Conjuntos de Instruções: Contador de Programa, Branches e Jumps

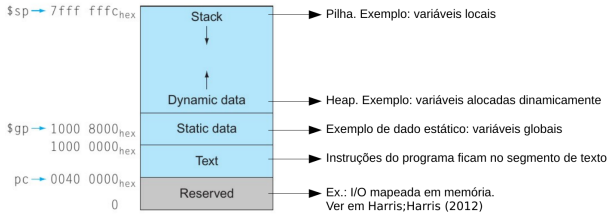
Yuri Kaszubowski Lopes

UDESC

Anotações

Convenção da memória

- Convenção sobre como um programa fica na memória principal da máquina (e.g., RAM)
  - pode mudar de implementação para implementação

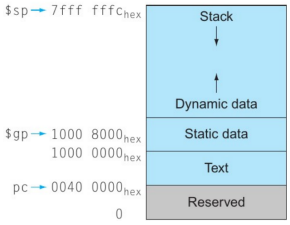


Anotações

Contador de Programa

- Considere o seguinte programa em assembly do MIPS

```
1 0x00400000    ori $t2, $zero, 25
2 0x00400004    lw $t3, 0($s0)
3 0x00400008    add $t4, $t2, $t3
4 0x0040000C    sub $t5, $t2, $t3
```



- O processador sabe qual a próxima instrução a ser executada através do contador de programa
  - Registrador PC (Program Counter)
  - No x86 o PC é chamado de IP (Instruction Pointer)
- Não é diretamente visível/acessível ao programador

Anotações

Contador de Programa

- Durante a execução:
  - O processador carrega a instrução no endereço apontado pelo registrador PC
  - Por motivos que veremos adiante, a primeira coisa que o processador faz é acrescentar +4 no PC para apontar para próxima instrução
    - Por que +4?
    - **+4 já que cada instrução ocupa 4 bytes no MIPS32**
  - O Processador executa a instrução carregada
  - O processo se repete

Anotações

Exemplo

- pc = 0x00400000

```
1 0x00400000    ori $t2, $zero, 25
2 0x00400004    lw $t3, 0($s0)
3 0x00400008    add $t4, $t2, $t3
4 0x0040000C    sub $t5, $t2, $t3
```

Anotações

Exemplo

- pc = 0x00400004

```
1 0x00400000    ori $t2, $zero, 25
2 0x00400004    lw $t3, 0($s0)
3 0x00400008    add $t4, $t2, $t3
4 0x0040000C    sub $t5, $t2, $t3
```

Anotações

Exemplo

- pc = 0x00400008

```
1 0x00400000    ori $t2, $zero, 25
2 0x00400004    lw $t3, 0($s0)
3 0x00400008    add $t4, $t2, $t3
4 0x0040000C    sub $t5, $t2, $t3
```


Anotações

Exemplo

- pc = 0x0040000C

```
1 0x00400000    ori $t2, $zero, 25
2 0x00400004    lw $t3, 0($s0)
3 0x00400008    add $t4, $t2, $t3
4 0x0040000C    sub $t5, $t2, $t3
```


Anotações

Contador de Programa

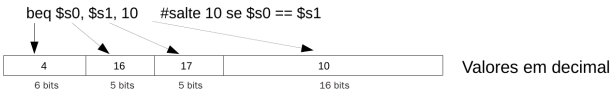
- Como os demais registradores, o contador de programa armazena 32 bits
- Qual o maior programa que podemos escrever em uma arquitetura MIPS de 32 bits?
  - $2^{32} = 4GiB$
  - Na prática esse número é muito menor, já que o programa não é composto somente de instruções
    - Temos o segmento de pilha, heap, dados estáticos, ...
  - Considerando que a primeira instrução do programa está em 0x00400000, existe a possibilidade de em algum momento o **pc** conter o valor 0x00400003 no MIPS32?
    - Não. Toda instrução ocupa 32 bits (4 bytes). Como a memória é endereçada em bytes, os saltos são de 4 em 4
    - As instruções sempre começam em um endereço múltiplo de 4: **Restrição de alinhamento**
    - Comum em muitas arquiteturas: **Não existe essa restrição em x86**

Anotações

Branches: `beq`

- `branch`: desvio
- Desvio condicionais
  - Instruções utilizadas para tomada de decisão
  - `if`, `while`, `for`
- Instrução do tipo-I
- **beq**: branch if equal (desvie se igual)

```
1 beq $s0, $s1, ENDEREÇO
```



Valores em decimal

Anotações

Branches: `bne`

- **bne**: branch if **not** equal (desvie se **não** igual)

```
1 bne $s0, $s1, ENDEREÇO
```

Valores em decimal

Anotações

Branches: Salto

- Problema: Se `ENDEREÇO` apontar para o endereço real da instrução, nenhum programa poderia conter mais de $2^{16} = 64$ Kbytes de instruções.
  - Solução: Os desvios geralmente são tomados para regiões próximas da instrução atual
  - O registrador PC aponta para a próxima instrução a ser executada
  - Dessa forma, o ENDEREÇO em um branch é um "salto" referente ao PC
  - Para aumentar o alcance, o salto é definido em palavras de instrução (4 bytes)
  - O salto pode ser positivo ou negativo
    - Alcance de $\pm 2^{15}$
  - Sendo assim, o endereço de memória efetivo do salto é $PC + 4 + ENDERECO \times 4$
  - Logo, caso a condição do branch se satisfaça:
    - `pc = pc + 4 + ENDERECO*4`

Anotações

Exemplo

- Desejamos ignorar a linha/instrução destacada se `$s0 == $s1` na Linha 5

```
1 0x00400000    lw $s0, 0($t0)
2 0x00400004    lw $s1, 4($t0)
3 0x00400008    lw $s2, 8($t0)
4 0x0040000C    beq $s0, $s1, ENDEREÇO
5 0x00400010    addi $s2, $s2, 5
6 0x00400014    addi $s2, $s2, 10
```

- Qual o valor devemos colocar em `ENDEREÇO`, considerando que caso `$s0` seja igual a `$s1`, devemos saltar para a instrução `0x00400014`?
  - Ao chegar na instrução `beq`, `pc = 0x0040000C`
  - A instrução é carregada para a CPU, e antes de executar a instrução, pc é incrementado e aponta para a próxima, ou seja, `0x00400010`
  - Então o beq deve assumir que o salto deve ser feito a partir de `0x00400010`
  - Como desejamos saltar para `0x00400014`, `0x00400014 - 0x00400010 = 0x4`
  - **Como o salto é feito em palavras**: $4_{16}/4_{16} = 1_{16}$ palavras
  - Logo endereço deve conter $1_{16} = 1_{10}$

Anotações

Facilidades do montador: rótulos (labels)

- Lidar com os endereços dos branches não é tarefa simples
  - Calcular o endereço pode ser confuso
  - Ao inserir uma instrução entre o branch e o seu endereço final, temos que atualizar o branch
  - Pseudo-instruções (vai virar 1, 2, ... instruções?)
- O montador nos poupa desse problema
- Podemos utilizar rótulos (labels) no programa, e pedir por um desvio para o rótulo
  - O montador se encarrega de substituir o rótulo pelo endereço/salto correto quando o programa é montado
  - Rótulos são definidos com um nome único, seguido de dois pontos

Anotações

Facilidades do montador: rótulos (labels)

```
1     lw $s0, 0($t0)
2     lw $s1, 4($t0)
3     lw $s2, 8($t0)
4     beq $s0, $s1, salto
5     addi $s2, $s2, 5
6 salto:
7     addi $s2, $s2, 10
```


Anotações

Comparações

- `slt`: set on less than (atribuir se menor que)
  - Instrução do tipo-R

```
1    slt $s0, $s1, $s2
```

- `$s0 = 1` se `$s1` < `$s2`, 0 caso contrário
- Variantes:
  - `slti`: para imediatos (tipo-I)
  - `sltu`: para comparações sem sinal
  - `sltiu`: para comparações imediatas (tipo-I) sem sinal

Anotações

Exercício Resolvido

- Considere o seguinte trecho de código em C

```
1 if (a > b) {
2     a += 30;
3 }
4 b += 10;
```

- Assumindo que a variável **a** está no registrador `$s0`, e **b** no registrador `$s1`, como fica esse trecho em Assembly do MIPS?

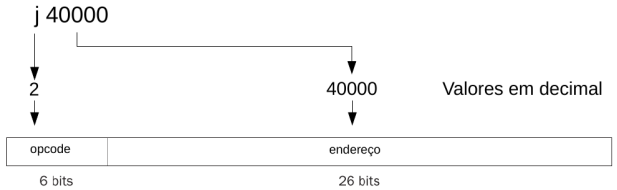
```
1     slt $t0, $s1, $s0  # $t0 recebe 1 se b < a, ou 0 caso contrário
2     beq $t0, $zero, b_maior_igual
3     addi $s0, $s0, 30
4 b_maior_igual:
5     addi $s1, $s1, 10
```

Anotações

Saltos incondicionais: `j`

- **j**: jump (salte para o endereço)

```
1 j 40000
```



- Jumps são instruções do Tipo-J
- Tipo mais simples de instrução

Anotações

Saltos incondicionais: `j`

- Diferente dos desvios condicionais, **os jumps não são relativos ao pc**
  - Estamos efetivamente saltando para a palavra `40000` no exemplo anterior
- Como o endereçamento é em palavras mais uma vez, multiplicamos por 4 para obter o endereço em bytes da instrução
  - Ou seja, `pc = ENDEREÇO << 2`
    - Deslocado 2 bits para multiplicar por 4
- Com isso, o salto tem capacidade efetiva de atingir endereços de até 28 bits
- Os 4 bits mais altos de pc ainda são emprestados para completar os 32 bits
  - necessários para representar um endereço completo: endereçamento pseudodireto
- Da mesma forma que com branches, podemos utilizar **rótulos**, e deixar o cálculo do endereço efetivo a cargo do montador

```
repetir:
    addi $s0, $s0, 1
    beq $s0, $s1, fim
    j repetir
fim:
    ...
```


Anotações

Exercício Resolvido

- Considere o seguinte trecho de código em C

```
while (vet[i] == k) {
    i += 1;
}
vet[i] = k + 10;
```

- Assumindo que as variáveis **i** e **k** se encontram nos registradores `$s3` e `$s5`, respectivamente, e que a base do vetor **vet** está em `$s6`, como fica o trecho em assembly do MIPS? Considere ainda que o vetor é de inteiros, e que cada inteiro ocupa uma palavra.

```
loop:
    sll $t0,$s3,2  # multiplicando i por 4 para ajustar as palavras
    add $t0,$t0,$s6  # adicionando o deslocamento à base do vetor
    lw $t1,0($t0)  # $t1 = vet[i]
    bne $t1,$s5,saida  # saia se vet[i] != k
    add $s3, $s3, 1  # adicionando 1 em i (corpo do loop)
    j loop  # depois de executar o corpo, retorna para o início
saida:
    addi $t1, $s5, 10  # $t1 = k+10
    sw $t1,0($t0)  # vet[i] = $t1, ou seja, k+10
```


Anotações

Exercício Resolvido

- Considere o seguinte desvio

```
    beq $s0, $s1, L1
    #conjunto de instruções 1
L1:
    #conjunto de instruções 2
```

- Considere que o número de instruções entre o `beq` e `L1` é muito grande, e não pode ser endereçado no campo de 16 bits do `beq` (instrução do Tipo-I). Como resolver esse problema adicionando um jump extra?

```
    bne $s0, $s1, L0
    j L1
L0:
    #conjunto de instruções 1
L1:
    #conjunto de instruções 2
```


Anotações

Exercício Resolvido

- Considere o programa em C a seguir

```
if ((a < b && b < 50) || a == -10) {
    vet[b] = vet[b] + vet[b - 20];
} else {
    a = 50;
}
b++;
```

- Assumindo que as variáveis **a** e **b** estão nos registradores `$s0` e `$s1`, respectivamente, e que o endereço base de vet está em `$s2`. Considerando também que o vetor é de inteiros, e que cada inteiro ocupa uma palavra, escreva o programa equivalente em Assembly do MIPS.

Anotações

Exercício Resolvido

- Possível Solução

```
    slt $t0, $s0, $s1  # $t0 = 1 se a < b
    beq $t0, $zero, L1  # se a >= b => pula p/ teste a == -10
    slti $t0, $s1, 50   # $t0 = 1 se b < 50
    beq $t0, $zero, L1   # se b >= 50 => pula p/ o teste a == -10
    j if  # passou pelas duas primeiras condições =>  pula p/ o if
L1:
    ori $t0, -10  # carrega a constante -10 em $t0
    bne $s0, $t0, else  # se a != -10, pula p/ o else
if:
    sll $t0, $s1, 2  # multiplicando b por 4 e salvando em $t0
    add $t0, $t0, $s2  # somando deslocamento com a base do vetor
    lw $t1, 0($t0)  # $t1 = vet[b]
    addi $t2, $s1, -20  # $t2 = b - 20
    sll $t2, $t2, 2  # $t2 * 4 => obter o deslocamento em bytes
    add $t2, $t2, $s2  # somando deslocamento com a base do vetor
    lw $t2, 0($t2)  # $t2 = vet[b-20]
    add $t1, $t1, $t2  # $t1 = vet[b] + vet[b+20]
    sw $t1, 0($t0)  # $vet[b] = $t1
    j saida  # pula para o rótulo saida p/ não executar o else
else:
    ori $s0, 50  # a=50
saida:
    addi $s1, 1  # b++
```


Anotações

Referências

- D. Patterson; J. Henessy. **Organização e Projeto de Computadores: Interface Hardware/Software**. 5a Edição. Elsevier Brasil, 2017.
- Andrew S. Tanenbaum. **Organização estruturada de computadores**. 5. ed. São Paulo: Pearson, 2007.
- Harris, D. and Harris, S. **Digital Design and Computer Architecture**. 2a ed. 2012.
- courses.missouristate.edu/KenVollmar/mars/

Anotações