Arquitecturas RISC

Arquitectura e Conjunto de Instruções

MIPS

Arquitecturas CISC

Caracterizam-se por:

- conjunto alargado de instruções
  - instruções complexas
  - instruções altamente especializadas
- existência de vários formatos de instruções
  - tamanho variável
- suporte de vários modos de endereçamento
  - incluindo modos complexos
- número reduzido de registos
- instruções com 0, 1 ou 2 operandos
- suporte para operandos em memória

Argumentação CISC (década 60, 70)

A programação em *assembly* e compiladores pouco sofisticados justificavam que as instruções estivessem perto das HLL

Acreditava-se que instruções complexas executavam mais rapidamente uma operação do que sequências de instruções simples

Instruções complexas permitiam que o programa fosse composto por menos instruções, poupando memória

Os processadores possuíam poucos registos, exigindo o recurso constante a operandos em memória – arquitecturas *register-memory* ou *memory-memory*

Argumentação RISC (década 80)

1. A utilização das instruções complexas e modos de endereçamento sofisticados é pouco frequente, pois os compiladores têm que encontrar casos que se adaptem exactamente à instrução.
2. Suportar instruções complexas implica uma unidade de controlo mais sofisticada, com um período de relógio maior. Isto aumenta também o tempo de execução das instruções mais simples, cuja utilização é mais frequente.

Conclusão - reduzir o conjunto de instruções e modos de endereçamento àqueles estritamente necessários. Operações complexas são conseguidas por **software**, à custa de sequências de instruções simples. A simplificação da unidade de controlo permite que estas instruções executem mais rapidamente.

Argumentação RISC (década 80)

1. Os operandos em memória aumentam o tempo de execução das operações da ALU.

2. As variáveis escalares são as mais frequentemente utilizadas nos programas. O acesso a estas variáveis deve ser o mais eficiente possível.

Conclusão - Aumentar o número de registos para que as variáveis escalares possam ser alocadas a estes.
Os operandos das instruções lógico-aritméticas têm que ser sempre registos.
A memória só é acedida por instruções especiais: *load* e *store.*

RISC vs. CISC: arquitectura

RISC	CISC
Reduzido nº de instruções que realizam operações elementares	Elevado nº de instruções incluindo operações complexas
Instruções de tamanho fixo	Instruções de tamanho variável
Poucos formatos de instruções	Muitos formatos de instruções
Poucos modos de endereçamento	Grande número de modos de endereçamento
Muitos registos de uso genérico	Poucos registos, alguns com utilizações pré-definidas

RISC vs. CISC: programação

RISC	CISC
Utilização intensiva de registos: operandos não podem estar em memória	Utilização intensiva de operandos em memória
Modos de endereçamento simples requerem cálculo de endereços por software	Modos de endereçamento complexos, permitem que muitos endereços possam ser calculados pelo hardware
3 operandos por instrução	2 operandos por instrução
Parâmetros, endereço de retorno e valor das funções em registos	Parâmetros e endereço de retorno na *stack*
Operações complexas conseguidas à custa de operações simples	Operações complexas implementadas com uma única instrução

Arquitectura: MIPS

Atributos visíveis ao programador:

- Tamanho da palavra
- Número de Operandos
- Registos visíveis
- Endereçamento de Operandos
- Tipos de Instruções
- O conjunto de instruções (ISA)

Palavra e número de operandos: MIPS

- Tamanho da palavra: 32 *bits*
- 3 operandos por instrução

sub $s0, $t0, $t1

$s0 = $t0 - $t1

Registos visíveis: MIPS

32 registos genéricos de 32 *bits*

Nº	Nome	Nº	Nome	Nº	Nome	Nº	Nome
0	$zero	8	$t0	16	$s0	24	$t8
1	$at	9	$t1	17	$s1	25	$t9
2	$v0	10	$t2	18	$s2	26	$k0
3	$v1	11	$t3	19	$s3	27	$k1
4	$a0	12	$t4	20	$s4	28	$gp
5	$a1	13	$t5	21	$s5	29	$sp
6	$a2	14	$t6	22	$s6	30	$fp
7	$a3	15	$t7	23	$s7	31	$ra

Apesar de genéricos, o registo $zero não pode ser escrito, e alguns registos ($at, $k0, $k1, $gp, $sp, $fp, $ra) têm funções específicas.

Endereçamento de operandos: MIPS

Modo	Observações
Imediato	1 constante (16 bits) codificada na própria instrução
Registo	add $t0, $v0, $v1
Memória (base + offset)	lw $s0, 1000 ($gp) sw $t5, -500($sp)
IMPORTANTE: A memória só pode ser acedida por instruções load / store Todos os operandos de operações lógico aritméticas devem ser registos ou constantes (1 constante por instrução, máximo)	

Operações lógico-aritméticas: MIPS

Caso geral: 3 operandos

```
add Rdest, Rsrc1, Rsrc2          sub Rdest, Rsrc1, Rsrc2
andi Rdest, Rsrc1, Imm16         mul Rdest, Rsrc1, Rsrc2
sra Rdest, Rsrc1, Shamt5
```

Onde:

- Shamt5 ∈ [0, 31] **- Imm16 valor 16 bits**

Excepções: 2 operandos

```
mult Rsrc1, Rsrc2                div Rsrc1, Rsrc2
# MSW em hi, LSW em lo           # lo = Rsrc1 / Rsrc2
mfhi $reg;   mflo $reg           # hi = Rsrc1 mod Rsrc2

abs Rdest, Rsrc1
```

Transferência de informação: MIPS

```
lb Rdest, Imm16(Rbase)          lw Rdest, Imm16(Rbase)

sb Rsrc, Imm16(Rbase)           sw Rsrc, Imm16(Rbase)

lui Rdest, Imm16

mfhi Rdest                      mflo Rdest
```

Onde:

- **Imm16 valor 16 bits**

Pseudo-instrução: `move Rdest, Rsrc`

Operações de teste: MIPS

```
slt Rdest, Rsrc1, Rsrc2

# Rdest=1 se Rsrc1<Rsrc2
# senão Rdest=0
```

```
slti Rdest, Rsrc1, Imm16

# Rdest=1 se Rsrc1<Imm16
# senão Rdest=0
```

Controlo de fluxo: MIPS

```
j label                         jr Rdest

beq Rsrc1, Rsrc2, label         bne Rsrc1, Rsrc2, label

bgez Rsrc, label                bgtz Rsrc, label
blez Rsrc, label                bltz Rsrc, label
```

INVOCAÇÃO DE PROCEDIMENTOS

```
jal label                       jalr Rsrc
# endereço de retorno em $ra

jr $ra
```

Pseudo-instruções

- O *assembly* suporta algumas pseudo-instruções;
  dão a ilusão de um conjunto de instruções mais rico mas são transformadas pelo *assembler* noutras instruções equivalentes

`move Rdst, Rsrc`

Move Rsrc para Rdst:

```
or Rdst, $zero, Rsrc
```

Pseudo-instruções

`b <label>`

Salto relativo incondicional:

```
beq $at, $at, <label>
```

`blt Reg1, Reg2, <label>`

Salto se Reg1 < Reg2:

```
slt $at, Reg1, Reg2
bne $at, $zero, <label>
```

Pseudo-instruções

- `li Rdst, <imediato>`
  Carrega um valor imediato para um registo
  A conversão para instruções nativas depende do tamanho da constante

`li Rdst, <Imm16>`

```
ori Rdst, $zero, <Imm16>
```

`li Rdst, <Imm32>`

```
lui $at, <16 bits + significativos de Imm32>
ori Rdst, $at, <16 bits - significativos de Imm32>
```

- `la Rdst, <addr32>`
  Carrega um endereço para um registo. É equivalente a
  `li Rdst, <imediato32>`

Pseudo-Instruções

`addi Rdst, Rsrc, <Imm32>`

Adição com constante maior que 16 *bits*:

```
lui $at, <16 bits + sig>
ori $at, $at, <16 bits – sig>
add Rdst, Rsrc, $at
```

`subi $Rdst, $Rsrc, <Imm>`

Subtração com constante

```
addi $Rdst, $Rsrc, -<Imm>
```

Reordenação de instruções

- Ao nível máquina a instrução imediatamente a seguir a um salto condicional é SEMPRE executada!

--- Considere este código ao nível máquina ----

```
bne $t0, $t1, label
addi $v0, $v1, 1    # EXECUTADA independentemente
                    # do resultado do bne !!!!
```

O *assembly* esconde este facto deixando a tarefa de reordenação para o *assembler*

Este pode inserir um nop após a instrução de salto!

Reordenação de instruções

Mas se o código *assembly* incluir a directiva `.set noreorder` então o *assembly* não reordena nem inclui `nop`

```
.set noreorder
bne $t0, $t1, label
addi $v0, $v0, 1   # EXECUTADA
```

Se `.set noreorder` não estiver activo então o *assembler* insere um `nop` após o salto condicional

MIPS - Arquitectura

Tema	Hennessy [COD]	Bryant [CS:APP]
RISC vs. CISC	Sec 3.12 a 3.15	
MIPS – Arquitectura	Sec 3.1 a 3.3 Sec 3.5	