Do alto-nível ao *assembly*

Compiladores, Aula Nº 3

João M. P. Cardoso

Alinhamento, empacotamento e enchimento

- Requisitos de alinhamento:
  - Inteiros tipo **int** (4 bytes) a começar em endereços com os 2 LSBs == “00”
  - Inteiros tipo **short** (2 bytes) a começar em endereços com o LSB == ‘0’
- Alinhamento requer:
  - Enchimento entre campos para assegurar o alinhamento
  - Empacotamento de campos para assegurar a utilização de memória

Alinhamento

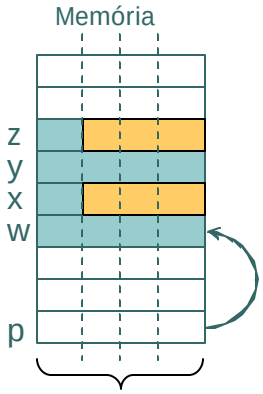
```
typedef struct {
  int w;
  char x;
  int y;
  char z;
} foo;

foo *p;
```

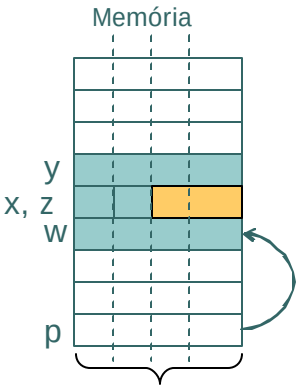
livre

ocupado

Organização "ingénua"



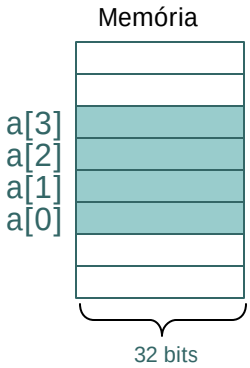
Organização Empacotada (poupa 4 bytes)



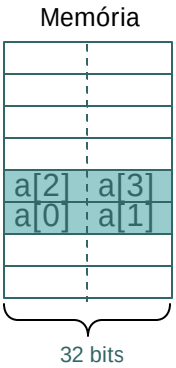
Arrays

- Afectação de posições de memória para os elementos do array
- Elementos são armazenados contiguamente

int a[4];



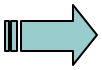
short a[4];



Arrays

- Utilizando registos do processador para armazenar as variáveis i e j:

```
int a[4];
proc() {
  int i, j;
  ...
  i = a[j];
  ...
}
```



```
        .data
A:      .space 16
        .text
Proc:
        ...
        la      $t0, A
        addi    $t2, $0, 4
        mult    $t1, $t2
        mflo    $t2
        add     $t3, $t2, $t0
        lw      $t4, 0($t3)
```

Endereço de a[j] = endereço de a[0] + (4 × j) = a + (4 × j)

Expressões

- a = b * c + d – e;
- a em $t4; b em $t0; c em $t1; d em $t2; e em $t3

```
...
mult    $t0, $t1
mflo    $t4
sub     $t5, $t2, $t3
add     $t4, $t4, $t5
...
```

```
...
mult    $t0, $t1
mflo    $t4
add     $t4, $t4, $t2
sub     $t4, $t4, $t3
...
```

Estruturas condicionais

If(a == 1) b = 2;

- a em $t0; b em $t1

```
        ...
        addi    $t2, $0, 1
        bne     $t2, $t0, skip_if
        addi    $t1, $0, 2
Skip_if: ...
```

If(a == 1) b = 2;
else b = 1;

- a em $t0; b em $t1

```
        ...
        addi    $t2, $0, 1
        bne     $t2, $t0, else
        addi    $t1, $0, 2
        j       skip_if
Else:   addi    $t1, $0, 1
Skip_if: ...
```


Estruturas condicionais

- *Branch-delay*
  - O processador executa sempre a instrução a seguir a uma instrução de salto (quer o salto seja realizado ou não)
  - Quando não é possível deslocar uma instrução para depois da instrução de salto, tem de se introduzir uma instrução **nop**

If(a == 1) b = 2;
C = a+1;

- a em $t0; b em $t1

```
        ...
        addi    $t2, $0, 1
        bne     $t2, $t0, skip_if
        addi    $t3, $t0, 1
        addi    $t1, $0, 2
Skip_if: ...
```

Ciclos

- Transformar o fluxo de controlo (while, for, do while, etc.) em saltos

```
Int sum(int A[], int N) {
  Int i, sum = 0;
  For(i=0; i<N; i++) {
    sum = sum + A[i];
  }
  return sum;
}
```

```
# $a0 armazena o endereço de A[0]
# $a1 armazena o valor de N
Sum:    Addi    $t0, $0, 0          # i = 0
        Addi    $v0, $0, 0          # sum = 0
Loop:   beq     $t0, $a1, End       # if(i == 10) goto End;
        Add     $t1, $t0, $t0       # 2*i
        Add     $t1, $t1, $t1       # 2*(2*i) = 4*i
        Add     $t1, $t1, $a0       # 4*i + base(A)
        Lw      $t2, 0($t1)         # load A[i]
        Add     $v0, $v0, $t2       # sum = sum + A[i]
        Addi    $t0, $t0, 1         # i++
        J       Loop                # goto Loop;
End:    jr      $ra                 # return
```


Ciclos

- Optimizações
  - Manter i e endereço de a[i] em registos
  - Determinar endereço de a[0] antes do corpo do ciclo, e incrementar de 4 (no caso de serem acessos a palavras de 32 bits) no corpo do ciclo
  - Caso o ciclo execute pelo menos uma iteração (N > 0) passar salto do ciclo para o fim do corpo

Ciclos

- Código após as optimizações

```
Int sum(int A[], int N) {
  Int i, sum = 0;
  For(i=0; i<N; i++) {
    sum = sum + A[i];
  }
  return sum;
}
```

```
# $a0 armazena o endereço de A[0]
# $a1 armazena o valor de N
Sum:   Addi   $t0, $0, 0        # i = 0
       Addi   $v0, $0, 0        # sum = 0
Loop:  Lw     $t2, 0($a0)       # load A[i]
       Add    $v0, $v0, $t2     # sum = sum + A[i]
       addi   $a0, $a0, 4       # add 4 to address
       Addi   $t0, $t0, 1       # i++
       bne    $t0, $a1, Loop    # if(i != 10) goto Loop;
End:   jr     $ra               # return
```


Procedimentos

- Protocolo entre os procedimentos que invocam e os procedimentos invocados
  - Dependente do processador
  - No MIPS:
    - Procedimento espera argumentos nos registos $a0-$a3
    - Coloca valores a retornar nos registos $v0-$v1
    - Outras formas de passagem de parâmetros utilizam a pilha de chamadas (por exemplo, sempre que o número de argumentos ultrapassa o número de registos para utilizar como argumentos)

Sumário

- Quais as responsabilidades do compilador?
  - **Esconder** do programador conceitos baixo-nível da máquina
  - Produzir **rapidamente** código eficiente
  - **Afectar** variáveis a registos locais ou posições de memória
  - **Cálculo** de expressões com constantes
  - **Manter** funcionalidade inicial
  - Geração de instruções de forma a suportar as chamadas aos procedimentos utilizadas no programa
  - **Optimizações**