Arquitetura de Computadores I

Instruções executáveis por um processador - a arquitetura MIPS

(2º grupo de slides)

António de Brito Ferrari
ferrari@ua.pt

8. Representação de inteiros com e sem sinal (**revisão**)

Inteiros positivos em binário

- Número em binário com n-bits

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Gama de representação: **0** a **$+2^n - 1$**
- Exemplo
  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
    $= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$
- Gama de representação com 32 bits
  - 0 a +4,294,967,295

Inteiros com sinal: 2s-Complement

- Número com n-bits em 2s-Complement:

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Gama de representação: -2^{n-1} a $+2^{n-1} - 1$
- Exemplo
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^2 + 0\times2^1 + 0\times2^0$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$
- Gama de representação com 32 bits
  - –2,147,483,648 a +2,147,483,647

Inteiros com sinal: 2s-Complement (2)

Bit 31 é o bit de sinal:
1 para numeros negativos
0 para numeros positivos e para zero
Gama de representação assimétrica: $-(-2^{n-1})$ não é representável em n-bits
Numeros não-negativos têm a mesma representação em unsigned e em 2s-complement
Exemplos:

0:	0000 0000 … 0000
–1:	1111 1111 … 1111
Mais-negativo:	1000 0000 … 0000
Mais-positivo:	0111 1111 … 1111

Negação (obtenção do simétrico)

- Complementar e somar 1
  – Complementar significa inverter bit a bit 1 → 0, 0 → 1

$$x + \bar{x} = 1111\ldots111_2 = -1$$

$$\bar{x} + 1 = -x$$

- Examplo: negar +2
  - $+2 = 0000\ 0000\ \ldots\ 0010_2$
  - $-2 = 1111\ 1111\ \ldots\ 1101_2 + 1$
    $= 1111\ 1111\ \ldots\ 1110_2$

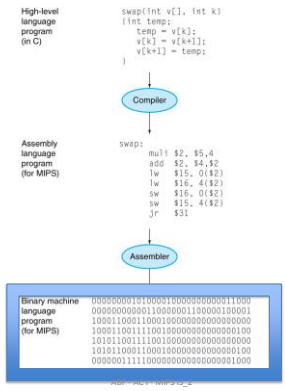
2s complement: Extensão do sinal

Representar um numero usando mais bits
No instruction set do MIPS
- `addi`: valor do imediato extendido a 32-bits
- `lb`, `lh`: extende o byte/halfword transferido da memória
- `beq`, `bne`: extende o valor do deslocamento

Replicar o bit de sinal para a esquerda
- c.f. unsigned values: extende com 0s

Exemplos: 8-bit para 16-bit
- +2: 0000 0010 => 0000 0000 0000 0010
- –2: 1111 1110 => 1111 1111 1111 1110

9. Representação das instruções



Codificação das instruções

Instruções codificadas em binário: ***código máquina***
Instruções MIPS
- Codificadas em 32-bits (***instruction words***)
  - – comprimento fixo
  - - Número reduzido de formatos de instrução
- Codificam: código de operação (***opcode***), número dos registos, …
- Regularidade!

Número dos registos
- $t0 – $t7 são os registos 8 – 15
- $t8 – $t9 são os registos 24 – 25
- $s0 – $s7 são os registos 16 – 23

Formatos de instrução: Tipo R

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- Instruction fields
  - – op: operation code (opcode)
  - – rs: first source register number
  - – rt: second source register number
  - – rd: destination register number
  - – shamt: shift amount (00000 for now)
  - – funct: function code (extensão do opcode)

Formato Tipo R: exemplo

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

```
add $t0, $s1, $s2
```

special	$s1	$s2	$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

$00000010001100100100000000100000_2 = 02324020_{16}$

Formatos de instrução: Tipo I

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

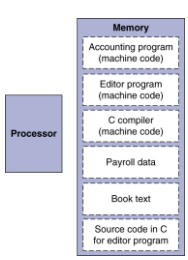
- Instruções load/store e operações com imediatos
  - rt: destination or source register number
  - Constant: -2^{15} to $+2^{15} - 1$
  - Address: offset added to base address in rs

Codificação das instruções: Tipo R e Tipo I

Instruction	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32_{ten}	n.a.
sub (subtract)	R	0	reg	reg	reg	0	34_{ten}	n.a.
add immediate	I	8_{ten}	reg	reg	n.a.	n.a.	n.a.	constant
lw (load word)	I	35_{ten}	reg	reg	n.a.	n.a.	n.a.	address
sw (store word)	I	43_{ten}	reg	reg	n.a.	n.a.	n.a.	address

Tipo R – instruções aritméticas e lógicas
Tipo I – loads, stores e operações com imediatos

The BIG Picture



- Instruções representadas em binário, tal como os dados
- Instruções e dados armazenados na memória
- Programas podem operar sobre programas
  - compilers, linkers, assembler…
- Compatibilidade Binária permite aos programas compilados serem executados em diferentes computadores
  - Standardized ISAs

Operações Lógicas

- Instruções para manipulação de bits

Operação	C	Java	MIPS
Shift left	<<	<<	`sll`
Shift right	>>	>>>	`srl`
Bitwise AND	&	&	`and, andi`
Bitwise OR	|	|	`or, ori`
Bitwise NOT	~	~	`nor`

  - Uteis para extrair e inserir grupos de bits numa *word*

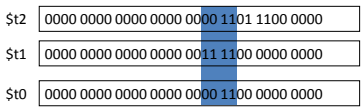
Operações de deslocamento (***shift***)

- shamt: indica quantas posições deslocar
- Shift left logical
  - Shift left e preencher com zeros
  - `sll` *i* bits multiplica por 2^i
- Shift right logical
  - Shift right preencher com zeros
  - `srl` *i* bits divide por 2^i (<u>só unsigned</u>)

AND

- Útil para mascarar (selecionar) bits numa *word*
  - Seleciona alguns bits, coloca os outros a 0

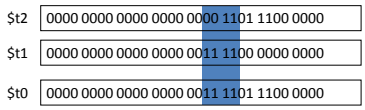
```
and $t0, $t1, $t2
```



OR

- Util para incluir bits numa *word*
  - Coloca alguns bits a 1, os restantes não se alteram

```
or $t0, $t1, $t2
```

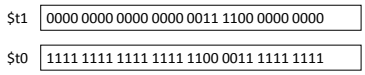


NOT

- Util para inverter bits numa *word*
  - Muda 0 para 1, e 1 para 0
- MIPS tem NOR 3-operand instruction
  - a NOR b == NOT (a OR b)

```
nor $t0, $t1, $zero
```

Register 0: always read as zero



10. Instruções de escolha – operações condicionais

Operações condicionais (***branch***)

Salta para a instrução indicada se a condição é verdadeira
Senão, continua sequencialmente

`beq rs, rt, L1`
if (rs == rt) branch to instruction labeled L1;

`bne rs, rt, L1`
if (rs != rt) branch to instruction labeled L1;

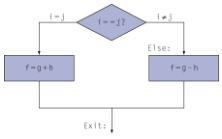
`j L1`
unconditional jump to instruction labeled L1

Compilação de If

- Código C:

```
if (i==j) f = g+h;
else f = g-h;
```

  – f, g, … em $s0, $s1, …



- Código Compilado MIPS:

```
      bne $s3, $s4, Else
      add $s0, $s1, $s2
      j   Exit
Else: sub $s0, $s1, $s2
Exit: …
```

Assembler calcula endereços

Compilação de ciclos

Código C:

```
while (save[i] == k) i += 1;
```

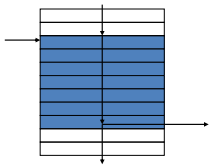
i em $s3, k em $s5, endereço de save em $s6

Código Compilado MIPS:

```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: …
```

Basic Blocks

- basic block - sequência de instruções sem
  - branches (exceto no fim)
  - branch *targets* (exceto no início)



- O compilador identifica *basic blocks* para optimização
- O processador pode acelerar a execução dos *basic blocks*

Outras operações condicionais

- Set result to 1 if a condition is true
  - Otherwise, set to 0
- `slt rd, rs, rt`
  - if (rs < rt) rd = 1; else rd = 0;
- `slti rt, rs, constant`
  - if (rs < constant) rt = 1; else rt = 0;
- Usado em combinação com `beq`, `bne`

```
slt $t0, $s1, $s2  # if ($s1 < $s2)
bne $t0, $zero, L  #   branch to L
```


Comparações ***signed*** e ***unsigned***

- Signed comparison: `slt`, `slti`
- Unsigned comparison: `sltu`, `sltui`
- Exemplo:
  - $s0 = 1111 1111 1111 1111 1111 1111 1111 1111
  - $s1 = 0000 0000 0000 0000 0000 0000 0000 0001
  - `slt  $t0, $s0, $s1  # signed`
    - –1 < +1 ⇒ $t0 = 1
  - `sltu $t0, $s0, $s1  # unsigned`
    - +4,294,967,295 > +1 ⇒ $t0 = 0

11. Invocação de funções/procedures (sub-rotinas)

Invocação de procedimentos

Passos necessários:

1. Colocar os parâmetros em registos
2. Transferir o controlo para o procedimento
3. Adquirir espaço de memória para o procedimento
4. Executar as instruções do procedimento
5. Colocar o resultado num registo para o passar ao invocador
6. Regressar ao ponto do programa onde foi feita a chamada do procedimento

Utilização dos registos

- $a0 – $a3: argumentos (r4 – r7)
- $v0, $v1: valores do resultado (r2 e r3)
- $t0 – $t9: temporaries
  - O seu conteúdo pode ser destruído pelo ***callee***
- $s0 – $s7: saved
  - Têm de ser preservados (saved/restored) pelo ***callee***
- $gp: global pointer for static data (reg 28)
- $sp: stack pointer (reg 29)
- $fp: frame pointer (reg 30)
- $ra: return address (reg 31)

Instruções para invocação de procedimentos

- Procedure call: jump and link
  `jal ProcedureLabel`
  - Endereço da instrução seguinte colocado em $ra
  - Salta para o endereço alvo
- Procedure return: jump register
  `jr $ra`
  - Copia $ra para o program counter **(PC) = ($ra)**
  - Pode tambem ser usado para "computed jumps"
    - e.g., case/switch statements

Procedimento que não invoca outro (*leaf*)

- Código C :

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

  - Argumentos **g, …, j** em $a0, …, $a3
  - **f** em $s0 (necessário preservar $s0 no stack)
  - Resultado em $v0

`leaf_example`: código MIPS

`addi $sp, $sp, -4` `sw $s0, 0($sp)`	Guardar $s0 no stack

`add $t0, $a0, $a1`
`add $t1, $a2, $a3`
`sub $s0, $t0, $t1`
	Corpo do procedimento

`add $v0, $s0, $zero`	Resultado em $v0
`lw $s0, 0($sp)` `addi $sp, $sp, 4`	Restaurar $s0
`jr $ra`	Regresso ao ***caller***

Procedimentos que invocam outros procedimentos

- Para invocações em cadeia o ***caller*** precisa de guardar no stack:
  - O seu endereço de retorno
  - Valores de argumentos e variáveis temporárias de que necessite depois da invocação
- Restaurar o stack quando o procedimento que invocou retorna

Exemplo 2 – código C

- C:

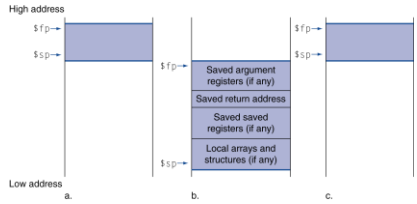
```
int fact (int n)
{
  if (n < 1) return f;
  else return n * fact(n - 1);
}
```

  - Argumento **n** em $a0
  - Resultado em $v0

Exemplo 2 – código MIPS

```
fact:
    addi $sp, $sp, -8     # ajusta o stack para 2 items
    sw   $ra, 4($sp)      # save return address
    sw   $a0, 0($sp)      # save argument
    slti $t0, $a0, 1      # teste se n < 1
    beq  $t0, $zero, L1
    addi $v0, $zero, 1    # if so, resultado = 1
    addi $sp, $sp, 8      #   pop 2 items do stack
    jr   $ra              #   e return
L1: addi $a0, $a0, -1     # else decrement n
    jal  fact             # recursive call
    lw   $a0, 0($sp)      # restore valor original de n
    lw   $ra, 4($sp)      #   e return address
    addi $sp, $sp, 8      # pop 2 items from stack
    mul  $v0, $a0, $v0    # multiplicar to get result
    jr   $ra              # return
```

Dados locais no stack



Local data allocated by callee
- e.g., C automatic variables

Procedure frame (activation record)
- Usado por alguns compiladores para administrar o stack

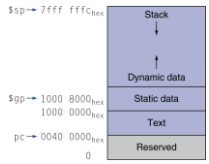
Mapa de memória

Text: program code

Static data: global variables
- e.g., static variables in C, constant arrays and strings
- $gp initialized to address allowing ±offsets into this segment

Dynamic data: heap
- E.g., ***malloc*** em C, ***new*** em Java

Stack: automatic storage



12. Representação de outros tipos de dados

Carateres

- Byte-encoded character sets
  - ASCII: 128 carateres
    - 95 graphic, 33 control
  - Latin-1: 256 carateres
    - ASCII, +96 more graphic characters
- Unicode: 32-bit character set
  - Usado em Java, C++ wide characters, …
  - Codifica a maior parte dos alfabetos exitentes mais os símbolos
  - UTF-8, UTF-16: variable-length encodings

Operações com ***bytes*** e ***halfwords***

- Podem usar operações bitwise
- MIPS byte/halfword load/store
  - String processing

```
lb rt, offset(rs)     lh rt, offset(rs)
```

  - Sign extend to 32 bits in rt

```
lbu rt, offset(rs)    lhu rt, offset(rs)
```

  - Zero extend to 32 bits in rt

```
sb rt, offset(rs)     sh rt, offset(rs)
```

  - Store just rightmost byte/halfword

Exemplo: String copy

- C:
  - Null-terminated string

```
void strcpy (char x[], char y[])
{ int i;
  i = 0;
  while ((x[i]=y[i])!='\0')
    i += 1;
}
```

  - Endereços de x, y em $a0, $a1
  - i em $s0

String copy: código MIPS

```
strcpy:
    addi $sp, $sp, -4      # ajustar stack para 1 item
    sw   $s0, 0($sp)       # save $s0
    add  $s0, $zero, $zero # i = 0
L1: add  $t1, $s0, $a1     # addr of y[i] in $t1
    lbu  $t2, 0($t1)       # $t2 = y[i]
    add  $t3, $s0, $a0     # addr of x[i] in $t3
    sb   $t2, 0($t3)       # x[i] = y[i]
    beq  $t2, $zero, L2    # exit loop if y[i] == 0
    addi $s0, $s0, 1       # i = i + 1
    j    L1                # next iteration of loop
L2: lw   $s0, 0($sp)       # restore saved $s0
    addi $sp, $sp, 4       # pop 1 item from stack
    jr   $ra               # return
```


13. Modos de endereçamento de *branch* e *jump*

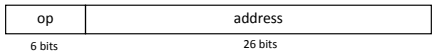
Endereçagem de ***branch***

- As instruções de *branch* especificam
  - Opcode, dois registos, target address
- A maioria dos branch saltam para instruções próximas
  - Forward or backward

  - Endereçagem PC-relative
    - Target address = PC + offset × 4
    - PC já a apontar para a instrução seguinte (previamente incrementado de 4)

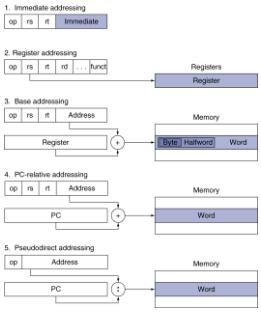
Endereçagem de ***jump***

- Jump (`j` and `jal`) targets podem estar em qualquer posição do segmento de texto (não existe a limitação dos *branch*) – endereço completo no código de instrução



- (Pseudo)Direct jump addressing
  - Target address = $PC_{31...28}$: (address × 4)

Modos de Endereçamento: sumário



Target Addressing: exemplo

– Assume-se que Loop em 80000

```
Loop: sll  $t1, $s3, 2     80000
      add  $t1, $t1, $s6   80004
      lw   $t0, 0($t1)     80008
      bne  $t0, $s5, Exit  80012
      addi $s3, $s3, 1     80016
      j    Loop            80020
Exit: …                    80024
```

0	0	19	9	4	0	
0	9	22	9	0	32	
35	9	8	0			
5	8	21	2			
8	19	19	1			
2	20000					

Instruções MIPS já vistas

MIPS instructions	Name	Format	Pseudo MIPS	Name	Format
add	add	R	move	move	R
subtract	sub	R	multiply	mult	R
add immediate	addi	I	multiply immediate	multi	I
load word	lw	I	load immediate	li	I
store word	sw	I	branch less than	blt	I
load half	lh	I	branch less than or equal	ble	I
load half unsigned	lhu	I	branch greater than	bgt	I
store half	sh	I	branch greater than or equal	bge	I
load byte	lb	I			
load byte unsigned	lbu	I			
store byte	sb	I			
load linked	ll	I			
store conditional	sc	I			
load upper immediate	lui	I			
and	and	R			
or	or	R			
nor	nor	R			
and immediate	andi	I			
or immediate	ori	I			
shift left logical	sll	R			
shift right logical	srl	R			
branch on equal	beq	I			
branch on not equal	bne	I			
set less than	slt	R			
set less than immediate	slti	I			
set less than immediate unsigned	sltiu	I			
jump	j	J			
jump register	jr	R			
jump and link	jal	J			