Arquitectura de Computadores

MIPS : Representação de instruções

Docente: Pedro Sobral
http://www.ufp.pt/~pmsobral

Programa guardado em memória

- Computadores têm por base 2 ideias:
  1) As instruções são representadas como números.
  2) Portanto, programas inteiros podem ser guardados em memória para leitura e escrita como os dados.
- Simplifica o SW/HW dos sistemas de computação:
  - A tecnologia de memória usada para os dados também é usada para os programas.

Consequência #1: Tudo tem endereço

- Como todas as instruções e dados se encontram em memória como números, tudo têm um endereço de memória: instruções e dados.
  - Os saltos (condicionais e incondicionais) usam estes endereços!
- Um registo guarda o endereço da instrução a ser executada: "Program Counter" (PC)
  - Basicamente é um apontador para a memória: A Intel chama-lhe "Instruction Address Pointer".

Consequência #2: Compatibilidade binária

- Os programas são distribuídos em binário…
  - E só executam num conjunto de instruções específico.
  - Versões diferentes para ARM e INTEL
- As novas máquinas devem permitir executar programas "antigos" (em binário) bem como programas compilados com as novas instruções.
- Leva a que o conjunto de instruções vá evoluindo
- A escolha do Intel 8086 em 1981 para o 1º IBM PC é a principal razão para que os PCs ainda usem o conjunto de instruções 80x86

Instruções são Números (1/2)

- Os dados usados habitualmente têm 32 bits
  - Cada registo têm 32 bits.
  - O `lw` e o `sw` acedem à memória 32 bit de cada vez.
- Portanto como representar as instruções?
  - Nota: O computador só entende 1s e 0s, portanto "`add $t0,$0,$0`" não tem significado.
  - O MIPS pretende simplicidade : como os dados estão em "words", também as instruções são "words".

Instruções são Números (2/2)

- Uma "word" tem 32 bits, portanto vamos dividi-la em "campos".
- Cada campo indica uma característica da instrução.
- Podemos definir campos diferentes para cada instrução, mas como o MIPS é baseado na simplicidade, temos 3 tipos de formatos para as instruções:
  - Formato-R de "register"
  - Formato-I de "Immediate"
  - Formato-J de "jump"

Formato das Instruções

- **Formato-I**:
  - usado para instruções com constantes, `lw` e `sw` (uma vez que o deslocamento conta como uma constante), e para os saltos condicionais (`beq` e `bne`),
  - (Mas não para as instruções de deslocamento...)
- **Formato-J**:
  - usado para o `j` e o `jal`
- **Formato-R**:
  - usado pelas restantes instruções

Instruções com o formato-R (1/5)

- Definem-se "campos" com o seguinte número de bits: 6 + 5 + 5 + 5 + 5 + 6 = 32

6	5	5	5	5	6

- Por simplicidade cada campo tem nome:

opcode	rs	rt	rd	shamt	funct

- Importante: Nestes slides e no livro, cada campo é visto como um inteiro sem sinal de 5 ou 6 bits e não como parte de um inteiro com 32 bits.
- Consequência: 5-bit representam um número de 0-31, enquanto 6-bit representam um número de 0-63.

Instruções com o formato-R (2/5)

- O que representam os campos?
  - `opcode`: indica, parcialmente, de que instrução se trata
    - Nota: Este número é sempre 0 para todas as instruções com formato-R.
  - `funct`: combinado com o `opcode`, este número especifica exactamente a instrução
  - Pergunta: porque é que o `opcode` e o `funct` não são um campo único com 12 bits?
    - Resposta: veremos mais tarde...

Instruções com o formato-R (3/5)

- Mais campos:
  - `rs` ("Source Register"): *geralmente* usado para especificar o registo contendo o primeiro operando
  - `rt` ("Target Register"): *geralmente* usado para indicar o registo contendo o segundo operando (note que o nome engana…)
  - `rd` ("Destination Register"): *geralmente* usado para indicar o registo que vai receber o resultado da computação

Instruções com o formato-R (4/5)

- Notas sobre os campos dos registos:
  - Cada campo dos registos tem exactamente 5 bits o que quer dizer que pode especificar um valor de 0-31. Cada um destes campos identifica um dos 32 registos pelo seu número.
  - A palavra "geralmente" foi usada porque há excepções que veremos depois, por exemplo,
    - `mult` e `div` não tem nada de importante no campo `rd` uma vez que os registos de destino são o `hi` e o `lo`
    - `mfhi` e `mflo` não têm nada de importante nos campos `rs` e `rt` uma vez que a fonte é determinada pela instrução (p. 264 P&H)

Instruções com o formato-R (5/5)

- Campo final:
  - `shamt`: este campo contém o valor do deslocamento para as instruções deste tipo. Deslocar uma palavra de 32 bits mais do que 31 é inútil portanto este campo tem apenas 5 bits (0-31).
  - Este campo está sempre a zero, excepto nas instruções de deslocamento.
- Para uma descrição detalhada dos campos das instruções consultar folha de referência do MIPS (elearning)

Exemplo do Formato-R (1/2)

° **Instrução MIPS:**

```
add    $8,$9,$10
```

`opcode` = 0 (ver na tabela de instruções)
`funct` = 32 (ver na tabela de instruções)
`rd` = 8 (destino)
`rs` = 9 (primeiro *operando*)
`rt` = 10 (segundo *operando*)
`shamt` = 0 (não é um deslocamento)

Exemplo do Formato-R (2/2)

° **Instrução MIPS:**

```
add    $8,$9,$10
```

Representação decimal dos campos:

0	9	10	8	0	32

Representação binária dos campos:

000000	01001	01010	01000	00000	100000

hex

Em hexadecimal: $012A\ 4020_{hex}$
Em decimal: $19{,}546{,}144_{10}$

• **Chama-se** Instrução em Linguagem Máquina

Formato das Instruções

- **Formato-I**:
  - usado para instruções com constantes, `lw` e `sw` (uma vez que o deslocamento conta como uma constante), e para os saltos condicionais (`beq` e `bne`),
  - (Mas não para as instruções de deslocamento...)
- **Formato-J**:
  - usado para o `j` e o `jal`
- **Formato-R**:
  - usado pelas restantes instruções

Instruções com o formato-R (recordando)

- Definem-se "campos" com o seguinte número de bits: 6 + 5 + 5 + 5 + 5 + 6 = 32

6	5	5	5	5	6

- Por simplicidade cada campo tem nome:

opcode	rs	rt	rd	shamt	funct

Instruções com o Formato-I (2/4)

° Definem-se "campos" com o seguinte número de bits: 6 + 5 + 5 + 16 = 32 bits

6	5	5	16

° Cada campo tem um nome:

opcode	rs	rt	immediate

° **Conceito chave**: Apenas um campo é inconsistente com o formato-R. E,mais importante, o `opcode` tem a mesma posição.

Instruções com o Formato-I (3/4)

° Qual o significado destes campos?

- `opcode`: o mesmo que no formato-R excepto que, como não há campo `funct`, o `opcode` especifica unicamente uma instrução no formato-I
- Esta é a razão da existência de 2 campos de 6-bit no formato-R em vez de 1 campo de 12 bit para identificar a instrução: Para manter a consistência com os outros formatos.
- `rs`: indica o *único* registo a operar (se existir)
- `rt`: indica o registo que recebe o resultado (é por isso que se chama "*target* register" - rt)

Instruções com o Formato-I (4/4)

- O campo "Immediate":
  - Para as instruções `addi`, `slti`, `sltiu`, a constante de 16 bit sofre extensão do sinal para 32 bits. Portanto é tratada como um inteiro com sinal.
  - 16 bits ➔ podem representar 2^{16} valores distintos.
  - Isto é suficiente para o deslocamento tipico nas instruções `lw` or `sw`, além de ser suficiente para uma larga maioria de valores usados na instrução `slti` .
  - Veremos mais à frente o que fazer se o número não for representável em 16 bits…

Exemplo do Formato-I (1/2)

- Instrução MIPS:

```
addi    $21,$22,50
```

`opcode` = 8 (ver na tabela)

`rs` = 22 (registo com o primeiro operando)

`rt` = 21 (registo de destino)

`immediate` = 50 (em decimal, por defeito)

Exemplo do Formato-I (2/2)

- **Instrução MIPS:**

```
addi    $21,$22,50
```

Representação em decimal:

8	22	21	50

Representação em binário:

001000	10110	10101	0000000000110010

hexadecimal: $22D5\ 0032_{hex}$

decimal: $584,384,562_{10}$

Questão?

Que instrução tem a mesma representação que 35_{10}?

1. add $0, $0, $0

opcode	rs	rt	rd	shamt	funct

2. subu $s0,$s0,$s0

opcode	rs	rt	rd	shamt	funct

3. lw $0, 0($0)

opcode	rs	rt	offset

4. addi $0, $0, 35

opcode	rs	rt	immediate

5. subu $0, $0, $0

opcode	rs	rt	rd	shamt	funct

6. Errado!
   Instruções não são números

Nomes e números dos registos:
0: $0, .. 8: $t0, 9:$t1, ..15: $t7, 16: $s0, 17: $s1, .. 23: $s7

Campos "Opcode" e "funct" (se necessário)

- `add`: opcode = 0, funct = 32
- `subu`: opcode = 0, funct = 35
- `addi`: opcode = 8
- `lw`: opcode = 35

Problemas do Formato-I (1/3)

- **Problema 1:**
  - Ha boas hipóteses de as constantes usadas pelas instruções `addi`, `lw`, `sw` e `slti` possam se representadas em 16 bit.
  - …mas e se não for possível?
  - Necessitamos de uma forma de lidar com uma constante de 32 bit em todas as instruções com o formato-I.

Problemas do Formato-I (2/3)

- **Solução do Problema 1:**
  - Resolver em software acrescentando uma nova instrução.
  - Não alterar as instruções existentes! Adicionar uma nova para resolver este problema.
- **Nova instrução:**

```
lui   registo, immediate
```

  - Quer dizer "Load Upper Immediate"
  - Pega num inteiro de 16-bit e coloca-o na metade mais significativa do registo especificado.
  - Coloca a metade menos significativa a 0s

Problemas do Formato-I (3/3)

- Solução do Problema 1 (continuação):
  - Como pode o `lui` ajudar?
  - Exemplo:

```
        addi    $t0,$t0, 0xABABCDCD
  fica:
        lui     $at, 0xABAB
        ori     $at, $at, 0xCDCD
        add     $t0,$t0,$at
```

  - Assim cada instrução no formato-I tem apenas uma constante de 16 bit…
  - Não era bom o assemblador fazer isto automaticamente? (veremos…)

Saltos: PC como endereço base (1/5)

- Formato-I

opcode	rs	rt	immediate

- `opcode` especifica `beq` ou `bne`
- `rs` e `rt` são os registos a comparar
- O que pode o "`immediate`" indicar?
  - `Immediate` só tem 16 bits
  - PC (Program Counter) tem o endereço da instrução que está a ser executada; um apontador de 32-bit para a memória
  - Portanto o "`immediate`" não pode indicar inteiramente o endereço para onde saltar!

Saltos: PC como endereço base (2/5)

- Onde usamos geralmente os saltos condicionais?
  - Resposta: `if-else`, `while`, `for`
  - Os ciclos são geralmente pequenos: tipicamente até 50 instruções.
  - Chamadas a funções e saltos incondicionais são feitos usando `j` e `jal`, e não `beq` e `bne`.
- Conclusão: Podemos querer saltar para qualquer zona da memória mas um salto condicional geralmente altera pouco o PC (soma ou subtrai um pequeno valor)

Saltos: PC como endereço base (3/5)

- Solução para codificar saltos numa instrução de 32 bits: Endereçamento Relativo ao PC
- O campo "`immediate`" com 16 bit representa um inteiro com sinal em complemento para 2 que vai ser *somado* ao PC se fizermos o salto.
- Assim podemos saltar $\pm 2^{15}$ bytes a partir do PC, o que deve ser suficiente para qualquer ciclo.
- Será que podemos optimizar ainda mais?

Saltos: PC como endereço base (4/5)

- Nota: Instruções têm 32 bits, portanto têm que estar alinhadas à palavra (o seu endereço é sempre um múltiplo de 4 portanto termina sempre em 00 em binário).
  - Portanto o número de bytes a somar ao PC é sempre um múltiplo de 4.
  - Portanto vamos indicar o "`immediate`" em palavras.
- Sendo assim podemos saltar $\pm 2^{15}$ palavras desde o PC (ou $\pm 2^{17}$ bytes), portanto podemos ter ciclos 4 vezes maiores

Saltos: PC como endereço base (4/5)

- Cálculo do salto:
  - Se não fizermos o salto:

    PC = PC + 4

    PC+4 = o endereço da próxima instrução
  - Se fizermos o salto:

    PC = (PC + 4) + (`immediate` * 4)
  - Observações
    - O campo "`immediate`" indica o número de palavras a saltar o que é simplesmente o número de instruções a saltar!
    - O campo "`immediate`" pode ser positivo ou negativo
    - Devido ao hardware, soma-se o "`immediate`" a (PC+4), e não a PC; porquê? Veremos…

Exemplo de um salto condicional (1/3)

- **Código MIPS:**

```
Loop:  beq   $9,$0,End
       add   $8,$8,$10
       addi  $9,$9,-1
       j     Loop
End:
```

- `beq` tem o formato-I:
  - `opcode` = 4 (ver tabela)
  - `rs` = 9 (primeiro operando)
  - `rt` = 0 (segundo operando)
  - `immediate` = ???

Exemplo de um salto condicional (2/3)

- **Código MIPS:**

```
Loop:  beq   $9,$0,End
       addi  $8,$8,$10
       addi  $9,$9,-1
       j     Loop
End:
```

- **Campo "`Immediate`":**
  - Número de instruções a adicionar (ou subtrair) ao PC, começando na instrução *depois* do salto.
  - No caso do `beq`, "`immediate`" = 3

Exemplo de um salto condicional (3/3)

- Código MIPS:

```
Loop: beq   $9,$0,End
      addi  $8,$8,$10
      addi  $9,$9,-1

      j     Loop
 End:
```

Representação decimal:

4	9	0	3

Representação binária:

000100	01001	00000	0000000000000011

Problemas dos saltos relativos ao PC

- Será que os endereços dos saltos se mantêm válidos se o código mudar de posição?
- O que fazer se o destino está a mais de $> 2^{15}$ instruções do salto?

Instruções com o Formato-J (1/5)

- Para os saltos condicionais especificamos apenas, a *alteração* a efectuar no valor corrente do PC.
- Para saltos absolutos (`j` e `jal`), necessitamos de poder saltar para *qualquer* parte da memória.
- O ideal era poder indicar um endereço de 32 bit para o qual saltar…
- Infelizmente, é impossível guardar um opcode de 6 bit e um endereço de 32 bit numa instrução de 32 bit, portanto é necessário um compromisso…

Instruções com o Formato-J (2/5)

- Definem-se campos com o seguinte número de bits:

6 bits	26 bits

- Cada campo tem um nome:

opcode	target address

- Conceitos chave
  - O campo `opcode` é idêntico ao dos outros formatos.
  - Combina todos o outros campos para poder guardar um maior endereço

Instruções com o Formato-J (3/5)

- Por agora já podemos indicar 26 bit de um endereço de 32 bit.
- Optimização:
  - Note que, saltos incondicionais, tal como os condicionais, só saltam para endereços alinhados à palavra, portanto os 2 últimos bits são 00 (em binário).
  - Portanto não vamos indica-los!

Instruções com o Formato-J (3/5)

- Agora já temos 28 de 32…
- Onde encontrar os outros 4?
  - Por definição, usar or 4 bits mais significativos do PC.
  - Tecnicamente, isto quer dizer que não podemos saltar para *qualquer* posição de memória, mas é apropriado 99.9999…% das situações, uma vez que os programas não são assim tão longos.
    - Só causa problemas para código > 256 MB
  - Se for necessário absolutamente especificar um endereço de 32-bit podemos sempre colocá-lo num registo e usar a instrução `jr`.

Instruções com o Formato-J (5/5)

- Sumário:
  - Novo PC = { PC[31..28], target address, 00 }
- Perceber a origem de cada campo!
- Nota: { , , } quer dizer concatenação
  { 4 bits , 26 bits , 2 bits } = 32 bit
  - { 1010, 11111111111111111111111111, 00 } = 10101111111111111111111111111100

Concluindo…

- Simplificar o MIPS: Definir instruções que têm o mesmo tamanho que as palavras de dados de forma a poder usar a mesma memória (compilador pode usar `lw` e `sw`).
- Computador guarda os programas como uma série de números de 32 bit.
- Instrução máquina do MIPS
  32 bits representam uma instrução

R	opcode	rs	rt	rd	shamt	funct
I	opcode	rs	rt	immediate		
J	opcode	target address				

Concluindo…

- Saltos condicionais usam endereçamento relativo ao PC
- Desassemblar uma instrução é simples e começa por descodificar o campo `opcode`.