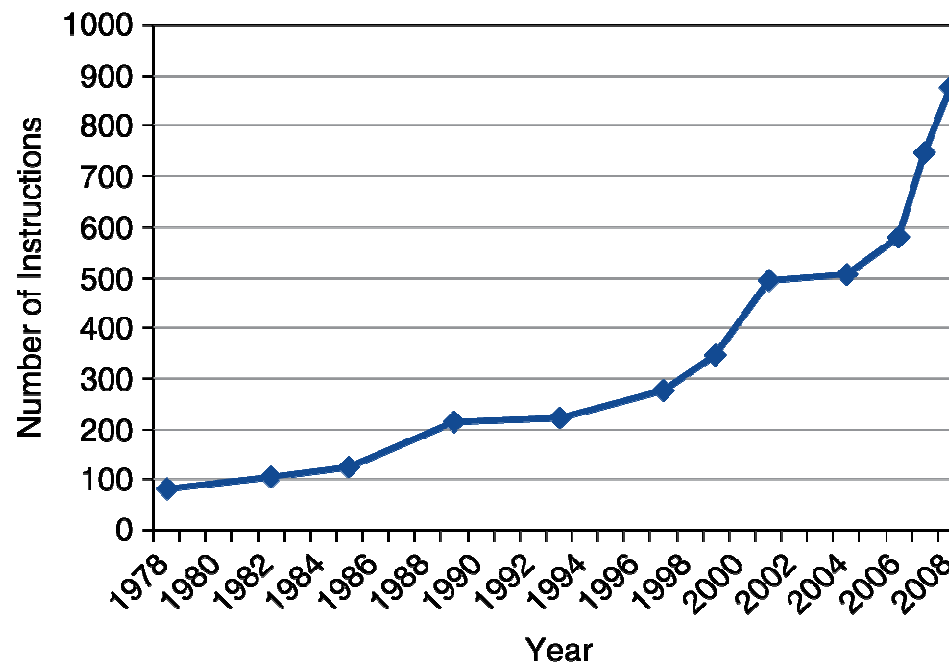
☐ Opcode
 ☐ Register
 ☐ Constant

Falácias (1)

- Instrução poderosa \Rightarrow desempenho mais alto
 - Menos instruções necessárias
 - Mas instruções complexas são difíceis de implementar
 - Pode desacelerar todas as instruções, incluindo as simples
 - Compiladores são bons em fazer código rápido a partir de instruções simples
- Use código *assembly* para alto desempenho
 - Mas compiladores modernos são melhores em lidar com processadores modernos
 - Mais linhas de código \Rightarrow mais erros e menos produtividade

Falácias (2)

- Compatibilidade com versões anteriores \Rightarrow conjunto de instrução não muda
 - Mas eles agregam mais instruções



Conjunto de
instrução do x86

Armadilhas

- Palavras sequenciais não estão em endereços sequenciais
 - Incremento de 4, não de 1!
- Mantendo um ponteiro para uma variável automática após retorno de procedimento
 - e.g., passando o ponteiro de volta através de um argumento
 - Ponteiro se torna inválido quando a pilha for esvaziada

Observações Finais (1)

- Princípios de Projeto
 1. Simplicidade favorece regularidade
 2. Menor é mais rápido
 3. Faça o caso comum rápido
 4. Um bom projeto exige bons compromissos
- Camadas de software/hardware
 - Compilador, montador, hardware
- MIPS: típico do RISC ISAs
 - Instruções simplificadas podem fornecer melhor desempenho
 - se esta simplicidade fornecer execução mais rápida de cada instrução (c.f. x86)

Observações Finais (2)

- Medir as execuções da instrução do MIPS em programas *benchmark*
 - Considere realizar o caso comum rápido
 - Considere compromissos

Classe de instrução	Exemplos do MIPS	SPEC2006 Int	SPEC2006 FP
Aritmética	add, sub, addi	16%	48%
Transferência de dados	lw, sw, lb, lbu, lh, lhu, sb, lui	35%	36%
Lógico	and, or, nor, andi, ori, sll, srl	12%	4%
Desvio condicional	beq, bne, slt, slti, sltiu	34%	8%
Salto	j, jr, jal	2%	0%



Exercício

- Comparar o tempos de execução entre os compiladores do C e do Java, utilizando otimizações (no caso do Java usando apenas a JVM e JVM/JIT)
 - Comparar a quantidade de instruções necessárias para execução dos algoritmos de ordenação por inserção e ordenação por intercalação
 - Compilar o código C com o seguinte comando:
 - `gcc <otimização> -o <arquivodeexecução> <arquivo.c>`
 - Comando de compilação do Java:
 - `javac <arquivo.java>`
 - Comando utilizado para execução sem JIT:
 - `java -Xint <arquivo>`