Conjuntos de instruções de microprocessadores

João Canas Ferreira

Novembro de 2015

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Tópicos de
Arquitetura e Organização de Computadores

Tópicos

Dois princípios

- Os computadores atuais seguem dois princípios-chave:
  1. Instruções são representadas como números;
  2. Programas (sequências de instruções) são guardados em memória, tal como dados.
- Programas podem ser fornecidos como ficheiros (de dados binários): os dados são as instruções do programa.
- Esses programas podem ser executados em computadores que aceitem o mesmo conjunto de instruções: *compatibilidade binária*.
- Um programa (A) também pode ser executado por outro programa (V), que *interpreta* as instruções de A: V é um *simulador* ou uma *máquina virtual*.
- **Questão:** Como codificar as instruções?
  - critérios (tipos de instruções, tipos de dados, modelo de execução)
  - formatos

Código-máquina e código assembly

- O código de um programa pode ser representado por números: *código-máquina*.

Exemplo (em hexadecimal, MIPS-32):

```
8D2804B0
02484020
AC2804B0
```

- Código simbólico para instruções (mnemónicas): *assembly code*

O mesmo exemplo:

```
lw    $t0, 1200($t1)
add   $t0, $s2, $t0
sw    $t0, 1200($t1)
```

- Conversão de código *assembly* para código-máquina também é feita por um programa: *assembler*
- Cada microprocessador tem o seu código-máquina, definido pela especificação do processador.

Modelo de programação

- O modelo de programação de um microprocessador é definido por:
  1. modelo de execução
  2. conjunto de instruções
     1. classes (ou tipos) de instruções
     2. modos de especificação de operandos (endereçamento)
  3. registos
     1. de uso geral
     2. dedicados (de uso específico)
- Modelo de execução:
  1. inicializar PC *(program counter)*
  2. obter instrução da posição PC da memória
  3. executar instrução e atualizar PC
  4. repetir a partir de 2
- PC é um registo dedicado.

Classes de instruções

- As classes de instruções mais comuns são:

1. Operações aritméticas com números inteiros
   - adição, subtração, multiplicação, divisão
2. Operações lógicas sobre conjuntos de bits
   - AND, OR, NOR, deslocamentos *(shift)*
3. Transferências de dados
   - leitura e escrita de dados em memória
4. Alteração do fluxo (sequencial) de execução
   - saltos condicionais e comparações
   - saltos incondicionais
   - execução de subrotinas

- As instruções de salto alteram o PC.
- Existem muitas alternativas…

Modos de endereçamento

- Modos de endereçamento = modos de especificação dos operandos

Os mais comuns são:

1. **imediato:** o valor (constante) está incluído na instrução.
2. **registo:** o valor está num registo; a instrução inclui a especificação do registo.
3. **direto:** a instrução inclui o endereço da posição de memória.
4. **indireto** (via registo): o registo contém o endereço da posição de memória onde está o valor; a instrução especifica o registo.
5. **indireto** com deslocamento constante: instrução especifica registo e um valor constante: a posição de memória é obtida por soma do valor constante com o conteúdo do registo.
   (É uma generalização da categoria anterior.)
6. **relativo ao PC**: a instrução inclui constante a adicionar ao valor de PC.

Classificação segundo a origem dos operandos

Mem.	Max. ops.	Arquitetura	Exemplos
0	3	reg-reg	Alpha, MIPS, SPARC
1	2	reg-mem	IBM 360/370, Intel 80x86
2	2	mem-mem	VAX
3	3	mem-mem	VAX

Tipo	Vantagens	Desvantagens
reg-reg	Codificação simples, comprimento único. Geração de código simplificada. Duração similar.	Número de instruções elevado. Programas mais compridos.
reg-mem	Acesso a dados sem "load" em separado. Tendem a ter boa densidade de codificação.	Operandos não são equivalentes. Duração varia com a localização dos operandos. Pode restringir o número de registos codificáveis.
mem-mem	Programas compactos. Não ocupa registos com resultados temporários.	Comprimento de instruções muito variável. Complexidade de instruções muito variável. Acesso a memória é crítico.

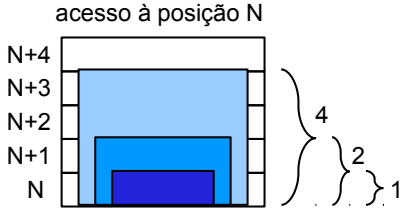
- As duas principais caraterísticas que dividem arquiteturas com registos de uso genérico são:
  1. número de operandos: 2 ou 3;
  2. quantos operandos podem residir em memória (de 0 a 3).

Tipos de operandos

- Tipos comuns de operandos:
  1. números inteiros de:
     - 4 bytes (1 palavra)
     - 2 bytes (meia palavra)
     - 1 byte
  2. números de vírgula flutuante:
     - 4 bytes (precisão simples)
     - 8 bytes (precisão dupla)

- A interpretação dos dados e o seu tamanho são definidos pela instrução usada para os processar. O programador e/ou o compilador são responsáveis pela utilização coerente das instruções.

- Endereço de memória do item especifica a posição do primeiro byte.



- Regras de *alinhamento* típicas:
  - palavra: só endereços múltiplos de 4
  - meia palavra: só endereços múltiplos de 2
  - byte: qualquer endereço

Caraterísticas das instruções MIPS-32

- Organização *reg-reg* (RISC)
- Acesso a memória: apenas *load* (leitura) e *store* (escrita)
- Instruções lógicas e aritméticas com 3 registos (2 operandos e 1 resultado)
- Conjunto de instruções "ortogonal"
  - Onde pode ser usado um registo, pode ser usado qualquer outro.
- Todas as instruções têm 32 bits de comprimento
- Memória endereçável: 2^{30} palavras (2^{32} bytes)
- 32 registos de uso geral (0-31) de 32 bits: `$0`, `$1`, etc.

Registo `$0` (`$zero`) tem *sempre* o valor zero.

- Convenções de utilização (para garantir interoperabilidade):
  - `$t0`-`$t9` e `$s0`-`$s7`: uso sem restrições [registos 8 a 25].
  - `$at`: reservado para *assembler* [registo 1].
  - Todos os outros registos (salvo `$0`) estão reservados para implementação de subrotinas, para o sistema operativo ou para conterem o endereço de zonas especiais de memória.
- Dois registos especiais (multiplicações e divisões): `$lo` e `$hi`.

Subconjunto de instruções MIPS-32 (I)

Instrução	Exemplo	Significado
add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3
subtract	`sub $s1,$s2,$s3`	$s1 = $s2 − $s3
add immediate	`addi $s1,$s2,20`	$s1 = $s2 + 20
and	`and $s1,$s2,$s3`	$s1 = $s2 AND $s3
or	`or $s1,$s2,$s3`	$s1 = $s2 OR $s3
and immediate	`andi $s1,$s2,20`	$s1 = $s2 AND 20
or immediate	`ori $s1,$s2,20`	$s1 = $s2 OR 20
shift left logical	`sll $s1,$s2,10`	$s1 = $s2 << 10
shift right logical	`srl $s1,$s2,10`	$s1 = $s2 >> 10
load word	`lw $s1,20($s2)`	$s1 = Mem[$s2 + 20]
store word	`sw $s1,20($s2)`	Mem[$s2 + 20] = $s1
load half	`lh $s1,20($s2)`	$s1 = Mem[$s2 + 20]
load half unsigned	`lhu $s1,20($s2)`	$s1 = Mem[$2 + 20]
store half	`sh $s1,20($s2)`	Mem[$s2 + 20] = $s1
load upper immediate	`lui $s1,20`	$s1 = 20 × 2^{16}

Subconjunto de instruções MIPS-32 (II)

Instrução	Exemplo	Significado
branch on equal	`beq $s1,$s2,25`	se ($s1 = $s2) PC = PC + 4 + 4×25
branch on not equal	`bne $s1,$s2,25`	se ($s1 ≠ $s2) PC = PC + 4 + 4×25
set less than	`slt $s1,$s2,$s3`	se ($s2 < $s3) $s1=1 senão $s1 = 0
set less than unsigned	`sltu $s1,$s2,$s3`	se ($s2 < $s3) $s1=1 senão $s1 = 0
set less than immediate	`slti $s1,$s2,20`	se ($s2<20) $s1=1 senão $s1=0
jump	`j 2500`	saltar para 2500×4 (PC=10000)
jump register	`jr $s1`	saltar para valor de $s1 (múltiplo de 4)

- As instruções começam *sempre* em posições cujos endereços são **múltiplos de 4**.

Subconjunto de instruções MIPS-32 (III)

Instrução	Exemplo	Significado
multiplicação (com sinal)	`mult $s1,$s2`	{$hi,$lo} = $s1 × $s2
multiplicação (sem sinal)	`multu $s1,$s2`	{$hi,$lo} = $s1 × $s2
divisão (com sinal)	`div $s1,$s2`	$lo = quociente de $s1/$s2 $hi = resto da divisão
divisão (sem sinal)	`divu $s1,$s2`	$lo = quociente de $s1/$s2 $hi = resto da divisão
cópia de $hi	`mfhi $s1`	$s1 = $hi
cópia de $lo	`mflo $s1`	$s1 = $lo

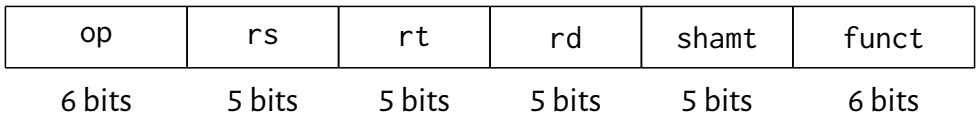
- O resultado da multiplicação de dois números de 32 bits tem 64 bits.

Os bits *menos significativos* do produto ficam em `$lo`, os *mais significativos* em `$hi`.

- A divisão de dois números inteiros de 32 bits produz um quociente e um resto, ambos de 32 bits.

Instruções lógicas e aritméticas

- Usam o **formato R** para instruções com 3 registos



- *op:* Código básico da operação **(opcode)**.
- *rs:* O primeiro operando (número do registo).
- *rt:* O segundo operando (número do registo).
- *rd:* O destino do resultado (número do registo).
- *shamt:* sempre zero, excepto para instruções de deslocamento *(shift amount)*.
- *funct:* Código da função, que especifica a variante da operação.

add	op=0, funct=20_{16}	and	op=0, funct=24_{16}
sub	op=0, funct=22_{16}	or	op=0, funct=25_{16}
slt	op=0, funct=$2A_{16}$		

Codificação de instruções lógicas e aritméticas

- Exemplo: add `$t0,$s1,$s2`

Números dos registos: `$t0`: 8, `$s1`: 17, `$s2`: 18

0	17	18	8	0	32
op	rs	rt	rd	shamt	funct

Em binário:

000000	10001	10010	01000	00000	100000
op	rs	rt	rd	shamt	funct

- Código máquina: 0x02324020 (o mesmo que 02324020_{16} ou 02324020_{H}).

Codificação de constantes imediatas

- Algumas instruções lógicas ou aritméticas (e outras) requerem um formato com dois registos e uma constante: o **formato I**.

op	rs	rt	constante
6 bits	5 bits	5 bits	16 bits

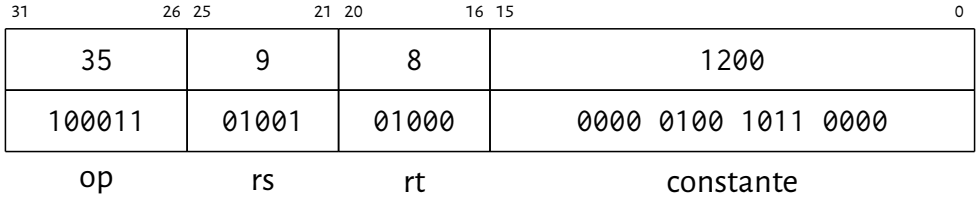
- *op:* Código da operação **(opcode)**.
- *rs:* Um registo (número do registo).
- *rt:* Outro registo (número do registo).
- *constante:* Valor imediato.

- Para instruções que usam valores com sinal (como `addi`) ou para *endereços* (`lw` e `sw`), o valor imediato é em complemento para 2.

addi	op=8	lw	op=23_{16}
ori	op=D_{16}	sw	op=$2B_{16}$
slti	op=A_{16}	lui	op=$0F_{16}$

Instruções de leitura de memória

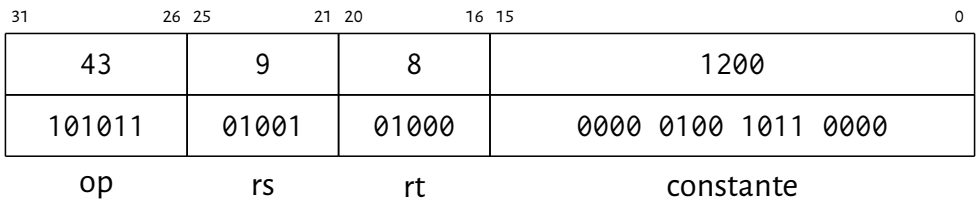
- Estas instruções usam o **formato I**.
- Exemplo: **`lw $t0,1200($t1)`**
- A constante (aqui: 1200) é interpretada como o deslocamento (com sinal) a somar ao valor do registo de endereçamento (aqui: `$t1`).
- Atenção: para acessos a palavras, o endereço final deve ser múltiplo de 4. Para acessos a meias palavras (`lh`), o endereço final deve ser múltiplo de 2.
- Número dos registos: `$t0`: 8, `$t1`: 9. **O campo `rs` é usado para o endereço de base.**



- Instrução `lh` faz expansão de sinal da meia-palavra lida de memória (para preencher o registo de destino, que é de 32 bits).
- Instrução `lhu` acrescenta 16 zeros à esquerda da meia-palavra lida.

Instruções de escrita em memória

- Estas instruções usam o **formato I**.
- Exemplo: **`sw $t0,1200($t1)`**
- A constante (aqui: 1200) é interpretada como o deslocamento (com sinal) a somar ao valor do registo de endereçamento (aqui: `$t1`).
- Atenção: para acessos a palavras, o endereço final deve ser múltiplo de 4. Para acessos a meias palavras (`sh`), o endereço final deve ser múltiplo de 2.
- Número dos registos: `$t0`: 8, `$t1`: 9. **O campo `rs` é usado para o endereço de base.**



- Instrução **sh** guarda os 16 bits menos significativos do registo especificado na memória.

Instruções de deslocamento

- Instruções de deslocamento movem os bits de um registo para a direita ou para a esquerda.
- As posições livres são preenchidas com 0.
- Estas instruções usam o formato R, geralmente com `shamt` $\neq 0$.
- Exemplo: **`sll $t2,$s0,4`**
- Estas instruções usam o **formato R**.
- Número dos registos: `$t2`: 10,`$s0`:16

0	0	16	10	4	0
op	rs	rt	rd	shamt	funct

- Se `$s0` = 0000 0000 0000 0000 0000 0000 0000 1001, então
  `$t2` = 0000 0000 0000 0000 0000 0000 1001 0000
- Deslocamento de N posições para a esquerda multiplica um número positivo por 2^N. (Porquê?)

Saltos condicionais

- Instruções de comparação (ex. `slt`) usam o **formato R**.
- Instruções de salto usam o **formato I**.
  - O salto é relativo: deslocamento em **relação à instrução seguinte**.
  - O deslocamento é multiplicado por 4 (para contar instruções, que têm 4 bytes).
  - O deslocamento pode ser positivo ou negativo.
- Exemplo:

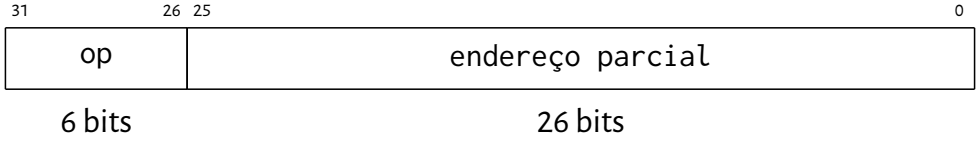
```
etiqueta: add ...
          slt $t0,$t1,$t2
          beq $t0,$zero, etiqueta
```

31 26	25 21	20 16	15 0
4	0	8	-3
000100	01000	00000	1111 1111 1111 1101
op	rs	rt	constante

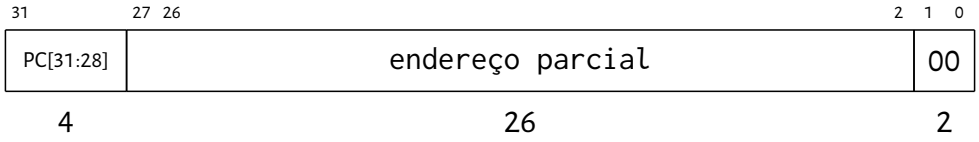
Instruções de salto incondicional

- A instrução **`jr`** usa o **formato R** com
  op=0, funct=8, rt=0, rd=0, rs=registo a usar.

- A instrução **`j`** (jump) usa o **formato J**.



- Endereço para onde salta (endereço do destino):



- Os 4 bits mais significativos do endereço do destino são iguais aos 4 bits mais significativos do PC.

Resumo dos modos de endereçamento MIPS-32

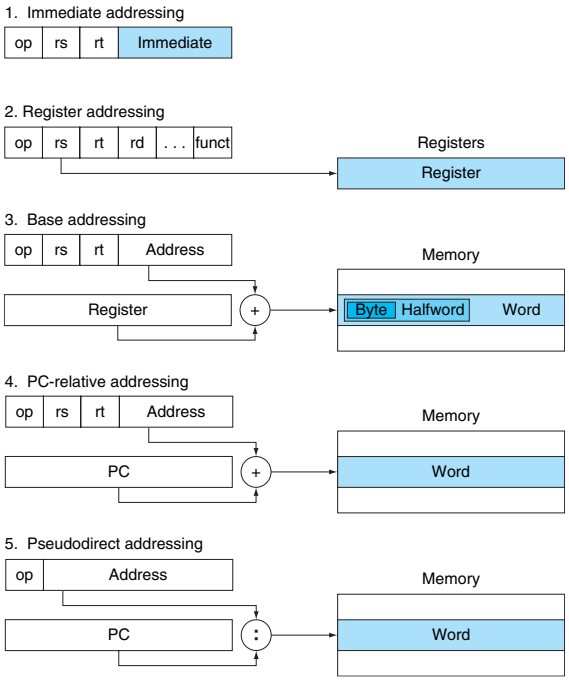


FIGURE 2.18 Illustration of the five MIPS addressing modes. The operands are shaded in color. The operand of mode 3 is in memory, whereas the operand for mode 2 is a register. Note that versions of load and store access bytes, halfwords, or words. For mode 1, the operand is 16 bits of the instruction itself. Modes 4 and 5 address instructions in memory, with mode 4 adding a 16-bit address shifted left 2 bits to the PC and mode 5 concatenating a 26-bit address shifted left 2 bits with the 4 upper bits of the PC.

Fonte: [COD4]

Fluxo (simplificado) de criação de programas

1. Preparar programa com editor de texto (1 ou mais ficheiros)
2. Invocar *assembler* para converter ficheiros para código-máquina
3. "Ligar" programa às subrotinas do sistema *(linker)*
4. Executar (talvez usando um emulador)
   O programa deve ser carregado previamente para memória *(loader)*
5. Depurar (voltar a 1)

➠ O que é preciso saber sobre o "sistema":

1. organização de memória (sistema operativo e aplicação)
2. onde fica colocado o código e as zonas de dados
3. subrotinas disponíveis (sistema ou bibliotecas de funções)
4. como invocar serviços do sistema operativo (se existir) e/ou
   como aceder a periféricos

➠ Emulador para MIPS-32: MARS
`http://courses.missouristate.edu/KenVollmar/MARS/`

Assembler

- Função principal: código *assembly* → código-máquina.
- Facilitar a programação:
  1. verificar a "legalidade" das instruções
     - sintaxe das instruções, tamanho das constantes, ...
  2. nomes para posições de memória: etiquetas
  3. reserva de zonas de memória para dados (alocação de memória)
  4. especificação de valores iniciais para zonas de memória
  5. síntese de instruções úteis (pseudo-instruções) ou de "sinónimos"
     - MIPS-32 não tem instrução para copiar um valor de um registo para outro, mas o *assembler* suporta a instrução

       `move RD, RS`

       que copia o conteúdo do registo RD para o registo RS.
  6. ajuste de saltos, dependendo da distância ao destino
  7. definir procedimentos para geração de grupos de instruções (macro-instruções)
     - O próprio *assembler* é programável!
- Para além do código-máquina, também pode produzir listagens anotadas do código gerado.

Pseudo-instruções

- Pseudo-instruções são convertidas pelo *assembler* em uma ou mais instruções nativas.
- Pseudo-instruções servem para colmatar uma desvantagem devida à simplicidade do conjunto de instruções.
- Exemplo:

```
move $t0, $t1
```

é convertida em:

```
add $t0, $zero, $t1
```

- Pseudo-instrução "branch on less than"

```
blt $t0, $t1, L
```

é convertida em:

```
slt $at,$t0,$t1
bne $at,$zero,L
```

- Por convenção, o registo `$at` está reservado para o *assembler*.

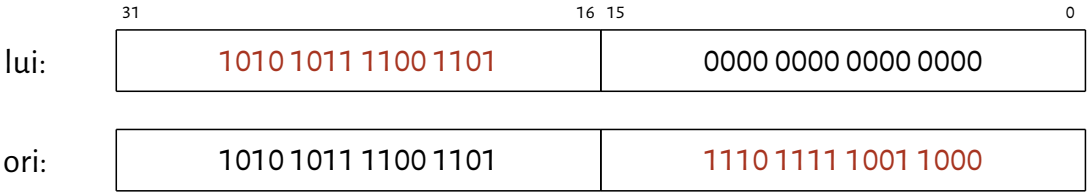
Constantes de 32 bits

- Porquê usar apenas valores imediatos de 16 bits?
  1. A maior parte das constantes que surge num programa tem um valor pequeno.
  2. A maior parte dos destinos dos saltos também está a curta distância.
  3. Uma instrução de 32 bits não tem "espaço" para mais (se necessitar de especificar vários registos).
- Instruções com valores imediatos de 16-bits são um *bom compromisso*.
- Solução para dados:

```
move $s0,0xABCDEF98
```

é transformada em:

```
lui $s0,0xABCD
ori $s0,$s0,0xEF98
```



Saltos a grandes distâncias

- Como efectuar saltos para distâncias superiores ao permitido pela codificação de endereços relativos em 16 bits?
- Gama normal de deslocamento: +32767 a -32768 instruções.
- Solução: o *assembler* converte o salto relativo em salto relativo + salto incondicional.

Para **L1** referente a uma posição fora da gama normal de deslocamentos, a instrução

```
beq $s0,$s1,L1
```

é transformada em:

```
    bne $s0,$s1,L2
    j   L1
L2: . . .
```

- Para distâncias ainda maiores, o *assembler* carrega o endereço de destino para um registo e usa a instrução **jr**.

Exemplos: expressões numéricas

- Código para calcular a expressão

```
f = (g + h) - (i + j)
```

- Atribuição de variáveis a registos: f, ...j →`$s0`, ..., `$s4`

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

- Código para calcular a expressão

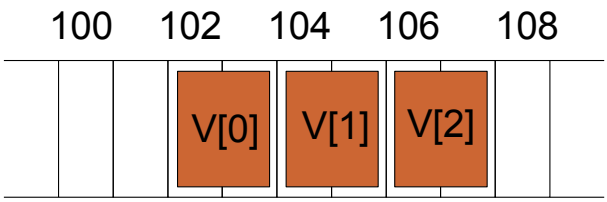
```
f = (g + h - 100) - (i + 320)
```

- Atribuição de variáveis a registos: f, ...i →`$s0`, ..., `$s3`

```
add  $t0, $s1, $s2
addi $t1, $s3, 320
addi $t0, $t0, -100
sub  $s0, $t0, $t1
```


Armazenamento de sequências em memória

- Sequências homogéneas:
  1. Sequência `A[]` com N elementos: `V[0]`, , ..., `V[N-1]`.
  2. Todos os elementos têm o mesmo tamanho S (em bytes).
  3. Uma sequência de N elementos, cada um de S bytes, ocupa N × S bytes.
  4. Cada elemento de uma sequência pode ser especificado pelo par de números (b, d):
     1. *endereço-base da sequência b*: endereço do primeiro elemento:
     2. *deslocamento d* dentro da sequência.
  5. O deslocamento associado a `V[i]` é: $d = i \times S$
- Exemplo: Disposição de uma sequência de 3 meias-palavras em memória:



- $b = 102$
- endereço de V[1]:

$$b + 1 \times 2 = 104$$

Exemplos: Acesso a memória

- Operandos em memória:

```
g = h + A[8]
```

- Atribuição de variáveis a registos: g →$s1, h →$s2, endereço-base de A →$s3

```
lw  $t0,32($s3)
add $s1,$s2, $t0
```

- Porque é que o deslocamento é 32?
- Código correspondente a:

```
A[12] = h + A[8]
```

- Atribuição de variáveis a registos: h →$s2, endereço-base de A →$s3

```
lw  $t0, 32($s3)
add $t0, $s2,$t0
sw  $t0, 48($s3)
```


Instruções condicionais

- Código correspondente a:

```
se (i = j)
        f = g + h
```

- Atribuição de variáveis a registos: f, g, ...j →$s0, $s1, ..., $s4

```
        bne   $s3, $s4, Cont
        add   $s0, $s1,$s2
Cont:   . . .
```

- Código correspondente a:

```
se (i = j)
        f = g + h
senão
        f = g - h
```

- Atribuição de variáveis a registos: f, g, ...j →$s0, $s1, ..., $s4

```
        bne   $s3, $s4, Alt
        add   $s0, $s1,$s2
        j     Cont
Alt:    sub   $s0, $s1, $s2
Cont:   . . .
```

Ciclos

- Código correspondente a:

```
enquanto (V[i] = k)
        i = i + 1
```

- Atribuição de variáveis a registos: i → $s3, k → $s5, base de V[] → $s6

```
Ciclo: sll   $t1, $s3, 2
       add   $t1, $t1,$s6
       lw    $t0, 0($t1)
       bne   $t0, $s5, Cont
       addi  $s3, $s3, 1
       j     Ciclo
Cont:  . . .
```

- Código-máquina (colocado em memória a partir de 80000):

80000:	0	0	19	9	2	0
80004:	0	9	22	9	0	32
80008:	35	9	8	0		
80012:	5	8	21	2		
80016:	8	19	19	1		
80020:	2	20000				
80024:	instrução seguinte					

Referências

COD4 D. A. Patterson & J. L. Hennessey, Computer Organization and Design, 4 ed.

MARS K. Vollmar, MARS (MIPS Assembler and Runtime Simulator), http://courses.missouristate.edu/KenVollmar/MARS/

Os tópicos tratados nesta apresentação são abordados nas seguintes secções de [COD4]:

- 2.1–2.3, 2.5–2.7, 2.10