Aula 2: Instruções - A linguagem de máquina - Parte III

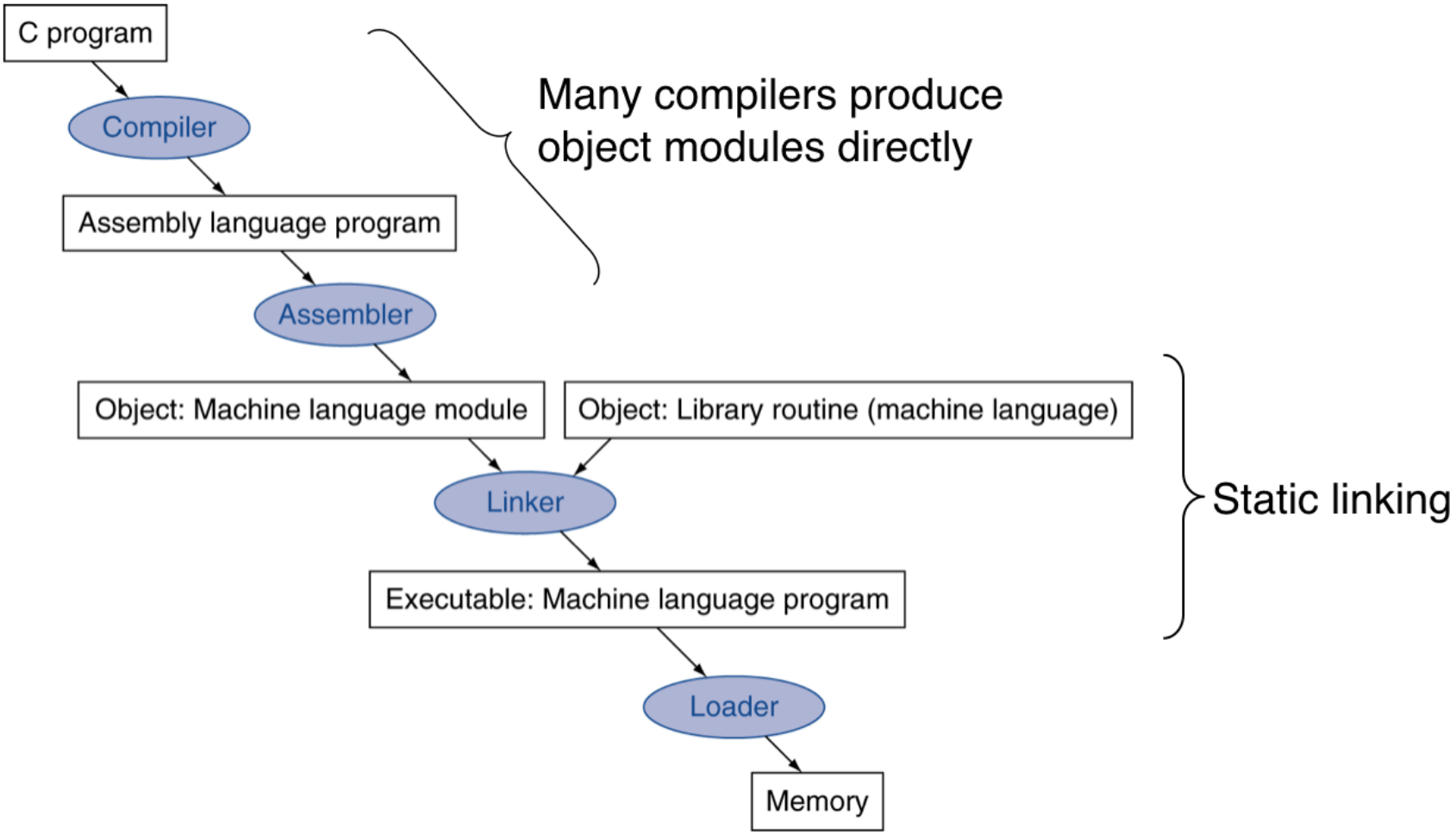
Professor(a): Virgínia Fernandes Mota
http://www.dcc.ufmg.br/∼virginiaferm

OCS (TEORIA) - SETOR DE INFORMÁTICA

Instruções - A linguagem de máquina

- Traduzindo e iniciando um programa
- Como os compiladores otimizam
- Um exemplo de ordenação em C para juntar tudo isso
- Array vs Ponteiros

Traduzindo e iniciando um programa



Traduzindo e iniciando um programa

- **Compilador**: Transforma código em linguagem de alto nível, como C, em um programa assembly
  - Assembly: linguagem simbólica, que pode ser traduzida para formato binário
- **Montador (Assembler)**: Converte instrução em assembly para o equivalente em linguagem de máquina
  - Gera arquivo objeto
  - Instrução em assembly não precisa ser necessariamente implementada pelo hardware
    - Montador pode converte-las para a(s) instrução(ões) real
    - Pseudo-instruções

Traduzindo e iniciando um programa

- Mais sobre o **Montador (Assembler)**!
- Exemplo: instrução move $t0, $t1 convertida para add $t0, $zero, $t1
- Montador também converte números em outras bases, como decimal, para binário
- Montador também produz tabela de símbolos
  - Combina nome de rótulos com endereços das words na memória
  - Usada para determinar endereços correspondentes a todos os rótulos

Traduzindo e iniciando um programa

- Mais sobre o **Montador (Assembler)**!
- Arquivo objeto em UNIX com formato bem definido:
  - Cabeçalho descreve tamanho e posição das outras partes do arquivo objeto
  - Segmento de texto contém código na linguagem de máquina
  - Segmento de dados estáticos contém os dados alocados por toda a vida do programa
  - Informações de relocação identificam instruções e words de dados que dependem de endereços absolutos quando o programa é carregado na memória
  - Tabela de símbolos contém rótulos restantes que não estão definidos (p.ex., referências externas)
  - Informações de depuração, que contêm dados que permitem a depuração de programas

Traduzindo e iniciando um programa

- **Link-editor**: Combina programas em linguagem de máquina montados de maneira independente, gerando código executável
- Cada módulo compilado de modo independente
  - Mudança em um módulo não exige recompilação dos demais módulos
- Informações de relocação e tabela de símbolos utilizados para resolver rótulos indefinidos
  - Se todas as referências resolvidas, link-editor determina locais da memória que cada módulo ocupará

Traduzindo e iniciando um programa

- **Loader**: Programa que coloca o programa na memória principal, de forma que ele possa ser executado. Passos:
  - Lê cabeçalho do executável para determinar tamanho dos segmentos de texto e dados
  - Cria espaço de endereçamento grande o suficiente para texto e dados
  - Copia instruções e dados do executável para memória
  - Copia parâmetros (se houver) do programa principal para a pilha
  - Inicia registradores da máquina e define stack pointer para primeiro local livre
  - Desvia para rotina de partida, que copia parâmetros para registradores de argumento e chama rotina principal
    - Quando esta retorna, rotina de partida termina programa com chamada a exit

Traduzindo e iniciando um programa

- Técnicas tradicionais descritas anteriormente possuem desvantagens!!
- Rotinas de bibliotecas tornam-se parte do executável
  - Se novas versões da biblioteca forem lançadas, programa link-editado estaticamente continua usando versão antiga
- Toda a biblioteca combinada com o executável, mesmo que apenas parte dela seja utilizada
- Código da biblioteca não pode ser compartilhado, mesmo que dois programas em execução utilizem a mesma biblioteca

Traduzindo e iniciando um programa

- Desvantagens levaram às **DLLs (Dinamically Linked Libraries)**!
- Em uma primeira versão, rotinas de biblioteca não são link-editadas e carregadas até que programa seja executado
- Segunda versão faz link-edição tardia de procedimento
- Rotina só é link-editada depois de chamada

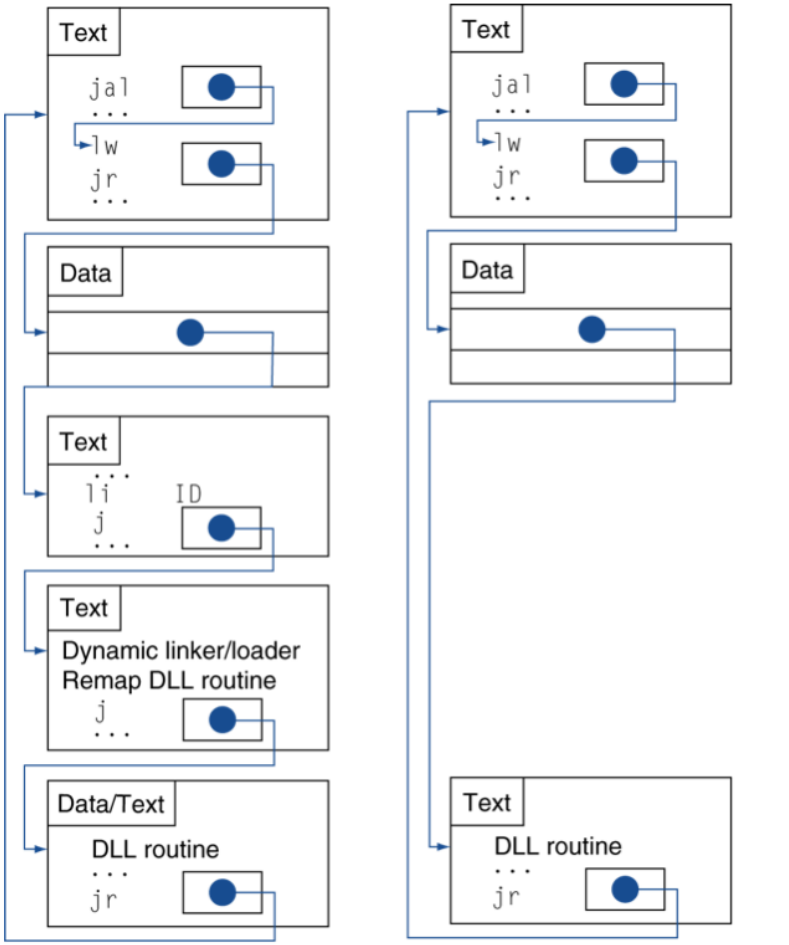
Traduzindo e iniciando um programa

Indirection table

Stub: Loads routine ID, Jump to linker/loader

Linker/loader code

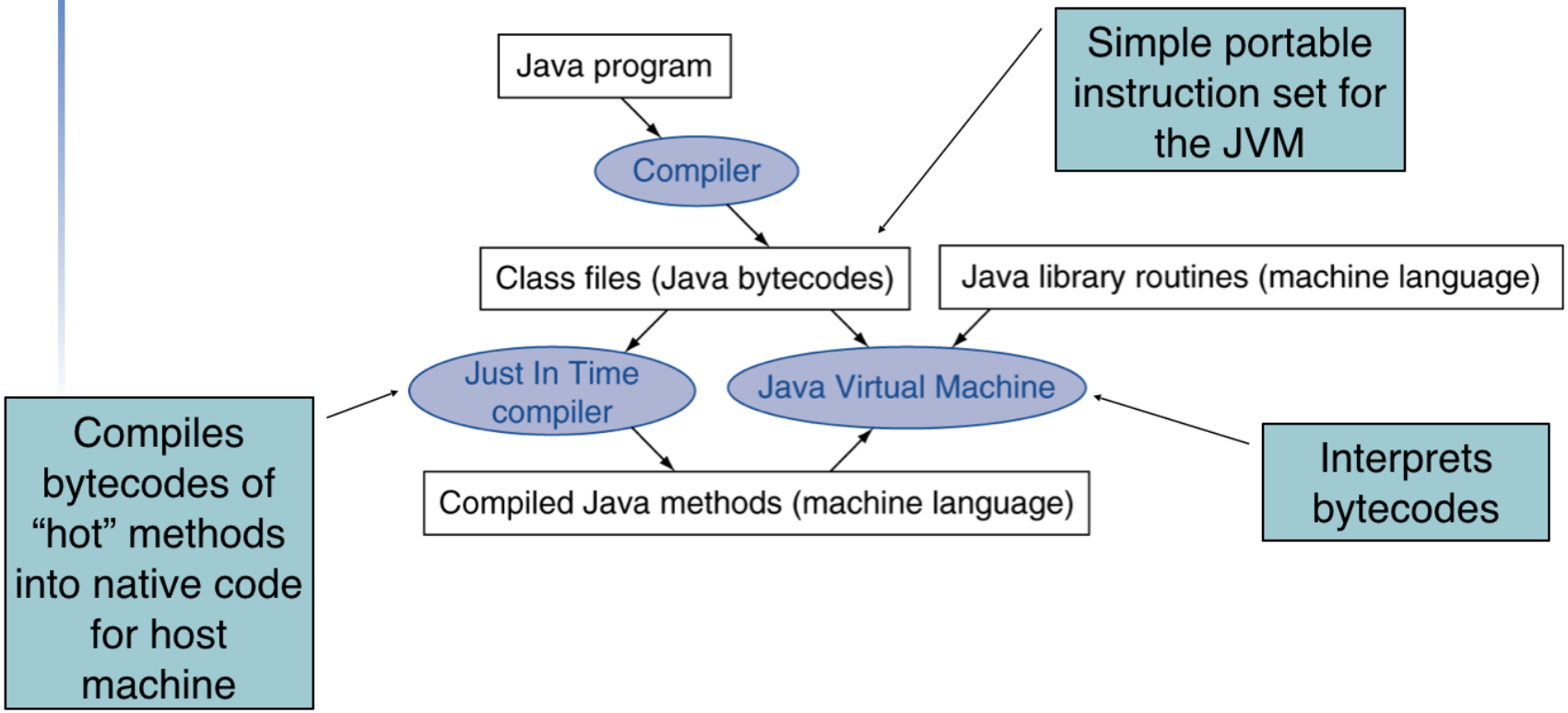
Dynamically mapped code



a. First call to DLL routine

b. Subsequent calls to DLL routine

Traduzindo e iniciando um programa



Como os compiladores otimizam

Dependências		***Função***
Dependente de linguagem; independente de máquina	Front-end por Linguagem	Transformar linguagem para forma intermediária comum
Um tanto dependente da linguagem; bastante independente da máquina	Otimizações de Alto Nível	Por exemplo, transformações de loop e procedimentos inline.
Pequena dependência da linguagem; ligeiras dependências da máquina (por exemplo, número / tipos de registrador)	Otimizador Global	Incluindo otimizações globais e locais + alocações de registradores.
Altamente dependente da máquina; independente de linguagem	Gerador de Código	Seleção detalhada de instrução e otimizações dependentes de máquina; pode incluir ou ser acompanhada de um montador

Como os compiladores otimizam

- Compilador é conservador!
- Primeira tarefa: produzir código correto
- Segunda tarefa: produzir código rápido
- Outros fatores também podem ser importantes: Tamanho do código

Vamos juntar tudo!

Um Exemplo de Ordenação em C para Juntar Tudo Isso!!

Exemplo: Ordenação

- Procedimento swap: Troca conteúdo de duas posições de memória.

```
1 void swap (int v[], int k) {
2     int temp = v[k];
3     v[k] = v[k+1];
4     v[k+1] = temp;
5 }
```

Exemplo: Ordenação

- Etapas gerais para traduzir de C para assembly:
  - Alocar registradores a variáveis do programa
  - Produzir código para o corpo do procedimento
  - Preservar registradores durante a chamada de procedimento

Exemplo: Ordenação

- Passo 1: Alocar registradores a variáveis do programa
  - Registradores $a0 e $a1 usados para passagem dos parâmetros
  - Variável temp associada com registrador $t0
  - Por ser um procedimento folha

Exemplo: Ordenação

- Passo 2: Produzir código para o corpo do procedimento
  - Erro comum em programação assembly: esquecer que endereços de words sequenciais diferem em 4 bytes
  - Temos de multiplicar índice k por 4
    sll $t1, $a1, 2 # registrador t1 = k * 4
    add $t1, $a0, $t1 # registrador t1 = v + (k * 4)

Exemplo: Ordenação

- Passo 2: Continuando
  - Levamos agora v[k] para t0
    lw $t0, 0 ($t1)
  - Em seguida lemos v[k+1] em t2
    lw $t2, 4 ($t1)
  - Agora armazenamos dados lidos nos endereços trocados
    sw $t2, 0 ($t1)
    sw $t0, 4 ($t1)

Exemplo: Ordenação

- Passo 3: Preservar registradores durante a chamada de procedimento
- Desnecessário, pois não chamamos procedimentos

Exemplo: Ordenação

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

Exemplo: Ordenação

- Procedimento sort: Ordena array de inteiros, usando ordenação por trocas.

```
1 void sort (int v[], int n) {
2     int i, j;
3     for (i=0; i < n; i++)
4         for (j = i-1 ; j > 0 && v[j] > v[j+1] ; j -= 1)
5             swap (v,j);
6 }
```

Exemplo: Ordenação

- Passo 1: Alocar registradores a variáveis do programa
  - Registradores $a0 e $a1 usados para passagem dos parâmetros
  - Registradores precisam ser alocados para i e j: Por exemplo, $s0 a i e $s1 a j

Exemplo: Ordenação

- Passo 2: Produzir código para o corpo do procedimento
  - Loop for composto por 3 partes:
    - Inicialização
    - Teste de loop
    - Incremento da iteração
  - Primeiro loop:

```
1                for (i=0; i < n; i++)
```

    - Inicialização:
      move $s0, $zero # i = 0 com uso de pseudoinstrução
    - Incremento da iteração
      addi $s0, $s0, 1

Exemplo: Ordenação

- Segundo teste termina se v[j] $\leq$ v [j+1]
  - Primeiro passo: calcular endereço de memória
    ssl $t1, $s1, 2 # $t1 = j * 4
    add $t2, $a0, $t1 # $ t2 = v + (j * 4)?
  - Agora lemos conteúdo de v[j] e v[j+1]
    lw $t3, 0 ($t2) # t3 = v[j]
    lw $t4, 4($t2) # t4 = v [j+1]
  - Teste
    slt $t0, $t4, $t3 # $t0=0 se $t4 >= $t3
    beq $t0, $zero, exit2 # desvia para exit se $t4>=$t3

Exemplo: Ordenação

- Passo 2: Continuando
  - Teste do loop: término quando i $\geq$ n

    for1tst:

    ```
        slt $t0, $s0, $a1
        beq $t0, $zero, exit 1
    ```
  - Final do loop só desvia de volta para o teste do loop:

    ```
    j for1tst
    exit1:
    ```

Exemplo: Ordenação

- Segundo for se parece com o primeiro

```
1     for (j = i-1 ; j > 0 && v[j] > v[j+1] ; j -= 1)
```

- Inicialização do loop:
  addi $s1, $s0, -1
- Decremento de j no final do loop:
  addi $s1, $s1, -1
- Teste do loop possui duas partes:
- Primeira condição j < 0
  for2tst:
  slti $t0, $s1, 0 # $t0 = 1 se j < 0
  bne $t0, $zero, exit2 # vai para exit2 se j < 0

Exemplo: Ordenação

- Próxima etapa: chamada do procedimento swap (v,j);
  jal swap
- Precisamos passar os parâmetros para swap
- Problema: sort também precisa dos parâmetros
- Precisamos então salvar os registradores
- Como temos registradores sobrando, podemos utiliza-los para salvar o estado, ao invés de usar a pilha que é mais lenta.

Exemplo: Ordenação

- Podemos usar $s2 e $s3 para este fim:
  move $s2, $a0
  move $s3, $a1
- Passamos os parâmetros para swap com:
  move $a0, $s2
  move $a1, $s1
- Passo final: preservar e restaurar registradores utilizados
- Usamos cinco registradores: $s0 a $s3 e $ra

Exemplo: Ordenação - Completo

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

Move params

Outer loop

Inner loop

Pass params & call

Inner loop

Outer loop

Exemplo: Ordenação - Completo

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

Lição do dia

O código influencia diretamente no desempenho! E nada pode consertar um algoritmo ruim, por mais que o compilador tente!

Array vs Ponteiros

- Vamos ver como ponteiros são mapeados em instruções MIPS
  - Comparando com uma versão que usa arrays
- Nosso exemplo: Procedimento para zerar uma seqüência de palavras na memória

Array vs Ponteiros

```
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
```

Array vs Ponteiros

- Dois códigos consideram que size maior do que zero
- Comparando as duas versões
  - Versão usando array tem multiplicação e soma dentro do loop
    - Variável i incrementada e cada endereço precisa ser recalculado a partir do novo índice
    - Versão com ponteiros incrementa ponteiro diretamente
  - Versão com ponteiros reduz instruções dentro do loop de 7 para 4

Para saber mais...

LEIAM!

Vida Real: Instruções do IA-32

Arquiteturas RISC *vs* Arquiteturas CISC

Falácias e Armadilhas

- Falácia: Instruções mais poderosas significam maior desempenho
  - Instrução com prefixo
    - Repete instrução até que contador chegue a zero
  - Instrução que carrega dados em registradores para então move-los para a memória de 1,5 a 2,0 mais rápido que instrução move usando prefixo
- Falácia: Escreva em assembly para obter maior desempenho
  - Compiladores cada vez mais sofisticados
  - Tempo maior gasto escrevendo e depurando código

Falácias e Armadilhas

- Armadilha: Esquecer que endereços sequenciais de words em máquinas com endereçamento em bytes não diferem em um
  - Incrementa-se o tamanho da word em bytes
- Armadilha: usando um ponteiro para uma variável automática fora de seu procedimento de definição
  - Por exemplo, passando resultado de um procedimento que inclui um ponteiro para um array que é local a esse procedimento
  - Array na pilha: logo memória será reutilizada assim que procedimento terminar

Conclusão

- Princípios de projeto:
  1. a simplicidade favorece a regularidade
  2. menor significa mais rápido
  3. agilize os casos mais comuns
  4. um bom projeto exige bons compromissos
- Camadas software/hardware: compilador, assembler, hardware

Conclusão

Instruction class	MIPS examples	SPEC2006 Int	SPEC2006 FP
Arithmetic	add, sub, addi	16%	48%
Data transfer	lw, sw, lb, lbu, lh, lhu, sb, lui	35%	36%
Logical	and, or, nor, andi, ori, sll, srl	12%	4%
Cond. Branch	beq, bne, slt, slti, sltiu	34%	8%
Jump	j, jr, jal	2%	0%

Figura: Benchmarks do MIPS32

Próxima aula

Exercícios e Prova