Arquitetura de Computadores

Assembly – Miscelâneas

Mário O. de Menezes

http://www.tf.ipen.br/~mario

Lembrando

- Instruções *Lógicas* e *Shift* operam em bits individuais, diferente de add/sub que operam na palavra toda.

```
sll, srl, sra, and, andi, or, ori
```

- Use instruções lógicas e shift para isolar campos, mascarando ou deslocando para trás e para frente.
- Use *shift left logical*, `sll`, para multiplicação por potências de 2.
- *shift right arithmetic*, `sra`, para divisão por potências de 2.

Panorama

- Loading/Storing Bytes
- Instruções MIPS Signed vs. Unsigned
- Pseudo−instructions
- Multiplicação/Divisão
- Ponteiros em Linguagem Assembly

Loading, Storing bytes

- Além de transferência de dados em palavras (`lw`, `sw`), MIPS tem também transferência de dados em bytes:
- load byte: `lb`
- store byte: `sb`
- mesmo formato que `lw`, `sw`

Loading, Storing bytes

- Que fazer com os outros 24 bits no registrador de 32 bits?
  - `lb`: extende o sinal para preencher os 24 bits superiores.
- Suponha que o byte 100 tem o valor 0x0F, o byte 200 tenha o valor 0xFF

```
lb $s0, 100($zero) # $s0 = ?

lb $s1, 200($zero) # $s1 = ??
```

- Múltipla escolha para $s0? $s1?

a) 15; b) 255; c) −1; d) −255; e) −15

Loading bytes

- Normalmente, com caracteres, nós não queremos extender o sinal.
- Instrução MIPS que não extende o sinal quando carregando bytes:

load byte unsigned: `lbu`

Panorama

- Loading/Storing Bytes
- **Instruções MIPS Signed vs. Unsigned**
- Pseudo−instructions
- Multiplicação/Divisão
- Ponteiros em Linguagem Assembly

Overflow em Aritmética (1/2)

- **Lembrete: Overflow ocorre quando há um erro na aritmética devido à precisão limitada dos computadores.**
- **Exemplo (números sem sinal 4−bit):**

+15	1111
+3	0011
+18	10010

  - **Mas não há espaço para o bit 5 da solução, então a solução deveria ser 0010, que é +2, e é *errado*.**

Overflow em Aritmética (2/2)

- Algumas linguagens detectam overflow (Ada), outras não (C).
- Solução MIPS é dois tipos de instruções aritméticas para reconhecer 2 escolhas:
  - add (add), add imediato (`addi`) e subtrai (sub) causam detecção de overflow
  - add unsigned (addu), add imediato unsigned (`addiu`) e subtrai unsigned (subu) não causam detecção de overflow
- Compilador seleciona aritmética apropriada.
  - Compilador MIPS C produz `addu`, `addiu`, `subu`

Desigualdades Sem Sinal –Unsigned

- Assim como instruções aritméticas sem sinal:

`addu, subu, addiu`

(na verdade instrução "don't overflow")

- Existem instruções de desigualdade sem sinal:

`sltu, sltiu`

que realmente fazem comparação sem sinal!

0x80000000 < 0x7fffffff signed (slt, slti)

0x80000000 > 0x7fffffff unsigned (sltu,sltiu)

Panorama

- Loading/Storing Bytes
- Instruções MIPS Signed vs. Unsigned
- **Pseudo−instructions**
- Multiplicação/Divisão
- Ponteiros em Linguagem Assembly

Verdadeira Linguagem Assembly

- **Pseudo−instrução: Uma instrução MIPS que não se transforma diretamente em uma instrução em linguagem de máquina.**
- **O que acontece com as pseudo−instruções?**
  - **Elas são quebradas pelo assembler em várias instruções MIPS "reais".**
  - **Mas o que é uma instrução "real" MIPS?**

Exemplos de Pseudo−instruções

- **Move Registrador:** `move reg2,reg1`

  Expande para:

  `add reg2,$zero,reg1`

- **Load Imediato:** `li reg,value`

  Se o valor é de 16 bits:

  `ori reg,$zero,value`

  senão:

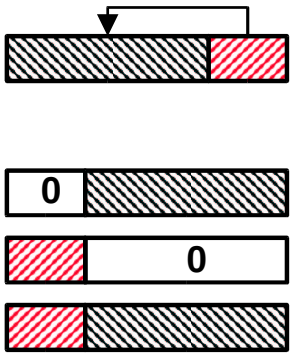
```
lui reg,upper 16 bits of value
ori reg,$zero,lower 16 bits
```


Verdadeira Linguagem Assembly

- **Problema:**
  - Quando quebra uma pseudo−instrução, o assembler pode precisar utilizar um registrador extra.
  - If it uses any regular register, it'll overwrite whatever the program has put into it.
- **Solução:**
  - Reservar um registrador ($1 or $at) que o assembler utilizará quando quebrar as pseudo−instruções.
  - Como o assembler pode utilizá−lo a qualquer momento, não é seguro utilizar este registrador no seu código assembly.

Exemplo de Pseudo−instrução

- **Instrução Rotate Right**




```
ror    reg, value
```

Expande em:

```
srl    $at, reg, value
sll    reg, reg, 32−value
or     reg, reg, $at
```

- **Instrução No operation**

```
nop
```

Expande em?

Linguagem Assembly Verdadeira

- **MAL (MIPS Assembly Language): o conjunto de instruções que um programador pode usar para codificar em MIPS; *inclue* as pseudo−instruções.**
- **TAL (True Assembly Language): o conjunto de instruções que pode realmente ser traduzida em uma única instrução de linguagem máquina (string binária de 32−bit)**
- **Um programa deve ser convertido de MAL em TAL antes de ser traduzido em 1s e 0s.**

Panorama

- Loading/Storing Bytes
- Instruções MIPS Signed vs. Unsigned
- Pseudo−instructions
- **Multiplicação/Divisão**
- Ponteiros em Linguagem Assembly

Multiplicação (1/3)

- **Exemplo em papel e lápis (unsigned):**

```
 Multiplicando    1000   8
  Multiplicador   x1001  9
                   1000
                  0000
                 0000
               +1000
               01001000
```

- **m bits x n bits = produto m + n bits**

Multiplicação (2/3)

- Em MIPS, nós multiplicamos registradores; assim:
  - valor 32−bit x valor 32−bit = valor 64−bit
- Síntaxe da Multiplicação:
  - `mult` register1, register2
  - Multiplica valores 32−bit nos registradores especificados e coloca o produto de 64−bit em dois registradores especiais de resultado:
    - coloca a metade superior em `hi`
    - coloca a metade inferior em `lo`
  - hi e lo são 2 registradores separados dos 32 de propósito geral.

Multiplicação (3/3)

- Exemplo:
  - em C: `a = b * c;`
  - em MIPS:
    - b em $s2; c em $s3; a em $s0 e $s1 (já que ele pode ter até 64 bits)

```
mult $s2,$s3   # b*c
mfhi $s0       # upper half of
               # product into $s0
mflo $s1       # lower half of
               # product into $s1
```

- Nota: Frequentemente, nós só precisamos a metade inferior do produto.

Divisão (1/3)

- **Exemplo com lápis e papel (unsigned):**

```
                    1001   Quociente
Divisor  1000|1001010   Dividendo
            -1000
                10
                101
                1010
               -1000
                  10  Sobra
          (ou resultado Modulo)
```

- **Dividendo = Quociente x Divisor + Sobra**

Divisão (2/3)

- **Síntaxe da Divisão:**
  - `div` register1, register2
  - Divide o valor 32−bits no registrador 1 pelo valor 32−bits no registrador 2:
    - coloca sobra da divisão em hi
    - coloca quociente da divisão em lo
- **Veja que isto pode ser utilizado para implementar tanto o operador divisão (/) quanto o operador módulo (%) da linguagem C.**

Divisão (3/3)

- **Exemplo:**
  - em C: `a = c / d;`

    `b = c % d;`
  - em MIPS:
    - a em $s0; b em $s1; c em $s2; e d em $s3

```
div  $s2,$s3  # lo=c/d, hi=c%d
mflo $s0      # pega quociente
mfhi $s1      # pega sobra
```


Mais Instruções de Overflow

- **Adicionalmente, MIPS tem versões destas duas instruções aritméticas para operandos sem sinal:**

  `multu`

  `divu`

Panorama

- Loading/Storing Bytes
- Instruções MIPS Signed vs. Unsigned
- Pseudo−instructions
- Multiplicação/Divisão
- **Ponteiros em Linguagem Assembly**

Problemas Comuns com Ponteiros: Hilfinger

1. Algumas pessoas não entendem a distinção entre `x = y` e `*x = *y`

2. Alguns simplesmente não têm bastante prática em hackear ponteiros, tal como inserir um elemento em uma lista (no meio).

3. Alguns não entendem a distinção entre `struct Foo x;` e `struct Foo *x;`

4. Alguns não entendem os efeitos de `p = &x` e os resultados subsequentes de atribuir através de dereferências de p, ou da desalocaçao de `x`.

Endereço vs. Valor

- Conceito fundamental de Ciência da Comp.
- Mesmo em planilhas de cálculo: selecione a célula A1 para usar na célula B1

	A	B
1	100	100
2		

- Você quer colocar o endereço da célula A1 na fórmula (=A1) ou o valor de A1 (100)?
- Diferença? Quando muda A1, células que usam o endereço mudam, mas não as células com o valor antigo.

Código Assembly para Implementar Ponteiros

- dereferenciando => transf. dados em asm.
  - ... = ... *p ...; => load

    (pegue valor da localização apontada por p)
    load word (lw) se for ponteiro para inteiro,
    load byte unsigned (lbu) se for ponteiro para char
  - *p = ...; => store

    (coloca o valor na localização apontada por p)

Código Assembly para implementar Ponteiros

c é `int`, tem valor 100, na memória no endereço 0x10000000, p em `$a0`, x em `$s0`

```
  p = &c;  /* p gets 0x10000000 */
  x = *p;  /* x gets 100 */
 *p = 200; /* c gets 200 */
```

```
  # p = &c;  /* p gets 0x10000000 */
lui $a0,0x1000 # p = 0x10000000

  # x = *p;  /* x gets 100 */
lw  $s0, 0($a0) # dereferencing p

  # *p = 200; /* c gets 200 */
addi $t0,$0,200
sw   $t0, 0($a0) # dereferencing p
```


Registradores e Ponteiros

- **Registradores não tem endereços**

  => registradores não podem ser apontados

  => **não se pode alocar uma variável a um registrador se ela pode ter um ponteiro para ela**

C vs. Asm com Aritmética de Ponteiros

```
int strlen(char *s) {
    char *p = s; /* p points to chars */

   while (*p != '\0')
     p++; /* points to next char */
  return p - s; /* end - start */
 }
```

```
        mov  $t0,$a0
        bu  $t1,0($t0) /* derefence p */
        eq  $t1,$zero, Exit

Loop:  addi $t0,$t0,1  /* p++ */
       lbu $t1,0($t0) /* derefence p */
       bne  $t1,$zero, Loop

Exit:  sub $v0,$t1,$a0
       jr $ra
```


"E, concluindo..."

- MIPS Signed v. Unsigned "overloaded" term
  - Faz/Não Faz extensão de sinal (lb, lbu)
  - Não tem overflow (addu, addiu, subu, multu, divu)
  - Faz comparação signed/unsigned (slt,slti/sltu,sltiu)
- Assembler usa $at para transformar MAL em TAL
- Instruções MIPS mult/div usam registradores hi, lo
- Dereferenciando Ponteiros diretamente suportado como load/store.