

Arquitecturas RISC

Programação em *assembly*

MIPS

MIPS vs. IA32

MIPS	IA32
Conjunto limitado de instruções	Conjunto alargado de instruções
Instruções simples e regulares	Instruções complexas e irregulares
Número alargado de registos	Número reduzido de registos
3 operandos por instrução	2 operandos por instrução
Operandos sempre em registos (arquitectura load/store)	Operandos em registos ou memória
Saltos condicionais : não utilizam flags.	Saltos condicionais : utilizam flags.
Modos de endereçamento simples	Modos de endereçamento complexos

MIPS: registos e 3 operandos

```
main () {  
    int a,b,c,d,e,f,g;  
  
    c = a * b;  
    d = e + f;  
    g = c - d;  
}
```

MIPS

```
main:  
    mul $t2, $t0, $t1    # c=a*b  
    add $t3, $t4, $t5    # d=e+f  
    sub $t6, $t2, $t3    # g=c-d  
    jr $ra
```

INTEL

```
main:  
    ...  
    movl -4(%ebp), %eax  
    movl -8(%ebp), %ecx  
    imull %eax, %ecx  
    movl %ecx, -12(%ebp) # c=a*b  
    movl -20(%ebp), %eax  
    addl -24(%ebp), %eax  
    movl %eax, -16(%ebp) # d=e+f  
    subl %edx, %ecx  
    movl %ecx, -28(%ebp) # g=c-d  
    ...
```

MIPS: operandos em registos

```
int a,b;
main () {

    a = a + b;
}
```

INTEL

```
main:
    ...
    movl b, %eax
    addl %eax, a          # a=a+b
    ...
```

MIPS

```
main:
    lw $t5, a          # $t5 = a
    lw $t1, b          # $t1 = b
    add $t5, $t5, $t1
    sw $t5, a          # a=a+b
    ...
```

Todos os operandos devem estar em registos, o que obriga a ler as variáveis de memória, antes de realizar as operações.

MIPS: modos de endereçamento

```
struct {int a,b;} arr[100];
main () {
    int i;

    for (i=0 ; i<100 ; i++)
        arr[i].b=i;
}
```

O modo de endereçamento mais simples, obriga ao cálculo do endereço de `arr` com 3 instruções em vez de uma, como seria no IA32:

```
movl Ri, 4(Rbase, Ri, 8)
```

MIPS

```
main:
    ...
    li $t0, arr # $t0=&arr
    li $t1, 0 # i=0
ciclo:
    slti $t2, $t1, 100
    bnez $t2
    sll $t3, $t1, 3 # escala
    add $t3, $t3, $t0
    sw $t1, 4($t3) # arr[i].b=i
    addi $t1, $t1, 1 # i++
    beq $t1, $t1, ciclo
    ...
```

MIPS: funções

```
int maximo;  
main ()  
{  
    maximo = max (100, -20);  
}
```

- O endereço de retorno é colocado no registo `$ra` pela instrução `jal`;
Qualquer função que chame outra deve guardar na stack o registo `$ra`
- As funções terminam saltando para o endereço contido no registo `$ra`
- Não existem instruções de `push` e `pop`
- Os parâmetros são colocados nos registos `$a0..$a3`
- O resultado da função é devolvido no registo `$v0`

MIPS

```
main:  
...  
addi $sp, $sp, -4  
sw $ra, 0($sp)      # push $ra  
li $a0, 100         # param.  
li $a1, -20  
jal max  
sw $v0, maximo  
lw $ra, 0($sp)  
addi $sp, $sp, 4    # pop $ra  
jr $ra
```

MIPS: registos seguros

```
int factorial (int n)
{
  if (n>0)
    return (n*factorial(n-1));
  else
    return (1);
}
```

- Os registos \$s.. são *callee save*, isto é, qualquer função que os use deve preservar os seus valores guardando-os na *stack*

```
main:
  addi $sp, $sp, -8
  sw $ra, 4($sp)      # push $ra
  sw $s0, 0($sp)     # push $s0
  move $s0, $a0
  blez $s0, else
  addi $a0, $a0, -1   # param.
  jal factorial
  mul $v0, $s0, $v0   # return
  beq $v0, $v0, fim
else:
  li $v0, 1          # return
fim:
  lw $s0, 0($sp)     # pop $s0
  lw $ra, 4($sp)     # pop $ra
  addi $sp, $sp, 8
  jr $ra
```

MIPS - Arquitectura

Tema	Hennessy [COD]	Bryant [CS:APP]
MIPS – Programação	Sec 3.1 a 3.3 Sec 3.5, 3.6	